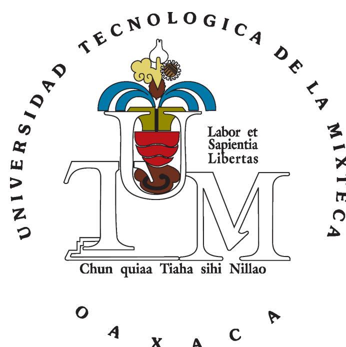

UNIVERSIDAD TECNOLÓGICA DE LA MIXTECA

DIVISIÓN DE ESTUDIOS DE POSGRADO

“IMPLEMENTACIÓN EN ERLANG DE SISTEMAS DISTRIBUIDOS A TRAVÉS DEL CÁLCULO DE EVENTOS”

T E S I S

Para obtener el grado de:

MAESTRO EN TECNOLOGÍAS DE CÓMPUTO APLICADO

Presenta:

ING. JOSÉ YEDID AGUILAR LÓPEZ

Directores de tesis:

DR. FELIPE DE JESÚS TRUJILLO ROMERO
DR. MANUEL HERNÁNDEZ GUTIÉRREZ

Huajuapan de León, Oaxaca, Méx. Marzo de 2014.

A mi familia, mi madre,
mi hermana, mis sobrinos
y a Rosario.

Agradecimientos

Mi primer agradecimiento es para las personas que dirigieron esta tesis: al Dr. Felipe de Jesús Trujillo Romero y al Dr. Manuel Hernández Gutiérrez, por toda la confianza, paciencia y apoyo que tuvieron para conmigo en todo este tiempo y así poder llevar acabo esta investigación. Fue gracias a las explicaciones, asesorías, discusiones, seguimiento y múltiples dudas que atendieron para poder realizar esto. Además de las revisiones y observaciones hechas al documento de tesis, ya ni decir de la gran calidad moral y sencillez humana que tienen.

Quiero agredecer también a los miembros del jurado que participaron en la revisión y presentación de mi tesis por su disposición, comentarios, críticas, recomendaciones y señalamientos valiosos que tuvieron para reforzar, mejorar y ajustar el trabajo de investigación, ellos son: la Dra. Lluvia Carolina Morales Reynaga, el Dr. Agustín Santiago Alvarado, el Dr. José Aníbal Arias Aguilar y el Dr. Santiago Omar Caballero Morales.

A la Jefatura de la División de Estudios de Posgrado a cargo del Dr. Agustín Santiago Alvarado por el constante apoyo y la actitud positiva mostrada a los estudiantes de posgrado. De igual manera mi más sincera gratitud a la Coordinadora Académica de la Maestría en Tecnologías de Cómputo Aplicado la Dra. Lluvia Carolina Morales Reynaga por su ayuda, cooperación y atención al desarrollo de esta tesis; por la excelente labor que desempeñan.

No puedo pasar por alto a los profesores que me impartieron clases, ya que compartieron con toda confianza su conocimiento académico y experiencia laboral, por sus consejos y enseñanzas; y a la Universidad Tecnológica de la Mixteca por haberme brindado la oportunidad de realizar mis estudios de maestría y así aprovechar la infraestructura que posee.

Al Consejo Nacional de Ciencia y Tecnología (Conacyt) por el patrocinio otorgado para concluir satisfactoriamente mis estudios de maestría y llevar a cabo el tema de tesis.

A mi familia y a Rosario por el incondicional apoyo, comprensión y alentadoras palabras, ya que siempre estuvieron ahí para mí.

A todos, muchas gracias: Yedid.

Resumen

El desarrollo de software es una actividad compleja que involucra el uso de procesos, metodologías, tecnologías, modelado de sistemas, prácticas de programación, entre otros, los cuales evolucionan y surgen nuevos debido a necesidades de la época: programación directa en hardware, automatización de cálculos científicos y financieros, aplicaciones para Internet, reúso de software, cómputo distribuido y las aplicaciones para dispositivos móviles, son unos casos; además de la necesidad de crear software cada vez más diverso, robusto y complejo. El campo que estudia esto es la ingeniería de software, una disciplina que se ocupa de todos los aspectos de la producción de software.

Entre las actividades fundamentales que se deben realizar en esta ingeniería, independientemente de la metodología que se siga, se encuentra la especificación de software, donde se definen las funcionalidades y limitaciones que tendrá el software apoyándose en notaciones gráficas, matemáticas y lógicas principalmente. Esta tesis de maestría se sitúa en este contexto, se propone una notación para el modelado de sistemas distribuidos y concurrentes con paso de mensajes, los cuales se caracterizan por el uso de procesos y las formas en que estos se comunican. La notación podrá ser utilizada en la etapa de especificación de estos sistemas y también en la etapa de implementación, ya que posee una correspondencia unidireccional con la programación concurrente del lenguaje funcional Erlang.

El modelado con esta notación será para aquellos procesos que se comuniquen a través del paradigma paso de mensajes, está basada en el formalismo lógico cálculo de eventos y a diferencia de otras notaciones, toma en cuenta la variable tiempo para indicar de forma explícita cuándo ejecutar la creación de nuevos procesos, el envío de mensajes y la recepción de estos entre procesos. Una distinción más con notaciones como las Redes de Petri o Álgebras de Procesos (CCS, CSP, ACP, Cálculo Pi) es que estas todavía carecen de una correspondencia unidireccional con la sintaxis de Erlang que permita dar la pauta para programar el sistema.

Se ha elegido a Erlang como plataforma de implementación debido a que está totalmente orientado al desarrollo de estos sistemas distribuidos y concurrentes, en contraste (incluso

la comparación es injusta) con otros lenguajes de propósito general como Pascal, C, C++, Java o Ruby, en los que también se pueden implementar pero se realiza con los procesos nativos del sistema operativo en los que hay cuidar la memoria compartida y evitar bloqueos, o con sus máquinas virtuales que demandan más capacidad que la de Erlang, ya que este emplea procesos ligeros y los gestiona prescindiendo de los nativos librándose así de problemas conocidos cuando los procesos comparten memoria. Además este lenguaje de programación está preparado para el uso de procesos a gran escala, existen aplicaciones y proyectos en la industria del software que utilizan este lenguaje, por ejemplo en aplicaciones web (la base de datos distribuida *Amazon SimpleDB*), en mensajería móvil (*Whatsapp*, *Facebook*) o en sistemas de telecomunicaciones (*T-Mobile*).

Así, la base de la notación está dada en el cálculo de eventos, que se emplea para el razonamiento acerca de acciones que ocurren en el mundo real y sus efectos. Estas acciones y efectos se describen en forma narrativa tomando en cuanta el factor tiempo para conocer cuándo ocurren. La notación que se plantea se llama ProME (*notation for Process Modeling Erlang*), se definen a detalle sus elementos y sintaxis a considerar para el modelado de estos sistemas, asimismo se describe cómo se da la correspondecia hacia la sintaxis de la programación concurrente de Erlang, consiguiendo una interpretación directa entre la notación basada en el cálculo de eventos y Erlang.

Las contribuciones de la tesis se dan en tres sentidos: primero, la aplicación del cálculo de eventos, como un subsistema de la lógica, para la especificación de sistemas distribuidos y concurrentes con paso de mensajes; segundo, proveer una notación, como herramienta teórica, que auxilie las etapas de especificación e implementación de este tipo de sistemas; y tercero, constituir una relación entre el cálculo de eventos y la sintaxis de Erlang que facilite la programación de estos sistemas.

Abstract

Software development is a complex activity that involves the use of processes, methodologies, technologies, systems modeling, programming practices, among others, which evolve and emerge new due to needs of the time: direct programming hardware, automation scientific and financial calculations, Internet applications, software reuse, distributed computing and mobile applications are some cases; addition to the need for increasingly diverse, robust and complex software. The field that study this is software engineering, a discipline that deals with all aspects of software production.

Key activities to be performed in this engineering, regardless of the methodology to be followed, is the specification of software, where the functionalities and limitations that will have the software, supported mainly by graphical notations, math and logic are defined. This master thesis is in this context, a notation for modeling concurrent and distributed systems with message passing, which are characterized by the use of processes and the ways in which they communicate is proposed. The notation may be used in the specification stage of these systems and in the implementation phase, as it has a unidirectional correspondence with the concurrent programming of the functional language Erlang .

Modeling with this notation will be for those processes that communicate via message passing paradigm, is based on the event calculus logical formalism and unlike other notations, consider the time variable to indicate explicitly when to run the creation new processes, sending and receiving messages between these processes. One more distinction with notations such as Petri Nets and Process Algebras (CCS, CSP, ACP, Pi Calculus) is that they still lack a unidirectional correspondence with Erlang syntax that allows to set the guideline for the programing of the system.

It has chosen Erlang as implementation platform because it is totally oriented to the development of these distributed and concurrent systems, in contrast (even the comparison is unfair) with other general purpose languages such as Pascal, C, C++, Java or Ruby, where you can also implement but is done with the native operating system processes where there

are caring shared memory and avoid locks, or your virtual machines require more ability than Erlang, since this uses lightweight processes and manages waged regardless of native and known issues when processes sharing memory. Besides this programming language is prepared for the use of large-scale processes, there are applications and projects in the software industry that use this language, for example in web applications (distributed database *Amazon SimpleDB*), mobile messaging (*Whatsapp*, *Facebook*) or telecommunications systems (*T-Mobile*).

Thus the basis of the notation is given in the event calculus, which is used for reasoning about actions that occur in the real world and its effects. These actions and effects are described in narrative form, taking into account the time factor for when they occur. The notation that arises is called ProME (*notation for Process Modeling Erlang*) are defined in detail its syntax elements to consider for modeling these systems, also describes how the correspondences is given to the syntax of the concurrent Erlang programming, achieving a direct interpretation between notation based on the event calculus and Erlang.

The contributions of the thesis are given in three ways: first, the application of the event calculus, as a subsystem of the logic for the specification of distributed and concurrent systems with message passing; second, provide a notation, as a theoretical tool that helps in the steps of specification and implementation of such systems; and thirdly, provide a relationship between the event calculus and Erlang syntax, thus to facilitate programming of these systems.

Índice general

Resumen **vii**

Abstract **ix**

Índice de figuras **xvi**

Índice de tablas **xvii**

1. Introducción **1**

1.1. La especificación del software 3
1.2. Problemática 6
1.3. Justificación 8
1.4. Hipótesis del trabajo 9
1.5. Objetivos de la tesis 9
1.5.1. Objetivo general 9
1.5.2. Objetivos específicos 10
1.6. Estructura de la tesis 10

2. El cálculo de eventos **11**

2.1. Antecedentes: el cálculo de eventos y agentes 11
2.2. Programación lógica 12
2.2.1. Programas lógicos 13
2.2.2. Unificación de términos 14
2.3. Introducción al cálculo de eventos 15
2.4. El cálculo de eventos de Kowalski y Sergot 17
2.5. El cálculo de eventos simplificado 18
2.6. Ajuste hacia el cálculo de eventos 20
2.6.1. Inicio y terminación de fluentes mediante eventos 22

2.6.2. Interpretaciones del cálculo de eventos 24

3. Sistemas distribuidos 27

3.1. Importancia de los sistemas distribuidos 27
3.2. Definición de un sistema distribuido 28
3.2.1. Características de un sistema distribuido 29
3.2.2. Aplicaciones de los sistemas distribuidos 30
3.3. Arquitecturas de sistemas distribuidos 31
3.3.1. Cliente/Servidor 32
3.3.2. Peer-to-peer 33
3.4. Modelos de comunicación de procesos 35
3.5. Especificación de sistemas distribuidos 36

4. Programación concurrente en Erlang 39

4.1. Introducción 39
4.2. Características de Erlang 40
4.3. La concurrencia y procesos en Erlang 42
4.4. Paradigmas de comunicación de procesos 44
4.4.1. Memoria compartida con bloqueo 44
4.4.2. Memoria transaccional de software 45
4.4.3. Paso de mensajes 46
4.5. Trabajando con procesos en Erlang 46
4.5.1. Creación de procesos 47
4.5.2. Envío de mensajes 49
4.5.3. Recepción de mensajes 50
4.5.4. Registro de procesos 53
4.5.5. Ejemplos de procesos Erlang 54
4.6. Arquitectura Cliente/Servidor 57

5. Una notación para la programación concurrente 61

5.1. Las bases de la notación 61
5.1.1. Programación concurrente 62
5.1.2. El cálculo de eventos 63
5.2. Acercando el cálculo de eventos a Erlang 64
5.2.1. Asignaciones a los eventos 64
5.2.2. El servidor como proceso 65
5.2.3. El cliente como proceso 72

5.3. ProME, una notación para la modelación de procesos Erlang 75
5.3.1. Elementos de la notación ProME 75
5.3.2. Sintaxis de los elementos de la notación ProME 78
5.4. Metodología para emplear la notación ProME 85
5.5. Ejemplo de narrativas ProME 86
5.6. Caso de estudio: un sistema enseñanza-aprendizaje en línea 89
5.6.1. Descripción del problema 89
5.6.2. Tipos de usuarios 90
5.6.3. Modelación con la notación ProME 92
5.6.4. Implementación en Erlang 96

6. Conclusiones y trabajo futuro 99
6.1. Resultados 99
6.2. Conclusiones 100
6.3. Trabajo futuro 101

Apéndices 103

A. Erlang básico 105
A.1. Programando en Erlang 105
A.2. El intérprete de comandos de Erlang 106
A.3. Tipos de datos 107
A.3.1. Numéricos 107
A.3.2. Átomos 108
A.3.3. Booleanos 109
A.3.4. Tuplas 109
A.3.5. Fecha y hora 109
A.3.6. Listas 110
A.3.7. Cadenas 111
A.3.8. Registros 112
A.3.9. Identificador de procesos 113
A.3.10. Variables 113
A.4. Tipos de operadores 115
A.4.1. Operadores aritméticos 115
A.4.2. Operadores lógicos 116
A.4.3. Operadores relacionales 116
A.4.4. Operadores a nivel de bits 117

A.4.5. Precedencia de operadores 118
A.5. Coincidencia de patrones de Erlang 118
A.6. Funciones como datos: *fun* 120

Bibliografía **123**

Índice alfabético **129**

Índice de figuras

1.1. Actividades fundamentales de la ingeniería de software 2
1.2. Actividades del proceso de ingeniería de requisitos 3
1.3. Elementos principales de la tesis y sus relaciones 7
1.4. El cálculo de eventos y Erlang 8

2.1. Representación básica de la granularidad del tiempo. 20
2.2. Evaluación de predicados conforme transcurre el tiempo. 21
2.3. Predicados afectados por la ocurrencia de eventos. 21
2.4. Inicio y terminación de la veracidad de un fluente. 23
2.5. Un evento originando más de un fluente. 23
2.6. Un evento terminando más de un fluente. 24
2.7. Cómo funciona el cálculo de eventos. 25

3.1. Los procesos con roles Cliente o Servidor 32
3.2. Interacción entre Cliente y Servidor 33
3.3. Arquitectura Peer-to-peer 34
3.4. Ejemplo de la ejecución de eventos en los procesos 36

4.1. Ejecución de procesos Erlang en el hardware 43
4.2. Lanzando un proceso Erlang 48
4.3. Paso de mensajes entre procesos Erlang 49
4.4. Eco de un mensaje 56
4.5. Arquitectura Cliente/Servidor mediante procesos Erlang 58
4.6. Ejemplo de petición y respuesta en la arquitectura Cliente/Servidor 60

5.1. Conceptos claves de Erlang y sus relaciones 63
5.2. Correspondencia entre la narrativa y Erlang para el servidor 71
5.3. Correspondencia entre la narrativa y Erlang para el cliente 76
5.4. Autómata finito determinista de las narrativas de la notación ProME 83

5.5. Gramática de la notación ProME en la forma Backus Naur extendida 84

Índice de tablas

A.1. Operadores aritméticos en Erlang. 116
A.2. Operadores lógicos en Erlang. 116
A.3. Operadores relacionales en Erlang. 117
A.4. Operadores aritméticos a nivel de bits en Erlang. 118
A.5. Precedencia de operadores en Erlang. 118

Capítulo 1

Introducción

Actualmente, el costo cada vez menor de hardware, los avances en la tecnología de las telecomunicaciones, el crecimiento explosivo de Internet y la dependencia cada vez mayor de las redes de computadoras para una amplia gama de aplicaciones que van desde comunicación social hasta transacciones financieras, han hecho que los *sistemas distribuidos* tengan mayor presencia en la vida cotidiana [Gho07]. Un sistema distribuido es uno que se ejecuta en varias computadoras, a diferencia de los *sistemas centralizados* donde todos los componentes del sistema se ejecutan en una sóla computadora [Som11]. La ingeniería de sistemas distribuidos tiene mucho en común con la ingeniería de otros sistemas[1], sin embargo hay aspectos específicos a considerar cuando se construyen este tipo de sistemas debido a que sus componentes pueden ejecutarse sobre computadoras administradas de forma independiente y se comunican a través de una red, estos aspectos son [CDKB11]:

- *Recursos compartidos*, un sistema distribuido permite compartir recursos de hardware y software como discos, impresoras, archivos y bases de datos.
- *Apertura*, los sistemas distribuidos son normalmente sistemas abiertos, lo que significa que permiten combinar diferentes tipos de computadoras y software de diversos vendedores.
- *Concurrencia*, en un sistema distribuido varios procesos pueden operar al mismo tiempo y comunicarse o no durante su operación.

[1] Existen tipos de software bien identificados, algunos son [Pre09]: software de sistemas, software de aplicaciones, software de ingeniería, software empotrado, software basado en web y software de inteligencia artifical.

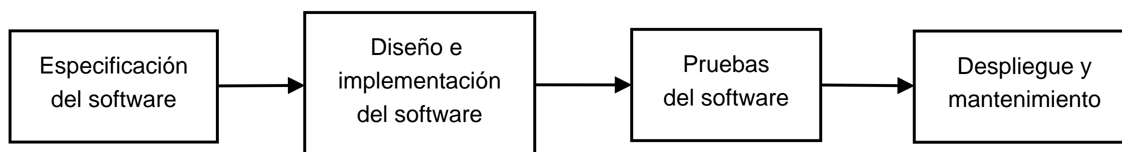

Figura 1.1: Actividades fundamentales de la ingeniería de software

- *Escalabilidad*, los sistemas distribuidos pueden aumentar sus capacidades mediante la adición de recursos para hacer frente a las nuevas demandas del sistema.
- *Tolerancia a fallas*, los sistemas distribuidos pueden ser tolerantes a algunos fallos de hardaware y software, esto se da con la disponibilidad de varias computadoras y la posibilidad de replicar información.

Una de las cosas que sí tienen en común los sistemas distribuidos con los otros tipos de sistemas es que para su desarrollo incluyen al menos cuatro actividades fundamentales de la ingeniería de software, independientemente de la metodología que se siga [Som11, Pre09]:

1. *Especificación del software*, se definen las funcionalidades y restricciones de su operación, los requisitos que debe tener.
2. *Diseño e implementación del software*, se construye el software que debe satisfacer la especificación.
3. *Pruebas del software*, se asegura que el software cumple con la especificación planteada antes de ser entregado al cliente.
4. *Despliegue y mantenimiento*, el software es instalado y puesto en marcha para que el cliente evalúe y utilice el producto. El mantenimiento implica la corrección de errores que no fueron señalados en las etapas anteriores así como la mejora de la implementación y de servicios del sistema a medida que se descubran nuevos requisitos, cuando esto último se presente habría que especificarlo también.

Estas actividades por su naturaleza se realizan en secuencia, aunque puede variar de acuerdo a la metodología que se use, por ejemplo intercalándose, sin embargo siempre serán consideradas de alguna forma. El diagrama de la Fig. 1.1 muestra estas actividades secuencialmente. En las cuatro actividades se debe notar la importancia de la especificación del software, ya que de ahí se desprenden las actividades posteriores. La misma fase de especificación es tan delicada, ya que una mala interpretación del problema derivará en implementaciones

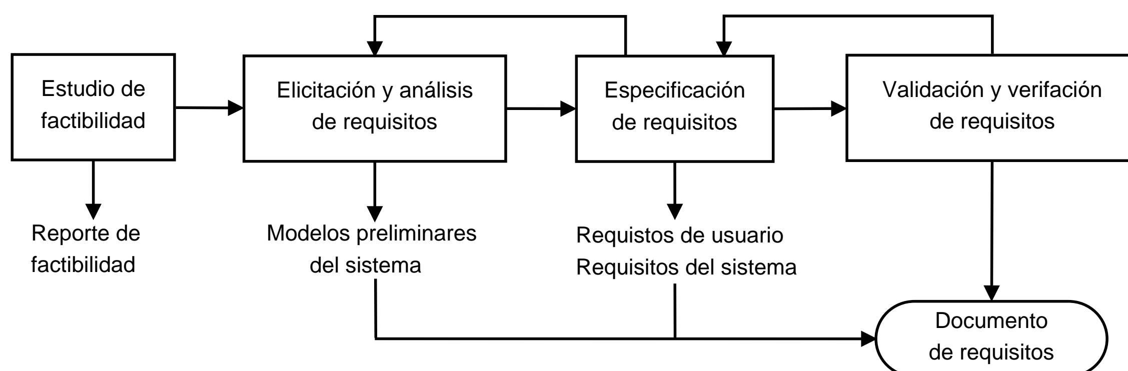

Figura 1.2: Actividades del proceso de ingeniería de requisitos

no deseadas, aún cuando no existan errores de ejecución del software esto no satisfacería al cliente puesto qué eso no es lo que quiere.

Esta tesis se encuentra enmarcada en este contexto, el de la especificación de software para sistemas distribuidos que se comunican a través del paradigma paso de mensajes. Con el fin de ahondar un poco más en esta actividad básica de la ingeniería de software, sumado al de atañer con el problema de tesis, se presenta el siguiente apartado y después se describe la propuesta de trabajo para dicha actividad en el caso de los sistemas distribuidos.

1.1. La especificación del software

El área de investigación que comprende y estudia la actividad de especificación del software es la *ingeniería de requisitos* [RR06, HJD05], se describe como el proceso de comprensión y definición de los requisitos que deberá tener el sistema, y la identificación de sus limitaciones de funcionamiento y de desarrollo, es decir, la definición de lo que se quiere obtener como producto final. La ingenería de requisitos es una etapa crítica, ya que los errores en esta etapa conducen inevitablemente a problemas posteriores en el diseño e implementación del sistema.

El proceso de la ingeniería de requisitos tiene como objetivo producir un *documento de requisitos*, el cual especifica un sistema que satisface las necesidades de las partes interesadas (*stakeholders*). Los requisitos se presentan por lo general en dos niveles de detalle: un nivel es para los usuarios finales y clientes, que se da en una declaración de alto nivel de los requisitos; y el otro, es para los desarrolladores del sistema, una especificación más detallada. El diagrama de la Fig. 1.2 presenta las actividades para llevar a cabo el proceso de ingeniería de requisitos así como sus salidas respectivas. Enseguida se explican estas actividades:

1. *Estudio de factibilidad.* Se revisa si las necesidades de los usuarios identificados pueden satisfacerse utilizando el software y tecnologías de hardware actuales. El estudio también analiza si el sistema propuesto será rentable desde el punto de vista empresarial y si se puede desarrollar dentro de las limitaciones presupuestarias existentes. Un estudio de factibilidad debe ser relativamente barato y rápido. El resultado de esta actividad es un reporte que contenga la decisión de si se debe o no continuar con el resto de las actividades del desarrollo.

2. *Elicitación y análisis de requisitos.* Este es el proceso de obtención de los requisitos del sistema a través de la observación de los sistemas existentes, las conversaciones con el cliente y los posibles usuarios, análisis de los procesos y tareas actuales, entre otros más. Esto puede implicar el desarrollo de uno o varios modelos de sistemas y prototipos, que ayuden a entender el sistema a especificar.

3. *Especificación de requisitos.* Es la actividad de traducir la información recopilada en la actividad anterior en un documento que define un conjunto de requisitos. Dos tipos de requisitos pueden ser incluidos en este documento: los *requisitos de usuario* y los *requisitos del sistema*. El primer tipo de requisito son declaraciones abstractas de las funcionalidades del sistema que va dirigido al cliente y al usuario final; el segundo tipo de especificación de requisitos son una descripción más detallada de las funcionalidades y está orientado principalmente a los diseñadores, desarrolladores, personal de prueba y mantenimiemto del sistema así como también a los analistas mismos y, a los administradores y líderes del proyecto.

4. *Validación y verificación de requisitos.* La validación involucra la revisión de los requisitos de usuario, buscando que refleje exactamente las necesidades de los clientes y de todos los demás involucrados. Mientras que en la verificación se revisa que los requisitos del sistema se ajusten a los requisitos de usuario, además de comprobar criterios como la consistencia y la completitud de los requisitos. En conclusión con la validación se asegura de que se construya el sistema correcto, mientras con la verificación se asegura de que se construya el sistema de forma correcta.

El documento de requisitos (algunas veces nombrado *Especificación de Requisitos de Software*) es la salida del proceso de ingeniería de requisitos y lo realizan los analistas de sistemas. Este documento servirá, en primera instancia, como una de las entradas a la actividad de diseño e implementación del software. Los requisitos de usuario se expresan en lenguaje natural con vocabulario no técnico y pueden también apoyarse en diagramas o tablas que fortalezcan

el entendimiento de estos. Por otro lado, los requisitos del sistema también puede escribirse en lenguaje natural y con los mismos tipos de diagramas o tablas que se usan para los requisitos de usuario, pero casi siempre es necesario emplear *notaciones* especializadas, que inevitablemente marcan la pauta para iniciar el diseño del sistema. Estas notaciones pueden ser [Som11]:

- *Plantillas o formularios*, donde los requisitos son escritos en lenguaje natural y cada campo proporciona información acerca de uno o más aspectos de los requisitos.
- *Lenguajes de descripción de diseño*, este enfoque utiliza un lenguaje como si se tratara de un lenguaje de programación, pero con características abstractas para definir el modelo operacional del sistema.
- *Notaciones gráficas*, son formas gráficas que se suelen suplementar con anotaciones de texto para definir los requisitos funcionales del sistema. Un ejemplo de esto son los diagramas de casos de uso y de secuencia del Lenguaje Unificado de Modelado (*UML*), [OMG11].
- *Especificaciones matemáticas*, estas notaciones se basan en conceptos lógicos-matemáticos, tales como las máquinas de estados finitos o la teoría de conjuntos. Estas especificaciones pueden reducir la ambigüedad de los requisitos del sistema.

Cabe mencionar que no es exclusivo usar tal o cual notación sino en función del problema a resolver es posible elegir una combinación de estas; y que la acción de especificar requisitos con alguna notación también se le denomina a menudo *modelado*. Una ventaja que se le puede explotar a una notación es que algunas están preparadas para permitir una traducción a la implementación (programación) parcial del sistema, lo que ayuda a los programadores a proporcionarles un marco de trabajo y al cual habrá que agregar los detalles de programación de las funcionalidades del sistema [HJD05].

Un ejemplo común de lo anterior se da con los los diagramas de clases de UML, en el desarrollo de software orientado a objetos, ya que es posible interpretar los diagramas a algún lenguaje de programación orientado a objetos como C++ o Java, esto en base al conocimiento uniforme del paradigma orientado a objetos. En estos libros [Mar03, SM02] se muestran varios ejemplos de ello. Incluso esta interpretación actualmente se realliza a menudo de forma automática utilizando un software que toma como entrada los diagramas de clases y basándose en la sintaxis del lenguaje de programación obtiene el código correspondiente del diagrama.

Una vez expuesto lo anterior, en esta tesis se propone una notación para el modelado de las funcionalidades de un sistema distribuido y concurrente. Esta notación será para aquellos procesos que se comuniquen a través del paradigma paso de mensajes que a diferencia de otras notaciones (como las Redes de Petri o las Álgebras de Procesos CCS, CSP, ACP o Cálculo Pi) toma en cuenta la variable tiempo para indicar de forma explícita cuándo ejecutar la creación de nuevos procesos, el envío de mensajes y la recepción de estos entre procesos, puesto que está basada en el formalismo lógico cálculo de eventos. Una distinción más, es que dichas notaciones todavía carecen de una correspondencia unidireccional con la sintaxis de programación concurrente de Erlang que permita dar la pauta para programar el sistema.

La notación propuesta encaja en la clasificación de lenguajes de descripción de diseño y tiene las bases para poder estar en la de especificaciones matemáticas.

Se ha elegido al lenguaje de programación Erlang como plataforma de implementación debido a que está totalmente orientado al desarrollo de estos sistemas distribuidos y concurrentes, en contraste (incluso la comparación es injusta) con otros lenguajes de propósito general como Pascal, C, C++, Java o Ruby, en los que también se pueden implementar pero que se realizan con los procesos nativos del sistema operativo en los que hay cuidar la memoria compartida y evitar bloqueos, o con sus máquinas virtuales que demandan más capacidad que la de Erlang, ya que este emplea procesos ligeros y los gestiona prescindiendo de los nativos librándose así de problemas conocidos cuando comparten memoria los procesos. También este lenguaje de programación está preparado para el uso de procesos a gran escala que a medida de la demanda, la memoria que utilizan los procesos se ajusta para tener más espacio o se libera espacio.

1.2. Problemática

Las notaciones para modelar y diseñar sistemas de software ayudan a la especificación del mismo y pueden hacerlo también en las etapas de diseño, implementación y pruebas. Así, la notación que se propone abarca las siguientes áreas de conocimiento:

1. el cálculo de eventos,

2. los sistemas distribuidos y

3. el lenguaje de programación Erlang.

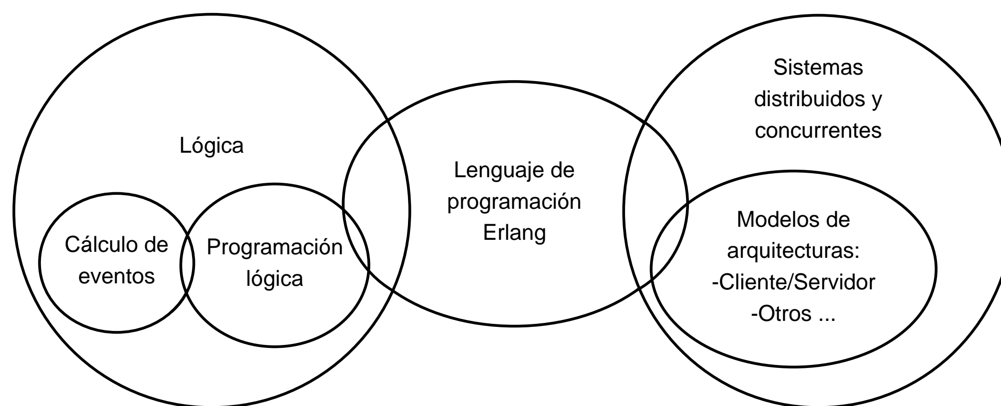

Figura 1.3: Elementos principales de la tesis y sus relaciones

Estas áreas de conocimiento tienen ciertas relaciones en común. La Fig. 1.3 trata de mostrar estas relaciones. Se parte de la idea de que las áreas de conocimiento de la lógica y la de los sistemas distribuidos y concurrentes son sistemas que tienen subsistemas y que pueden relacionarse (intersección) con otros sistemas. Entre los sistemas con los que se relacionan se encuentra el lenguaje de programación Erlang, ya que está inspirado en la programación lógica (a su vez subsistema de la lógica) y fue creado para desarrollar sistemas distribuidos altamente concurrentes, en el que es posible implementar arquitecturas comunes que tienen los sistemas distribuidos como la de Cliente/Servidor. Así también se aprecia el subsistema cálculo de eventos como parte de la lógica y su relación con la programación lógica ya que originalmente fue formulado como un programa lógico.

Entonces, el problema que se plantea resolver en esta tesis es la obtención de una variante del cálculo de eventos, para que con ello sea posible especificar sistemas distribuidos con paso de mensajes mediante una notación lógica-matemática y una semántica, con el objetivo de construir programas distribuidos en Erlang. Esta adaptación del cálculo de eventos tendrá fundamentos lógicos que se aprovecharán para relacionarla unidireccionalmente con la sintaxis del lenguaje de programación funcional Erlang. Haciendo así un puente conceptual entre la notación propuesta y la implementación de sistemas distribuidos con paso de mensajes. Esto auxiliará la especificación y programación de sistemas distribuidos en Erlang, obteniendo estos programas desde la notación.

Para guiarnos en el desarrollo y aplicación de nuestros resultados se desarrollará el sistema propuesto como caso de estudio en [CGH+09], donde se planteó un sistema distribuido de enseñanza y aprendizaje teniendo como usuarios a estudiantes y profesores.

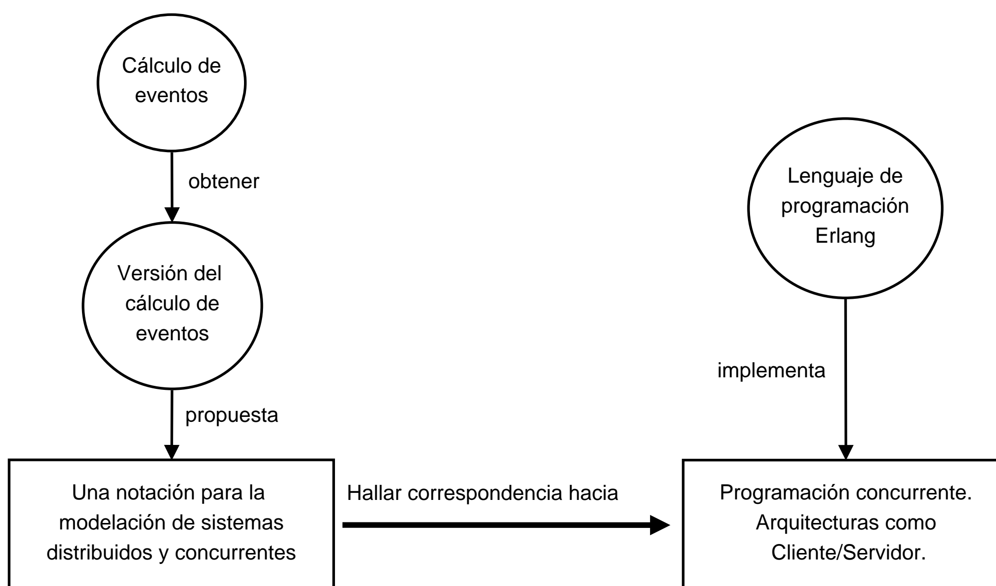

Figura 1.4: El cálculo de eventos y Erlang

Con la notación propuesta y la correspondencia unidireccional hallada con la sintaxis de Erlang para *programación concurrente*, sería posible entonces una transformación automática de la notación propuesta a código Erlang. Dicha transformación queda fuera de esta tesis y será planteado como trabajo futuro.

La propuesta de tesis en general se representa en el diagrama de la Fig. 1.4.

1.3. Justificación

La programación de sistemas distribuidos es una tarea sujeta a cometer errores dada su complejidad. Esto en particular es aún más cierto cuando se habla de varias computadoras ejecutando alguna tarea concurrentemente. Es así necesario contar con herramientas que permitan allanar el terreno para lograr que la programación sea éxitosa. Entre estas herramientas se deben contar con lenguajes de programación adecuados, una especificación suficiente del sistema, y la interpretación entre estas especificaciones y las implementaciones correspondientes.

Este trabajo atacará el problema de la programación concurrente en sistemas distribuidos partiendo de modelos de un conjunto de fórmulas lógicas; estos modelos se implementan vía la ejecución de programas en Erlang, un lenguaje de programación orientado a cómputo distribuido; las fórmulas lógicas están basadas en las teorías de cláusulas de Horn y el cálculo

de eventos. La mecanización así como la visualización de esta traducción es posible, al menos para casos específicos y de un nivel mediano de complejidad, lo cual formará parte de las aportaciones.

1.4. Hipótesis del trabajo

La hipótesis principal de este trabajo es que el cálculo de eventos es una herramienta apropiada para la especificación de sistemas concurrentes. Por ello, es posible conseguir una notación basada en el cálculo de eventos para especificar sistemas distribuidos con paso de mensajes, que podrá interpretarse hacia la programación concurrente del lenguaje Erlang. Esto auxiliará la programación distribuida así como el análisis lógico del sistema.

Por lo que la principal parte metodológica de esta tesis es *la aplicación del cálculo de eventos, como un subsistema de la programación lógica, al análisis y síntesis de programas en Erlang que implementan sistemas distribuidos basados en paso de mensajes*; y la principal *contribución* científica y tecnológica esperada de la culminación de esta tesis es la obtención de herramientas teóricas basadas en la lógica para obtener programas que implementen sistemas distribuidos.

1.5. Objetivos de la tesis

Los objetivos de este trabajo son indicados a continuación:

1.5.1. Objetivo general

Establecer una notación para la especificación de sistemas distribuidos concurrentes con paso de mensajes a través de una adaptación del cálculo de eventos. Consiguiendo también una correspondencia unidireccional desde notación del cálculo de eventos hacia la programación concurrente que emplea el lenguaje de programación Erlang, de modo que permita la traducción entre la especificación (creada a partir de la notación) y la implementación en Erlang.

1.5.2. Objetivos específicos

1. Hallar un cálculo de eventos adecuado a la naturaleza del comportamiento de los sistemas distribuidos con paso de mensajes.
2. Proponer una notación para la especificación de sistemas distribuidos basandosé en el tipo de cálculo de eventos hallado.
3. Establecer limitaciones y alcances de la notación de cálculo de eventos obtenida.
4. Constituir la sintaxis que tendrá la notación del cálculo de eventos propuesta.
5. Encontrar la correspondencia unidireccional desde la notación propuesta del cálculo de eventos a la programación concurrente del lenguaje Erlang.
6. Verificar las salidas en Erlang dadas las especificaciones del cálculo de eventos.
7. Modelar un sistema de enseñanza y aprendizaje como caso de estudio empleando la notación de cálculo de eventos y obtener los programas en Erlang correspondiente.

1.6. Estructura de la tesis

Una vez sentada la problemática así como la solución que se pretende desarrollar, el resto de los capítulos que conforman este documento de tesis son: el capítulo 2, donde se define el formalismo lógico del cálculo de eventos y sus diversas variaciones así como sus fundamentos lógicos. Enseguida se detalla la teoría de los sistemas distribuidos y concurrentes en el capítulo 3. Lo refente a la concurrencia de sistemas distribuidos con el lenguaje de programación Erlang se aborda en el capítulo 4. Luego, las aportaciones del trabajo de tesis describiendo la propuesta de solución de acuerdo a los objetivos planteados inicialmente se presentan en el capítulo 5. Los resultados y conclusiones finales del trabajo de tesis, así como las líneas futuras de investigación se exponen en el capítulo 6. Por último se encuentra el apéndice A que describe las características básicas del lenguaje de programación Erlang.

Capítulo 2

El cálculo de eventos

El *cálculo de eventos* es un formalismo lógico que puede ser útil para representar la ocurrencia de eventos del mundo real. En esta tesis será una herramienta principal, ya que se pretende obtener una adaptación de este para conseguir una especificación que posea una correspondencia unidireccional con programas concurrentes del lenguaje Erlang, es decir, con la sintaxis y programación de procesos concurrentes de Erlang, que se aborda a detalle en el capítulo 4. Por lo que en este apartado se describe la teoría de dicho formalismo así como una breve introducción a sus fundamentos lógicos.

2.1. Antecedentes: el cálculo de eventos y agentes

Los eventos en el cálculo de eventos [KS86] son elementos indivisibles de un conjunto no vacío $\mathcal{E}$. Este conjunto tiene como característica principal el determinar el valor de verdad de elementos de otro conjunto no vacío $\mathcal{F}$, conjunto de fluentes. La idea inicial de los autores del artículo citado es realizar una descripción lógica de un *mundo* que percibe dinámicamente un *agente*; al que se le llamará agente A.

Este agente fue descrito teniendo dos capacidades esenciales: la capacidad de percibir y alterar su entorno (reactividad) y de razonar acerca de este entorno. En particular, el agente se consideró capaz de percibir eventos y de "saber" cómo estos eventos influyen en su ambiente.

En el principal componente reactivo del agente descrito, el supuesto básico fue que el agente debe percibir a los eventos ocurriendo en el tiempo; para esto, el evento debería monitorear cómo los fluentes cambiarían de valor booleano cuando los eventos ocurrieran.

Los siguientes supuestos están considerados en el planteamiento del agente mencionado arriba:

1. No todos los fluentes necesitan ser monitoreados constantemente para verificar su cambio de valor booleano; y

2. los niveles del razonamiento estarían condicionados por cierta capacidad restringida en el consumo de recursos computacionales (en particular el agente tendría que considerar necesariamente que sus reacciones requerirían un tiempo adecuado para producirse);

Casi al mismo tiempo en que surge el cálculo de eventos también aparace la propuesta de Rodney Brooks [Bro86] en la cual argumenta que la racionalidad era básicamente innecesaria para la producción de inteligencia, y que en todo caso bastarían agentes con un amplio abanico de tratamiento de percepciones para generar una respuesta adecuada al ambiente, haciendo que la inteligencia racional no se necesitará. En este aspecto, se habló de algo llamado *capas de control*, con lo que se pretendía modelar reactivamente al cerebro humano argumentando que las necesidades de sobrevivencia pueden jerarquizarse en diversos niveles de respuesta: las respuestas eleboradas requerirían algún grado de elaboración, mientras las simples estarían en un nivel automatizado, monótono y conservador. El cálculo de eventos tiene relación con la programación lógica ya que fue formulado como un programa lógico [Mue08], en la siguiente sección se describe la programación lógica.

2.2. Programación lógica

El cálculo de eventos se considera un formalismo (esquema) lógico basado en una concepción general de axiomas cuyas instancias darían razonamientos a nivel material para el tratamiento de problemas que se caracterizan por su conjunto de eventos y su conjunto de fluentes. Uno de los aspectos principales de este formalismo fue el estar basado en la *programación lógica* [KS86]. Este paradigma lógico nació como un producto de las investigaciones en la demostración automática de teoremas y los algoritmos de resolución en los años 60's y 70's del siglo pasado. Las ideas principales fueron algunas *fórmulas lógicas* llamadas *cláusulas*, que eran especialmente adecuadas para realizar demostraciones automáticas vía *refutaciones* y utilización de *términos*.

Así por ejemplo, la fórmula lógica llamada *cláusula de Horn* es una implicación que se compone de una conjunción de literales en el antecedente de la implicación llamado el *cuerpo*

de la cláusula y de un consecuente consistente de un átomo, la *cabeza de la cláusula*. La fórmula es la siguente:

$$(p \wedge q \wedge \cdots \wedge t) \rightarrow u$$

Una *literal* es un átomo o la negación de un átomo; un átomo es un predicado junto con sus respectivos términos. Un *término* es una constante tomado de un conjunto dado no vacío, una variable, o bien una expresión del tipo:

$$f(t_1, \ldots, t_n)$$

con $n > 0$ y tal que n es llamado la *aridad del functor* f. Un *programa lógico* se tomó como una sucesión finita de cláusulas de Horn. Los programas lógicos tienen dos puntos de vista en su lectura: en la lectura *procedural* se tomaron como programas convencionales al estilo de C o de Pascal, orientándose a una interpretación ejecutable, siendo estos programas vistos en sus implementaciones en Prolog (lenguaje basado en la programación lógica como programas convencionales de índole genérica). La siguiente interpretación de un programa lógico fue en la versión *declarativa*. En esta versión, los programas lógicos son considerados como un conjunto de aserciones lógicas acerca de un conjunto de términos. Esta versión resultó atractiva para razonar con los programas lógicos, y llevo a considerar posibilidades de llevar a cabo consecuencias de planteamientos axiomáticos hasta ese momento sólo posibles de llevar a cabo mediante traducción desde los programas tradicionales especializados [Dij75, Hoa69].

2.2.1. Programas lógicos

Para explicar los programas lógicos, se aborda esto con un modelo especial, se trata del *modelo de Herbrand* [NCdW97]. Tiene la ventaja particular de tener una *base de Herbrand* conformada por elementos (términos) extraidos del mismo programa lógico. Por ejemplo, para el siguiente programa lógico:

Programa lógico 1

$$true \Rightarrow nat(0) \tag{2.1}$$

$$nat(X) \Rightarrow nat(s(X)) \tag{2.2}$$

describiría lógicamente (vía una función biyectiva) a los números naturales. Es tradicional llamar *hechos* a las cláusulas del tipo 2.1 y además invertir el orden usual de la implicación para obtener el programa lógico siguiente:

Programa lógico 2

$$nat(0) \leftarrow true \tag{2.3}$$

$$nat(s(X)) \leftarrow nat(X) \tag{2.4}$$

Inclusive, es común omitir parte *true* de la cláusula 2.3 dejando al programa lógico en su versión más simplificada:

Programa lógico 3

$$nat(0) \leftarrow \tag{2.5}$$

$$nat(s(X)) \leftarrow nat(X) \tag{2.6}$$

2.2.2. Unificación de términos

Se muestra ahora la definición de términos, cuya principal ventaja es incrementar la expresividad de los programas lógicos:

1. Una constante es un término;
2. una variable es un término;
3. una expresión del tipo:
$$f(t_1, t_2, \ldots, t_n)$$
es un término si cada t_i de $i = 1$ hasta n ($n > 0$) es un término;
4. ninguna otra cosa es un término (propiedad de cerradura de términos).

Algunos ejemplos de términos son:

- 'a', término constante carácter
- 2, término constante numérico
- x, una variable
- $x + y$, un término tipo expresión aritmética
- `suma(multiplicación(2,3),4)`, otra forma de expresar la expresión aritmética $2*3+4$

- `auto(motor, volante, llantas(4))`, una posible definición de automóvil como un término, notando que `llantas(4)` es a su vez un término de aridad 1.

Para visualizar el incremento de expresividad mencionado, se considera ahora la construcción de los números naturales como términos:

Definición 1 (Números naturales, definición usual) *Un número natural se define como un elemento del conjunto*

$$\{1, 2, 3, 4, 5, \ldots\}$$

Sea 0 un símbolo, coincidentemente utilizado para representar el número 0 pero se deja abierto esta posibilidad sin comprometerse con ningún significado, y la expresión del tipo $s(_)$, en donde ha propósito se ha dejado sin argumento, pero que tiene como aridad 1, considerar la siguiente sucesión:

$$0, s(0), s(s(0)), s(s(s(0))), s(s(s(s(0)))), \ldots$$

Lo que se ha hecho es básicamente construir los números naturales desde un punto de vista lógico, haciendo que el $s(0)$ sea 1, $s(s(0))$ sea 2, $s(s(s(0)))$ sea 3, $s(s(s(s(0))))$ sea 4, y así sucesivamente. La expresión s se llama símbolo de función de la función *sucesor*, y es la manera en que el lógico matemático italiano Peano "creó" los números naturales (junto con otros axiomas que se omiten) [Ham82]. Se puede considerar al número 0 como el "caso base" y a la descripción "$s(x)$ es un número natural si x es un número natural", o bien, como fórmula lógica:

$$nat(0) \wedge (nat(x) \Rightarrow nat(s(x)))$$

Considérese ahora la pregunta de si $s(0)$ es diferente o igual a $s(s(0))$. Esta pregunta exigiría investigar si $0 = s(0)$, entonces , por ejemplo, si tal cosa pasará todo se reduciría a 0, así que un supuesto básico es que $0 \neq s(0)$. Pero aún más, se nota que $s(X) = s(0)$ se puede ver como una ecuación con una incógnita o variable X que se puede obtener como 0, haciendo correctamente que $s(0) = s(0)$. Los procedimientos para realizar tal tipo de identificaciones entre términos son conocidos como *algoritmos de unificación*, de entre los que destacan el de Robinson [HV09] o de Martelli-Montanari [MM82].

2.3. Introducción al cálculo de eventos

Habiendo ya considerado una versión introductoria a la programación lógica, ahora se abordará el cálculo de eventos como formalismo lógico que posee fundamentos de la lógica.

El *cálculo de eventos* es un formalismo basado en la narrativa para el razonamiento acerca de las acciones y el cambio (sus efectos) respectivo. Fue propuesto originalmente desde la perspectiva de programación lógica por Bob Kowalski y Marek Sergot [KS86] con la premisa de que la *lógica formal* se puede utilizar para representar muchos tipos de conocimiento para muchos propósitos, por ejemplo puede ser utilizado para la especificación formal de programas (que a su vez puede ser una especificación), bases de datos, el uso en la legislación y el lenguaje natural en general, y que para muchas de estas aplicaciones de la lógica era necesaria una representación del tiempo, un tipo de *lógica temporal* no clásica. Está inspirado en el *cálculo de situaciones* [MH69].

Al igual que el cálculo de situaciones, el cálculo de eventos tiene acciones, llamados *eventos* y propiedades que varían en el tiempo, *fluentes*. Sin embargo, en el cálculo de situaciones la realización de una acción en una situación da lugar a una situación sucesora, además las acciones en el cálculo de situaciones son hipotéticas y el tiempo toma forma de árbol, mientras que en el cálculo de eventos hay una única línea de tiempo en el cual los eventos actuales ocurren [Mue08].

Como se mencionaba, el cálculo de eventos está basado en la *narrativa*, que es una especificación de un conjunto de ocurrencias de eventos reales, a diferencia del cálculo de situaciones en el cual es representada como una secuencia exacta de acciones hipotéticas.

Ciertos fenómenos se abordan de forma más natural en el cálculo de eventos, incluyendo eventos concurrentes, tiempo continuo, el cambio continuo, eventos con duración, los efectos no deterministas, eventos parcialmente ordenados y eventos desencadenados (*triggered events*). A continuación se presenta un ejemplo trivial tomado de [Mue08] para ilustrar lo que hace el cálculo de eventos: suponer que se desea razonar sobre el encendido y apagado de una lámpara.

Primeramente se tiene que representar el conocimiento general acerca de los efectos que pueden dar lugar los eventos:

1. Si el interruptor de la lámpara se gira hacia arriba, entonces la luz se encenderá.

2. Si el interruptor de la lámpara se gira hacia abajo, entonces la luz se apagará.

Ahora se representa un escenario específico:

1. La lámpara está apagada en el tiempo 0.

2. El interruptor de la lámpara se giró hacia arriba en el tiempo 5.

3. El interruptor de la lámpara se giró hacia abajo en el tiempo 8.

Al utilizar el cálculo de eventos se concluye lo siguiente:

1. En el tiempo 3, la lámpara está apagada.
2. En el tiempo 7, la lámpara está prendida.
3. En el tiempo 10, la lámpara está apagada.

2.4. El cálculo de eventos de Kowalski y Sergot

Desde la versión original del cálculo de eventos este ha evolucionado considerablemente y varios autores han propuesto sus versiones como Shanahan [MS02]. En este apartado se describe el *cálculo de eventos clásico* propuesto por Kowalski y Sergot en 1986 [KS86].

Trabaja con eventos (que ocurren), fluentes y períodos de tiempo, con lo que a partir de estos se definen los predicados (funciones) que utiliza este cáculo de eventos. Sean e,e_1, e_2=eventos que ocurren, f,f_1, f_2 =fluentes y p=período de tiempo, los predicados del cálculo de eventos son:

$Holds(p)$	El periodo de tiempo p se mantiene.
$Start(p, e)$	El evento e comienza en el periodo de tiempo p.
$End(p, e)$	El evento e finaliza en el periodo de tiempo p.
$Initiates(e, f)$	El evento e inicia el fluente f.
$Terminates(e, f)$	El evento e termina el fluente f.
$e_1 < e_2$	El evento e_1 precede al evento e_2.
$Broken(e_1, f, e_2)$	El fluente f es interrumpido entre los eventos e_1 y e_2.
$Incompatible(f_1, f_2)$	Los fluentes f_1 y f_2 son incompatibles.
$After(e, f)$	Periodo de tiempo después de ocurrir el evento e en el cual el fluente f se mantiene.
$Before(e, f)$	Periodo de tiempo antes de ocurrir el evento e en el cual el fluente f se mantiene.

Considerar el ejemplo de la sección anterior para aplicar algunos de los predicados del cálculo de eventos, el encendido y apagado de una lámpara. Se tiene un evento *prender* que refiere a que el interruptor se gira hacia arriba y provoca que la lámpara se encienda (fluente *prendido*)

y el evento *apagar* cuando el interruptor se gira hacia abajo y entonces la lámpara se apaga (fluente *apagado*). Suponer que el evento *prender* precede al evento *apagar*, $prender < apagar$ y que los fluentes planteados son incompatibles, $Incompatible(prender, apagar)$, ya que la lámpara sólo puede estar apagada o encendida. Siguiendo el escenario sucede lo siguiente:

1. $Before(prender, apagado)$, antes de que se gire el interruptor hacia arriba la lámpara se encuentra apagada.

2. $Initiates(prender, prendido)$, cuando el interruptor se gira hacia arriba la lámpara es prendida.

3. $After(prender, prendido)$, después de que el interruptor se gira hacia arriba la lámpara permanece prendida.

4. $Terminates(apagar, prendido)$, cuando el interruptor se gira hacia abajo la lámpara deja de estar prendida.

5. $Initiates(apagar, apagada)$, cuando el interruptor se gira hacia abajo la lámpara es apagada.

6. $After(apagar, apagada)$, después que el interruptor se gira hacia abajo la lámpara permanece apagada.

Notar que no se han considerado los puntos de tiempo específicos del ejemplo, los tiempos 0, 5 y 8, debido a que esta versión del cálculo de eventos no lo contempla como sí lo hace la siguiente iteración que Kowalski hizo a su original cálculo de eventos, el cual se presenta en la siguiente sección.

2.5. El cálculo de eventos simplificado

Considerado por algunos autores [Mue08, Ale95] como el *cálculo de eventos simplificado*, esta versión [Kow92] a diferencia de la primera agrega el concepto de puntos de tiempo específicos en que un evento ocurre y quita el de periodo de tiempo en el que un evento pudo haber ocurrido. Sea e=evento, f=fluente y t, t_1, t_2=puntos de tiempo, se tienen los siguientes predicados:

$Initially(f)$	El fluente f es verdadero en el punto de tiempo 0.
$HoldsAt(f,t)$	El fluente f es verdadero en el tiempo t.
$Happens(e,t)$	El evento e ocurre en el tiempo t.
$Initiates(e,f,t)$	Si el evento e ocurre en el tiempo t, entonces el fluente f es verdadero después del tiempo t.
$Terminates(e,f,t)$	Si el evento e ocurre en el tiempo t, entonces el fluente f es falso después del tiempo t.
$StoppedIn(t_1,f,t_2)$	El fluente f es detenido entre los tiempos t_1 y t_2.

Algunas diferencias con el original cálculo de eventos son:

a) Reemplaza el concepto de período de tiempo por puntos de tiempo (*timepoints*) específicos, los cuales son números enteros o reales no negativos.

b) Elimina la noción de fluentes incompatibles y a cambio se le otorga un valor de verdad al fluente.

c) Agrega el predicado $Initially(f)$ con el que se representa que el fluente f es inicialmente verdadero.

d) Se agrega un tercer argumento a los predicados $Initiates$ y $Terminates$ que es el tiempo en que ocurre el evento.

Continuando con el ejemplo del funcionamiento de la lámpara para utilizar ahora estos predicados, considerar que el fluente *prendido* es verdadero cuando ocurre el evento *prender* y falso cuando sucede el evento *apagar*, y que los tiempos $t_1 = 5$ y $t_2 = 8$:

1. $\neg Initially(prendido)$,en el tiempo 0 la lámpara está apagada, el fluente *prendido* es falso.

2. $Happens(prender, t_1)$, en el tiempo 5 ocurre el evento *prender*.

3. $Initiates(prender, prendido, t_1)$, después del evento *prender* en el tiempo 5 el fluente *prendido* es verdadero.

4. $Happens(apagar, t_2)$, en el tiempo 8 ocurre el evento *apagar*.

5. $Terminates(apagar, prendido, t_2)$, después del evento *prender* en el tiempo 8 el fluente *prendido* es falso.

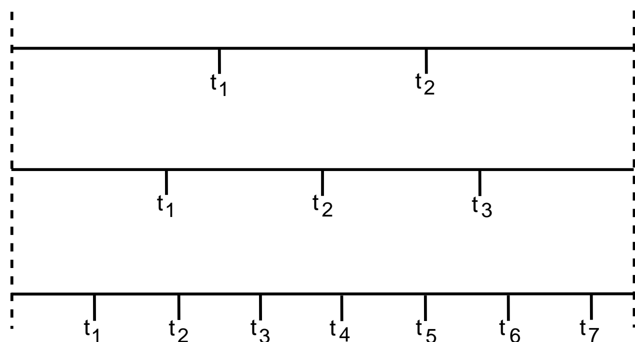

Figura 2.1: Representación básica de la granularidad del tiempo.

Por lo que se puede concluir que:

1. $\neg HoldsAt(prendido, 3)$, en el tiempo 3 la lámpara está apagada.

2. $HoldsAt(prendido, 7)$, en el tiempo 7 la lámpara está prendida.

3. $\neg HoldsAt(prendido, 10)$, en el tiempo 10 la lámpara está apagada.

Este tipo de inferencia puede programarse por ejemplo en lenguaje Prolog para automatizar las salidas en función de los argumentos. Por otro lado, el comportamiento de los eventos y fluentes se define en el siguiente apartado donde se considera la variable tiempo y cómo se aplican en los predicados arriba mencionados.

2.6. Ajuste hacia el cálculo de eventos

En esta tesis se supondrá que el tiempo tiene suficiente granularidad (instantes, ver Fig. 2.1) como para razonar sobre los eventos que ocurren, por muy corto que sea el período de ocurrencia entre estos eventos. Notar que esto no necesariamente conduce a un supuesto de continuidad del tiempo, ya que tal supuesto no se requiere, sólo basta señalar la partición adecuada de tiempo para realizar la correspondiente discretización. Se nombra a los puntos específicos de tiempo como *instantes*.

En base a la teoría de la lógica, para una adecuada asignación de valores de verdad de las proposiciones que componen una fórmula lógica Q se obtiene una *evaluación de veracidad* de esta fómula. Para el caso de un predicado que varía en el tiempo su función de evaluación debe adaptarse con el nuevo parámetro de tiempo, de tal forma que dependiendo de los instantes de tiempo la fórmula lógica Q puede ser falsa y en otros verdadera.

	t_0	t_1	t_2	t_3	
P_1	V	V	F	F	
P_2	F	V	V	F	
P_3	V	V	V	F	

Figura 2.2: Evaluación de predicados conforme transcurre el tiempo.

	e_1	e_2	e_3
	P_1 P_2	P_1	 P_3

Figura 2.3: Predicados afectados por la ocurrencia de eventos.

Se define un *mundo* como un conjunto de predicados $\mathcal{P}$ en los que en un instante dado se evalúan como cierto o a falso, y un *submundo* como un subconjunto de estos predicados.

Es necesario hacer constar que este concepto de mundo (y el de submundo) es *dinámico*, así que la veracidad de los predicados cambia conforme transcurre el tiempo, sin embargo, el mundo a tratar tiene predeterminado un conjunto finito de predicados que lo definen. En el diagrama de la Fig. 2.2 se aprecia la continuidad de este cambio en los predicados P_1, P_2 y P_3.

Ahora considerar un evento e que ocurre en un instante t, si tal evento no altera en absoluto a ningún predicado del mundo tratado entonces no tendría caso que existiera. Esto hace que los eventos sólo tengan significado a través de los predicados que afectan. En cambio, si un predicado fuera verdadero durante un periodo *demasiado* corto podría no ser verificable que realmente fue verdadero durante ese periodo. Así que dependerá en algún grado de coeficiente de *observabilidad* el que un evento afecte el mundo o no. Finalmente, algunos predicados pueden considerarse *globales*, con un monitoreo constante de un agente omnisciente mientras que otros serán locales (un subconjunto de los globales) con un monitoreo propio a la visión de un agente (su *grado* de conocimiento).

Sea un conjunto de eventos $\mathcal{E}$, y que para cada e_i el conjunto de predicados $P(e_i)$ que son afectados por la ocurrencia de e_i. Veáse el diagrama de la Fig. 2.3 para ver la posible dependencia entre los eventos e_1, e_2 y e_3 con respecto a los predicados P_1, P_2 y P_3.

Este tipo de dependencia debe hacerse explícita para el tipo de dominio de razonamiento

que posteriormente se utilizará. Antes, se abordarán los detalles de la axiomatización del cálculo de eventos.

2.6.1. Inicio y terminación de fluentes mediante eventos

La principal parte para axiomatizar es el establecimiento entre la ocurrencia de un evento y el predicado que a partir del evento comenzaría a hacerse verdadero, para ello se veía que se tiene el siguiente predicado como una *reificación* (materialización):

$$\mathit{Initiates}(e, f, t_0) \leftarrow$$

cuyo *significado deseado* es que el fluente f es verdadero a partir del momento t_0 en que el evento e ocurre, y en donde $\mathit{Happens}(e, t_0)$ es verdadero cuando el evento e ocurre en el instante t_0. El predicado $\mathit{Initiates}(e, f, t)$ hace que se consideren tres posibles subdominios: el primer subdominio es el de todos los eventos; el segundo subdominio es el de todos los predicados que temporalmente dependen de la ocurrencia de los eventos en $\mathcal{E}$; finalmente, el tercer subdominio está dedicado al conjunto de los instantes de tiempo.

Así que $\mathit{Initiates}/3$ sería un ejemplo de un predicado afirmando relaciones entre predicados y sus argumentos de eventos y tiempo. En lógica de primer orden esto no está permitido, pues conduciría a la lógica de segundo orden. Es por ello que los predicados afectados por los eventos son reificados con otros predicados especiales, sin considerar que estos predicados variarán en algún momento.

Para distinguir a los predicados pasados como parámetros, de ahora en adelante se llamarán *fluentes*, y formarán un conjunto específico bien determinado $\mathcal{F}$. Se exigirá que este conjunto sea no vacío y tal que $\mathcal{F} \cap \mathcal{E} = \emptyset$, de tal manera que se impide que un fluente sea un evento y viceversa. La siguiente cláusula define el dual de *Initiates*:

$$\mathit{Terminates}(e, f, t_0) \leftarrow$$

y afirma que la ocurrencia del evento e *finaliza* la veracidad del fluente f, y que esto se mantiene así a partir del tiempo t_0 en adelante. En el diagrama mostrado en la Fig. 2.4 se tiene que el evento e que ocurre en el tiempo t_1 determina durante un periodo la veracidad del fluente f. En este mismo diagrama ocurre un evento llamado e' en el instante t_2 $(t_1 < t_2)$, y este evento finaliza o termina el fluente f en cuanto su periodo $\Delta t = t_2 - t_1$. Se debe notar que la *existencia* del fluente como tal permanece independientemente de que ocurran otros eventos y por definición el periodo Δt es necesariamente diferente de 0.

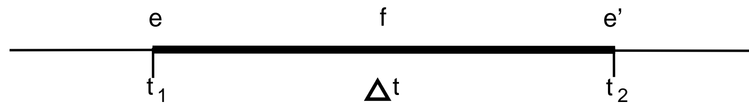

Figura 2.4: Inicio y terminación de la veracidad de un fluente.

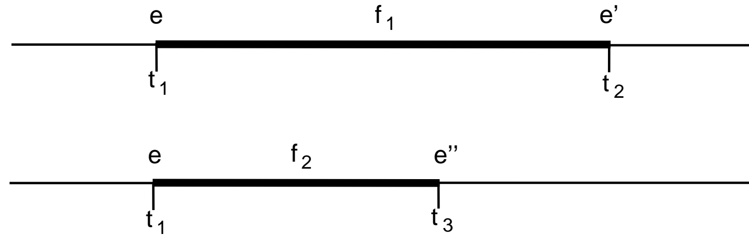

Figura 2.5: Un evento originando más de un fluente.

Algunas notas aclaratorias son pertinentes en este punto. En las definiciones de los predicados *Terminates*/3 y de *Initiates*/3 pueden pasar varios fenómenos. El primero de ellos es que un mismo evento puede generar diversos fluentes (ver Fig. 2.5). Así que tenemos un fenómeno de *simultaneidad*. A partir de ahí, los fluentes bien podrían terminar en tiempos distintos (como t_2 y t_3 en el diagrama citado). Similarmente, más de un evento puede afectar la veracidad de un fluente. En este caso, se puede hacer un tipo de *razonamiento abductivo* [DK02] para investigar qué fluentes pudieron hacer a un fluente verdadero o falso. La siguiente observación es con respecto a la *consistencia* de tiempos y de ocurrencias y su efecto en fluentes. El siguiente ejemplo ilustra el problema:

$$\mathit{Initiates}(e_1, f, t_0) \leftarrow \tag{2.7}$$

$$\mathit{Terminates}(e_1, f, t_0) \leftarrow \tag{2.8}$$

Para el caso en que se afirme que $\mathit{Happens}(e_1, 3)$, es decir $t_0 = 3$, la situación conduce a una inconsistencia que hace que un evento f comienza y termina para un mismo evento a partir de un instante dado t_0, que si bien puede no afectar al resto del sistema, sí afecta los principios de economía del razonamiento y en definitiva es un error de análisis o de diseño.

Finalmente, para esta parte de la discusión, la posibilidad de que un evento termine varios fluentes se ilustra en el diagrama dado en la Fig. 2.6.

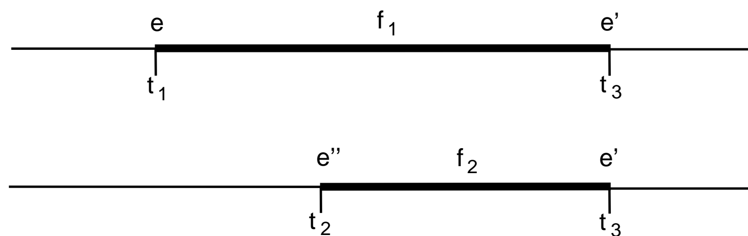

Figura 2.6: Un evento terminando más de un fluente.

2.6.2. Interpretaciones del cálculo de eventos

El cálculo de eventos tiene algunas interpretaciones [DK02, Sha00], las cuales están representadas en el diagrama de la Fig. 2.7. Esta interpretación resulta muy general, pero posee tres interpretaciones especializadas: una *inductiva*, una *abductiva* y otra *deductiva*.

2.6.2.1. Punto de vista inductivo

La interpretación inductiva requiere que se conozcan suficientes eventos y fluentes para obtener hipótesis inductivas generales. Se trata de hallar con toda precisión qué eventos se pueden generalizar para modificar siempre uno o varios fluentes. Este punto de vista requiere una inspección rigurosa y experimental de un caso de estudio en donde el cálculo de eventos se quiere aplicar, para definir los eventos y los fluentes afectados.

Por ejemplo, un temblor podría ser de gravedad extrema con una combinación de fluentes diversos: *sobreexplotaciónAcuífera*, *erupciónVolcánica*, *construccionesCivilesDébiles*. Ahora, se requiere hallar qué eventos originaron este tipo de fluentes. Habría eventos de índole geológica, de índole poblacional, y otros de índole económico-social. El fluente principal sería *ciudadDestruida*.

2.6.2.2. Punto de vista abductivo

La interpretación abductiva es cuando se conocen bien los efectos, vía fluentes, y se quieren investigar las causas, se *conjetura* qué eventos podrían ser el origen de la alteración de verdad de un fluente. Para el ejemplo, se conjeturaría que tal vez no es cierto que la sobre explotación acuífera genera temblores, sino el movimiento tectónico. El caso es que se desea que las *hipótesis abductivas* sean consistentes con los fenómenos observados.

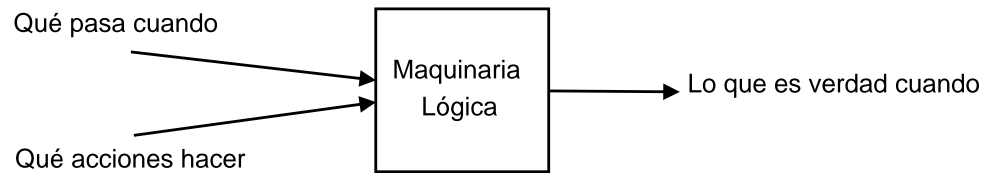

Figura 2.7: Cómo funciona el cálculo de eventos.

2.6.2.3. Punto de vista deductivo

Consiste en concluir qué fluentes son verdaderos en puntos de tiempo particulares de un intervalo de tiempo dado cuando un evento ha ocurrido [MS02]. Continuando con el ejemplo de un temblor, y desde el punto de vista deductivo, se formula la situación como sigue: "Si se sobreexplotan los mantos acuíferos y si hay erupciones volcánicas cercanas entonces habrá temblores devastadores".

En esta tesis la utilización principal del cálculo de eventos es con respecto a la parte deductiva y parte de las aportaciones consiste en establecer la siguiente hipótesis: el cálculo de eventos es una herramienta apropiada para el análisis de sistemas de cómputo. Para que esta hipótesis sea confirmada se aplicará el cálculo de eventos al caso de los sistemas distribuidos con paso de mensajes.

Capítulo 3

Sistemas distribuidos

Este capítulo describe de forma breve los sistemas distribuidos. Se menciona la importancia, las características y las aplicaciones que tienen, así como las arquitecturas básicas más comunes y los modelos de comunicación que emplean sus entidades. Puesto que está tesis pretende la formulación de una notación para la especificación de estos sistemas, es conveniente conocer la forma en que trabajan y qué especificaciones existen hoy en día para ello.

3.1. Importancia de los sistemas distribuidos

En los últimos años los sistemas distribuidos han tomado gran importancia debido a su presencia en la vida diaria, esto debido a las siguientes razones [Gho07, KS08]:

- La necesidad de tener un entorno de cómputo distribuido geográficamente. Por ejemplo, en el caso de los bancos que deben mantener el control de las cuentas de sus clientes en cualquier parte que hagan uso de ellas. Se comunican con otros bancos para monitorear transacciones interbancarias o para registrar fondos transferidos desde cajeros automáticos dispersos geográficamente. Otro caso común de esto es el Internet mismo.
- La necesidad de acelerar el cálculo de cómputo. Una técnica para obtener más potencia computacional es el uso de múltiples procesadores, dividiendo un problema en subproblemas más pequeños y asignándolos a procesadores separados físicamente.
- La necesidad de compartir recursos: hardware y software. Las bases de datos distribuidas son un ejemplo donde se comparten recursos de software, ya que una gran base de datos debe ser almacenada en varias computadoras y frecuentemente es actualizada

o consultada por una gran número de procesos agentes. El acceso remoto a datos y recursos es el denominador común.

- La necesidad de tener tolerancia a fallas del sistema (capacidad de recuperarse ante fallos del sistema). La desventaja de sistemas de cómputo construidos en torno a un único nodo central son propensos a un colapso total cuando falle, lo cual es muy arriesgado cuando se desea tener siempre un correcto funcionamiento.

3.2. Definición de un sistema distribuido

Para poder entender esta definición compuesta de dos términos, se debe recordar qué es un *sistema*. Un sistema es un conjunto de elemetos que se relacionan entre sí para alcanzar un fin común [Sky05]. Respecto al término *distribuido* esto no sólo se refiere a la *distribución física*, no porque un procesador de un sistema de cómputo se encuentre a 100 metros de la memoria principal se trata de un sistema distribuido, sino también se debe considerar la *distribución lógica* o funcional de las capacidades de procesamiento. Dicha distribución lógica está basada en el siguiente conjunto de criterios [Gho07]:

- Procesos múltiples. El sistema consiste de más de un proceso secuencial. Estos procesos pueden ser del sistema o procesos del suario.

- La comunicación entre procesos. Los procesos se comunican entre sí mediante mensajes, que toman un tiempo finito para viajar de un proceso a otro. La demora para el tránsito de los mensajes dependerá de las características físicas de los enlaces de mensajes. A estos enlaces de mensajes también se les conoce como canales.

- Espacios disjuntos de direcciones. Los procesos tienen espacios de direcciones disjuntos, así que si se tienen multiplés procesadores que emplean memoria compartida esto no es una representación real de un sistema de cómputo distribuido.

- Meta colectiva. Los procesos deben interactuar entre sí para alcanzar un objetivo común. Considerar dos procesos: P y Q dentro de una red de procesos. Si P calcula $f(x) = x^2$ para un determinado conjunto de valores de x y por un lado Q multiplica un conjunto de números por π, entonces a esto no se le considera un sistema distribuido, ya que no hay interacción entre P y Q. Sin embargo, si P y Q cooperan uno con el otro para calcular las áreas de un conjunto de círculos con radio x, entonces el sistema de procesos (en este caso P y Q) es un ejemplo de sistema distribuido significativo.

Esto siginifica que la distribución física pudiera ser sólo un requisito previo para la distribución lógica. El término de *sistemas distribuidos* ha sido definido por varios autores, como Lesli Lamport desde 1978 [Lam78], estas son algunas definiciones que intencionalmente se enlistan en forma cronológica:

1. Un sistema distribuido consiste de una colección de distintos procesos los cuales están espacialmente separados y se comunican unos con otros a través del intercambio de mensajes [Lam78].
2. Un sistema distribuido es un conjunto de computadoras que no comparten memoria o un reloj físico en común, se comunican por mensajes que pasan a través de una red de comunicación donde cada equipo tiene su propia memoria y ejecuta su propio sistema operativo. Normalmente, las computadoras son semiautónomas y están débilmente acopladas mientras cooperan para resolver un problema de forma colectiva [SS94].
3. Un sistema distribuido es una colección de computadoras independientes que aparecen a sus usuarios como un único sistema [TvS07].
4. Un sistema distribuido es una colección de entidades independientes que cooperan para resolver un problema que no se puede resolver individualmente [KS08].
5. Un sistema distribuido es el que sus componentes están localizados en una red de computadoras, las cuales se comunican y coordinan sus acciones sólo por el paso de mensajes [CDKB11].

La definición en esencia no ha cambiado a lo largo de los años, se habla de un conjunto de componentes que pudieran ser procesos o equipos de cómputo independientes en su funcionamiento interno y que se comunican a través del envío de mensajes para tratar parte de la solución de un problema. Enseguida se mencionan las características de los sistemas distribuidos y sus aplicaciones en las diversos sectores industriales y sociales.

3.2.1. Características de un sistema distribuido

Los sistemas distribuidos se caracterizan principalmente por [CDKB11, KS08]:

- No poseer un reloj físico común. Se asume que cada proceso en un sistema distribuido tiene un reloj local. Cuando estos necesitan cooperar sólo coordinan sus acciones mediante el intercambio de mensajes.

- No compartir la misma memoria. Esta es una característica clave que se requiere para la comunicación de paso de mensajes e implica la ausencia del reloj físico común.

- Separación geográfica. Procesadores separados geográficamente y que se comunican para realizar alguna tarea en conjunto es una representación de un sistema distribuido. Puede estar funcionando bajo *arquitecturas de redes* como LAN, MAN o WAN.

- Concurrencia. La ejecución de programas concurrentes es un distintivo. Por ejemplo la división de una tarea en subtareas más pequeñas que puedan realizarse concurrentemente (al mismo tiempo) y la respectiva asignación de estas a procesadores aumentaría la velocidad de respuesta.

- Autonomía y heterogeneidad. Los procesadores están débilmente acoplados, ya que tienen diferentes velocidades y cada uno puede estar ejecutando un sistema operativo distinto. Por lo general, no son parte de un sistema dedicado, pero cooperan entre sí para ofrecer servicios o resolver un problema de manera conjunta.

3.2.2. Aplicaciones de los sistemas distribuidos

Enseguida se presentan algunos sectores industriales y sociales en donde se han aplicado [CDKB11] los sistemas distribuidos:

- Finanzas y comercio. El comercio electrónico ejemplificado en compañias como eBay y Amazon; plataformas tecnológicas de pago como PayPal y la banca en línea, así como sistemas complejos de mercados financieros tienen parte de sistemas distribuidos.

- La sociedad de la información. El crecimiento de la *World Wide Web* como un repositorio de información y conocimiento. El desarrollo de motores de búsqueda como Google y Yahoo que actúan sobre un basto repositorio. Las bibliotecas digitales e información antigua digitalizada como Google Books. El contenido generado por todo tipo de usuarios en sitios como YouTube, Wikipedia y Flickr, ni que decir de las redes sociales.

- Grupos de ciencia. La aparición del cómputo *Grid*[1] como una tecnología fundamental para apoyar el almacenamiento, análisis y procesamiento de grandes cantidades de datos científicos, permitiendo la colaboración a nivel mundial entre los grupos científicos.

[1] *Grid Computing* se refiere a una colección de recursos de equipos de cómputo localizados desde múltiples lugares que trabajan de forma coordinada para alcanzar una meta [JBFT05].

- Industria del entretenimiento. La aparición de juegos en línea como una forma novedosa y altamente interactiva, la disponibilidad de música, videos y cine en el hogar a través de centros multimedia en Internet, la descarga y *streaming* de este tipo de contenido que ha revolucionado la red han estado presentes los sistemas distribuidos.

- Educación. Aquí se ha dado con la aparición del *e-learning* (aprendizaje electrónico) a través de herramientas basadas en web como los entornos virtuales de aprendizaje; el apoyo correspondiente para el aprendizaje a distancia y colaborativo.

- Transportes y logística. El uso de tecnologías de localización como el GPS en sistemas de búsqueda de rutas y de gestión del tráfico ya sea para medios de transporte terrestres o aéreos. El desarrollo de servicios de mapas basados en web como MapQuest, Google Maps y Google Earth también han sido favorecidos.

- Administración ambiental. El uso de la tecnología de sensores tanto para monitorizar como gestionar el entorno natural, y así por ejemplo proporcionar alertas tempranas de desastres naturales como terremotos, inundaciones o tsunamis y coordinar la respuesta de emergencia; la recopilación y análisis de los parámetros ambientales globales para entender mejor los fenómenos naturales complejos como el cambio climático.

3.3. Arquitecturas de sistemas distribuidos

En esta sección se presentan las arquitecturas principales empleadas en los sistemas distribuidos. En particular los modelos Cliente/Servidor y Peer-to-peer que son las más utilizadas [CDKB11] , sin embargo también existen otras como:

- *arquitectura de objetos distribuidos*, las entidades son objetos con los que se puede trabajar independientemente de donde se localicen.

- *arquitectura multiprocesador*, se distingue por el hardware debido a que el sistema se ejecuta sobre varios procesadores que pueden estar en una misma computadora.

- *arquitectura en varias capas*, basada en la arquitectura Cliente/Servidor y en los que estos se estructuran de cierta forma para separar las funcionalidades del sistema tomando roles más definidos, las capas pueden ir de dos a más.

- *arquitectura orientada a servicios*, está orientada a diseñar el sistema distribuido a gran escala basándose en los servicios que debe proporcionar considerando también a terceros como posibles proveedores del sistema.

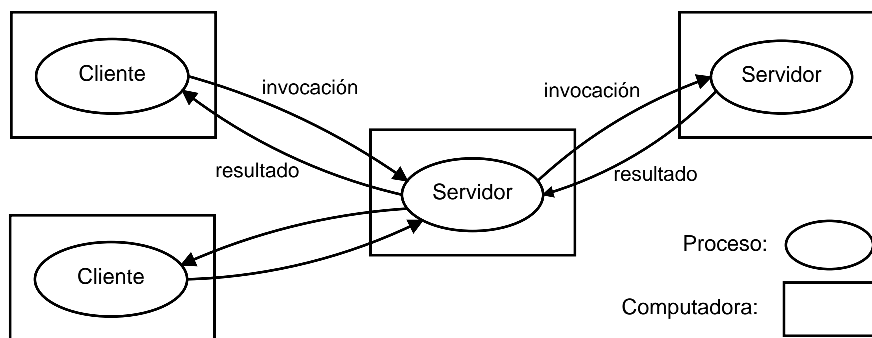

Figura 3.1: Los procesos con roles Cliente o Servidor

- *arquitectura modelo vista controlador*, es un caso específico de la arquitectura en varias capas, posee tres capas y está enfocado al desarrollo web.

En este trabajo se abordarán las dos arquitecturas mencionadas debido a la filosofía de programación que maneja Erlang, ya que no es basado en objetos y sus programas pueden ejecutarse en varias computadoras separadas. Tampoco puede ofrecer un diseño basado en servicios.

La *arquitectura de un sistema distribuido* es su estructura en términos de sus componentes especificados por separado y las interrelaciones entre estos. El objetivo general es asegurar que la estructura sea fiable, manejable, adaptable y rentable, además de proporcionar un marco de referencia consistente.

Sin embargo, para entender las construciones fundamentales de los sistemas distribuidos es necesario considerar cuáles son las *entidades* que se comunican en un sistema distribuido y cómo lo hacen. A estas entidades se les llama típicamente *proceso*, pueden ser representados por hilos (*threads*) de los sistemas operativos o de las plataformas de programación, o por *nodos* que se encuentran en la red. Las entidades desde la perspectiva de programación también pueden ser representadas por *objetos* en el enfoque de sistemas distribuidos basados en objetos.

3.3.1. Cliente/Servidor

La arquitectura Cliente/Servidor frecuentemente es la más citada cuando se habla de sistemas distribuidos. Los procesos toman el rol de *cliente* o *servidor*. En particular, los procesos clientes interactúan con procesos servidores con el fin de accesar a recursos que estos administran. Pueden estar hospedados en distintos equipos de cómputo. En la Fig. 3.1

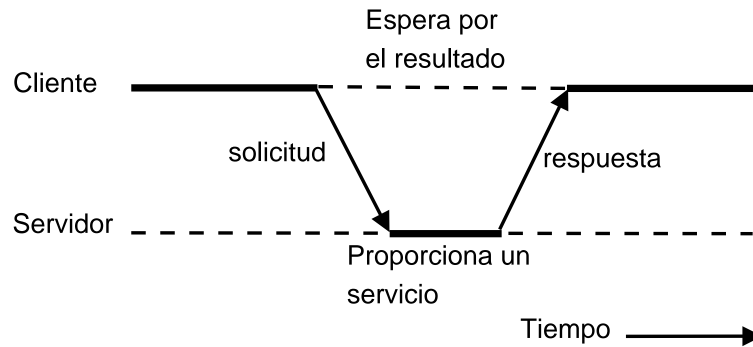

Figura 3.2: Interacción entre Cliente y Servidor

se ilustra una estructura simple donde los procesos toman estos roles.

Los servidores pueden ser a su vez clientes de otros servidores, como lo muestra el gráfico. Por ejemplo, un servidor web es a menudo un cliente de un servidor de archivos local que gestiona los archivos en los que las páginas web se almacenan. Los servidores web así como la mayoría de los otros servicios de Internet son clientes del servicio DNS, el cual traduce los nombres de dominio de Internet en direcciones de red. Otro ejemplo relacionado con la web se refiere a los motores de búsqueda, los cuales permiten a los usuarios ver los resúmenes de la información disponible en las páginas web en los sitios de todo Internet.

La interaccción entre cliente y servidor se da cuando el cliente hace una *solicitud* al servidor y entra en un tiempo de espera mientras este último trabaja para proporcionar la *respuesta*. La Fig. 3.2 muestra dicha interacción.

En la sección 4.6 del capítulo 4 se abordará más a detalle pero ahora se verá implementada en el lenguaje de programación Erlang, mostrando la primitivas correspondientes para la interacción entre estos roles de procesos; ya que servirá como ejemplo para explicar la notación propuesta en esta tesis.

3.3.2. Peer-to-peer

En la arquitectura Peer-to-peer todos los procesos involucrados en una tarea o actividad desempeñan roles similares, interactuando cooperativamente como iguales (*peers*) sin distinción alguna entre procesos cliente y servidor o equipos de cómputo en los que se ejecutan. En términos prácticos, todos los procesos que participan ejecutan el mismo programa y ofrecen el mismo conjunto de interfaces entre sí. Mientras que el modelo Cliente/Servidor ofrece un enfoque directo y relativamente sencillo para el intercambio de datos y otros recursos.

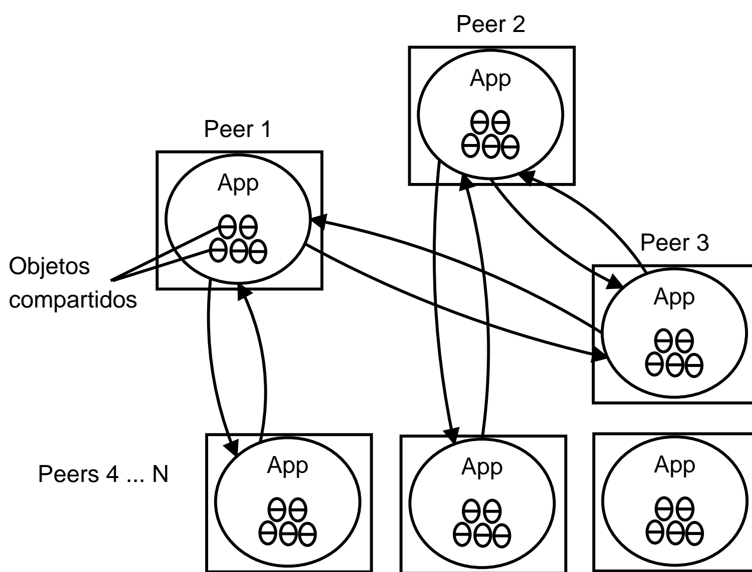

Figura 3.3: Arquitectura Peer-to-peer

La cuestión fundamental que trata de atacar esta arquitectura es la necesidad de distribuir recursos compartidos más ampliamente con el fin de compartir las cargas de cómputo y comunicaciones que incurren al acceder a estos recursos por un número mucho mayor de computadoras y conexiones de red. Así, la idea clave que condujo al desarrollo de los sistemas Peer-to-peer es que los recursos de red y de cómputo propiedad de los usuarios de un servicio también podrían ser objeto de uso para apoyar a ese servicio. Esto tiene la consecuencia útil que los recursos disponibles para ejecutar el servicio crecen con el número de usuarios. Más aún que las capacidades de hardware y de los sistema operativos de las computadoras actuales superan a la de los servidores de ayer, y la mayoría pueden estar equipadas con conexiones permanentes de banda ancha a la red.

El objetivo de la arquitectura Peer-to-peer es explotar los recursos tanto de datos como de hardware de un gran número de equipos participantes para la realización de una tarea o actividad determinada. Las aplicaciones y sistemas Peer-to-peer permiten a decenas o cientos de miles de computadoras proporcionarles acceso a datos y a otros recursos que se almacenan y gestionan colectivamente. Uno de los primeros ejemplos de este tipo de sistemas fue la aplicación Napster para compartir archivos de música digital en el año 2000. Otro ejemplo más reciente y ampliamente utilizado es el sistema de intercambio de archivos BitTorrent.

La Fig. 3.3 ilustra la arquitectura Peer-to-peer, donde las aplicaciones están compuestas de un gran número de procesos iguales que se ejecutan en computadoras separadas. Muchos objetos de datos son compartidos, una computadora tiene sólo una pequeña parte de la base

de datos de la aplicación y las cargas de almacenamiento, procesamiento y comunicación para el acceso a los objetos se distribuyen a través de muchas computadoras y conexiones de red. Cada objeto se replica en varias computadoras para distribuir aún más la carga y proporcionar capacidad de recuperación en caso de desconexión de equipos individuales. La necesidad de colocar objetos individuales para recuperarlos y mantener las réplicas entre muchas computadoras hace que esta arquitectura sea mucho más compleja que la de Cliente/Servidor.

3.4. Modelos de comunicación de procesos

La *comunicación entre procesos* es el corazón del cómputo distribuido [Gho07]. Los procesos de usuario se ejecutan en computadoras conectadas una a otra a través de una red que transporta las señales que se propagan desde un proceso a otro. Estas señales representan datos[2].

Existen un par de modelos básicos para la comunicación entre procesos que se denominan [KS08]: *síncronos* y *asíncronos*. El modelo de comunicación síncrono es un tipo de bloqueo, debido a que un mensaje se envía y el proceso emisor se detiene hasta que el mensaje ha sido recibido por el proceso receptor. El proceso emisor continúa la ejecución sólo después de que se entera de que el proceso receptor ha aceptado el mensaje. Por lo tanto los procesos emisor y receptor se deben sincronizar para el intercambio de un mensaje.

Por otro lado el modelo de comunicación asíncrono es sin bloqueo, ya que el emisor y el receptor no se sincronizan para el intercambio de mensajes. Después de haber enviado un mensaje, el emisor no espera a que el mensaje sea entregado al proceso receptor. El mensaje es almacenado por el búfer del sistema y entregado al proceso receptor cuando este listo para aceptar el mensaje.

Ninguno de los modelos de comunicación es superior al otro. La comunicación asíncrona proporciona mayor paralelismo, porque el proceso emisor se puede seguir ejecutando mientras el mensaje está en tránsito hacia el receptor. Sin embargo una implementación de la comunicación asíncrona requiere una gestión de memoria más compleja. Además, debido al mayor grado de paralelismo y no determinismo, es mucho más difícil de diseñar, verificar y aplicar algoritmos distribuidos para comunicaciones asíncronas. La comunicación síncrona es más sencilla de manejar y poner en práctica. Sin embargo, debido al bloqueo frecuente, es probable que tenga un rendimiento inferior y que sea más susceptible a los interbloqueos.

[2] La palabra dato aquí se utiliza para cualquier cosa que pueda ser representado por una cadena de bits.

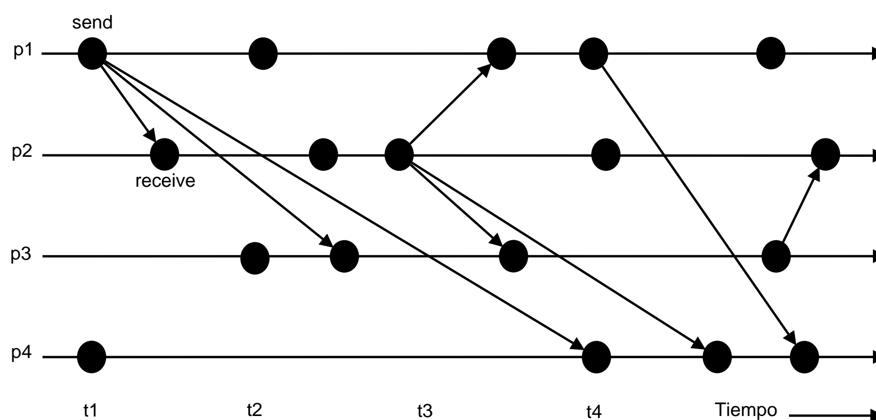

Figura 3.4: Ejemplo de la ejecución de eventos en los procesos

Posteriormente en la sección 4.4 del capítulo 4 se presentarán los paradigmas de comunicación de procesos centrando la atención en el que emplea el lenguaje Erlang: el paso de mensajes.

En muchos casos es necesario conocer si el evento (envío o recepción de un mensaje) en un proceso ocurre antes, después o al mismo tiempo (concurrentemente) a otro evento de otro proceso. Así la ejecución de un sistema puede ser descrita en términos de eventos y el orden en que ocurren estos, a pesar de la falta de un reloj físico común [CDKB11]. El *diagrama espacio-tiempo* de la Fig. 3.4 muestra un ejemplo de la ejecución de cuatro procesos en un sistema distribuido. Las líneas horizontales representan el progreso de los procesos; un punto remarcado en una línea indica un evento; las flechas inclinadas entre las líneas horizontales representan la transferencia de un mensaje y apuntan al destino del mensaje. Un proceso puede tener eventos internos, en la figura son puntos remarcados que no están conectados con ninguna flecha. Generalmente la ejecución de un evento toma una cantidad finita de tiempo. Para el caso del proceso p_1, el primer evento es el envío de un mensaje hacia p_2, el segundo es un evento interno, el tercer evento es la recepción de un mensaje proveniente de p_2, el cuarto es el envío de un mensaje pero ahora al proceso p_4 y el último es otro interno.

3.5. Especificación de sistemas distribuidos

Los sistemas distribuidos presentan a lo largo de su desarrollo retos de alta complejidad en las etapas de especificación, diseño, implementación y verificación, lo que ha dado lugar en el caso de la especificación a que se elaboren propuestas de herramientas y formalismos que ayuden a realizar más eficientemente el resto del ciclo de desarrollo. Como ya se mencionó en

la sección 1.1 del capítulo 1, una de las ventajas de la especificación es que a partir de ella es posible desarrollar técnicas y herramientas que hagan que la etapa de implemetación se realice de forma automática al menos parcialmente.

Así, coadyuvando al desarrollo de sistemas distribuidos con paso de mensajes, el presente trabajo de tesis se centra en proponer una especificación para estos sistemas a través del cálculo de eventos, además de presentar una correspondencia entre está especificación y la sintaxis del lenguaje de programación Erlang, el cual aborda la complejidad del desarrollo a gran escala de sistemas concurrentes y distribuidos. Esta correspondencia que se menciona ayudará a convertir una notación del cálculo de eventos a código Erlang con las limitantes que más tarde se indicarán. Esta tesis no cubre la automatización de la conversión sino que da las bases para poder conseguirla y que en trabajos futuros pueda ser desarrollada, por ejemplo, como complemento (*plugin*) en algún IDE como Eclipse o Netbeans, o tal vez una página web en la que se puedan obtener los programas Erlang.

Algunas especificaciones para sistemas distribuidos así como otras especialmente para el desarrollo con Erlang se basan en el formalismo *Álgebra de Procesos*, que es usado para definir un enfoque axiomático de procesos y para modelar sistemas concurrentes. Algunos de los principales álgebras de procesos son el *Algebra of Communicating Processes*(ACP)[BK82], *Calculus of Communicating Systems* (CCS)[Mil82], *Communicating Sequential Process*(CSP)[Hoa85] y el *Cálculo-Pi* [Mil99].

En el año 2005 Thomas Noll [NR05] presenta una modelación formal para programas escritos en el lenguaje Erlang. Dicha modelación está basada en el Cálculo Pi, la cual posee métodos de análisis y verificación de procesos que ya han sido desarrollados. Noll ofrece un mapeo de la programación Erlang al Cálculo Pi propuesto, así más tarde propone la verificación automática de programas Erlang [RNRC06] traducidos a especificaciones en el Cálculo Pi. Respecto a la propuesta planteada en esta tesis, sería una antesala a la de Noll, ya que se comenzaría a definir el sistema con el cálculo de eventos para enseguida tener su representación en Erlang, y apartir de entonces es posible emplear el Cálculo Pi.

Otras propuesta basadas en el Cálculo Pi se expone en [Hen07, SFS09], en los cuales se desarrollan versiones del Cálculo Pi para describir el comportamiento de agentes distribuidos y sistemas distribuidos respectivamente, sin aterrizar la implementación en ninguna herramienta de programación.

Capítulo 4

Programación concurrente en Erlang

El lenguaje de programación Erlang fue concebido para hacer frente a problemas relacionados con el desarrollo de sistemas distribuidos de tiempo real altamente concurrentes, buscando una solución para [CT09]: a) poder desarrollar este tipo de software de forma rápida y eficiente, b) disponer de sistemas tolerantes a fallos de software y hardware y c) poder actualizar el software sobre la marcha, sin detener la ejecución del sistema. En este apartado se describe cómo el lenguaje de programación Erlang afronta estos problemas utilizando el concepto de *proceso* y se define la sintaxis de la *programación concurrente* en Erlang. Cabe mencionar que en el ápendice A se presenta la programación básica empleada en Erlang para comprender su filosofía, aquí solamente se abordará la programación concurrente de este lenguaje de programación, la cual es una parte medular del trabajo de tesis ya que de ello desprenderán las soluciones para la consecución de los objetivos planteados.

4.1. Introducción

A mediados de la década de los 80's el Laboratorio de Ciencias de Computación de la compañía Ericsson se dio a la tarea de investigar los lenguajes de programación adecuados para el desarrollo de la próxima generación de productos de telecomunicaciones por lo que Joe Armstrong, Robert Virding y Mike Williams después de dos años de investigación concluyeron que aunque los lenguajes de programación existentes tenían características interesantes y relevantes, no había alguno que reuniera todas las que necesitaban, por lo que decideron crear uno propio.

Erlang está influenciado por lenguajes funcionales como ML (*Meta Lenguaje*) y Miranda,

lenguajes concurrentes como ADA, Modula y Chill, pero principalmente por el lenguaje de programación lógica Prolog [CT09]. Es un lenguaje orientado a la concurrencia que facilita la programación distribuida y cuenta con mecanismos para la tolerancia a fallas. Diseñado para ejecutar programas de forma ininterrumpida, lo que significa que es posible modificar el código de las aplicaciones sin detener su ejecución. Muchos proyectos de software y organizaciones han utilizado Erlang en sus sistemas de producción algunos son [Kes12, CT09]:

- *Amazon* para implementar SimpleDB, la prestación de servicios de bases de datos como parte de la Amazon Elastic Compute Cloud (EC2)[1].
- *Facebook* para alimentar el soporte de comunicaciones de la mensajería en su servicio de chat [Let09].
- *Ericsson* lo utiliza en sus nodos de soporte de redes móviles 3G GPRS.
- *Yahoo!* utilizó Erlang en su servicio de marcadores sociales *Delicious*, ahora propiedad de AVOS systems.
- *T-Mobile* lo emplea en sus sistemas de autenticación, de SMS y control de llamadas.
- *Yaws* es un servidor web basado en Erlang de código abierto conocido por su fiabilidad, estabilidad y escalabilidad [Vin11].
- *GitHub*, un servicio de hosting basado en la web para proyectos de desarrollo de software que utilizan el sistema de control de versiones Git. Erlang se utiliza para proxies RPC(Remote Procedure Call) a procesos Ruby [Inf10].
- *WhatsApp*, la aplicación móvil de mesajería lo emplea para el envío de cientos de miles de millones de mensajes por día entre sus usuarios [Bou14, Sol12].

4.2. Características de Erlang

Las características que posee el lenguaje de programación Erlang y por las cuáles se creó son [LMC11, CT09]:

[1] Amazon EC2 es un servicio web que proporciona capacidad computacional en la nube, diseñado para facilitar a los desarrolladores recursos computacionales escalables basados en web, www.aws.amazon.com/es/ec2.

- **Construcciones de alto nivel.** Es un un lenguaje declarativo, ya que sigue el principio de tratar de describir lo que debe ser computado, en lugar de decir cómo calcularlo. Así, la definición de una función será leída como un conjunto de ecuaciones, por ejemplo para calcular el área de una figura:

```
1 area({cuadrado, Lado})-> Lado * Lado;
2 area({circulo, Radio})-> math:pi() * Radio * Radio.
```

  Esta definición de función toma como argumento una tupla en la que se especifica una figura (un cuadrado o un círculo) y dependiendo del tipo de figura que se trate en el primer elemento de la tupla[2] devuelve el área correspondiente.

- **Crear sistemas distribuidos.** Para balancear la carga entre los sistemas de hardware, permite la creación y comunicación de procesos no sólo en una máquina sino también en otras.

- **Tolerante a fallos.** Cuando una parte del sistema presenta fallos y se detiene, esto no significa que todo el sistema también tenga que detenerse. En sistemas de software como PLEX o C, un fallo en el código determina una interrupción completa del programa con todos sus hilos y procesos. Hay otros lenguajes como Java, Python o Ruby que manejan estos errores como excepciones, afectando sólo a una parte del programa y no a todos sus hilos. No obstante, en los entornos con memoria compartida, un error puede dejar corrupta esta parte de memoria por lo que esa opción no garantiza tampoco que no afecte al resto del programa. Erlang reduce al mínimo los daños causados de las posibles fallas de los procesos.

- **Escalabilidad.** Poder mantener un elevado número creciente de procesos en ejecución. Los sistemas de telefonía que desarrollaba Ericsson se basaban en tener un proceso por cada llamada entrante, que fuera controlando los estados de la misma. Por lo que se buscaba un sistema que pudiese gestionar desde cientos de miles, hasta millones de procesos.

- **Código en caliente.** Que el sistema no se detenga nunca, incluso cuando haya que realizar actualizaciones. Por lo que Erlang posee la capacidad de modificar el código en caliente, sin necesidad de parar el sistema y sin afectar al código en ejecución.

- **Orientado a la Concurrencia.** Como una especie de nueva forma de programar, este lenguaje se orienta a la concurrencia de manera que las rutinas más íntimas del propio

[2] Erlang hace uso de su coincidencia de patrones, *pattern matching*, la cual sirve para seleccionar entre diferentes casos y para extraer componentes desde estructuras de datos complejas.

lenguaje están preparadas para facilitar la realización de programas concurrentes y distribuidos.

- **Paso de mensajes en lugar de memoria compartida.** Uno de los problemas de la programación concurrente es la ejecución de secciones críticas de código para acceso a porciones de memoria compartida. Este control de acceso acaba siendo un cuello de botella. Para simplificar e intentar eliminar el máximo posible de errores, Erlang/OTP se basa en el paso de mensajes en lugar de emplear técnicas como semáforos o monitores. El paso de mensajes hace que un proceso sea el responsable de los datos y la sección crítica se encuentre sólo en este proceso, de modo que cualquiera que pida ejecutar algo de esa sección crítica, tenga que solicitárselo al proceso en cuestión. Esto abstrae al máximo la tarea de desarrollar programas concurrentes, simplificando enormemente los esquemas y eliminando la necesidad del bloqueo explícito.

4.3. La concurrencia y procesos en Erlang

Erlang fue diseñado desde cero pensando en la *concurrencia* —tener múltiples tareas en ejecución simultáneamente. Este soporte natural para la concurrencia utiliza el concepto de proceso para obtener una separación clara entre las tareas, lo que permite crear arquitecturas tolerantes a fallos y utilizar plenamente el hardware multinúcleo disponible hoy en día [LMC11]. Por lo que el término proceso en el lenguaje de programación Erlang es un concepto fundamental, es la *unidad de concurrencia* en Erlang.

Un *proceso* representa una actividad en marcha, esto es, la ejecución de una pieza de código de un programa y que es concurrente a otro proceso ejecutando también su propio código. La única manera en que los procesos interactúan es a través del *paso de mensajes*, donde el dato enviado va de un proceso a otro [CT09].

Los procesos en Erlang están separados unos de otros y asegurados para no perturbarse entre sí. Lo mismo pasa por ejemplo, cuando se utilizan a la vez aplicaciones tales como el procesador de texto y el navegador web, se ejecutan dentro de un proceso y en caso de que el procesador de texto llegué a fallar y no responda, el navegador web generalmente permanece ejecutándose como si no pasará nada. Esto significa que los procesos son como un tipo de burbuja que proporcionan *aislamiento* con otros procesos [LMC11].

Cada proceso en Erlang tienen su propia memoria de trabajo y un *buzón* para recibir mensajes, mientras que los *hilos* en muchos otros lenguajes de programación y sistemas

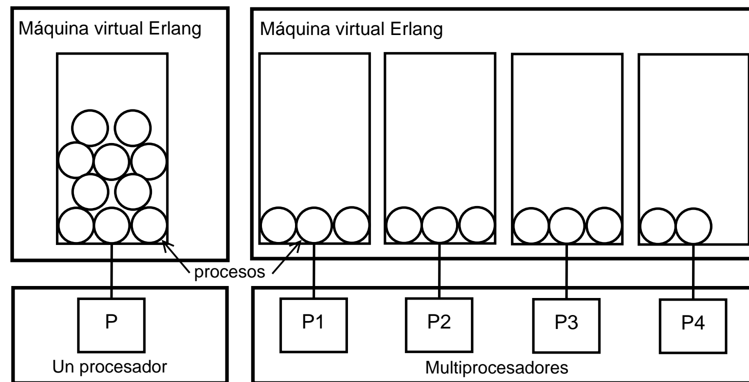

Figura 4.1: Ejecución de procesos Erlang en el hardware

operativos son actividades concurrentes que comparten el mismo espacio de memoria. Los procesos de Erlang puede trabajar con seguridad bajo el supuesto de que ningún otro proceso intervendrá para cambiar sus datos. Entonces se dice que los procesos *encapsulan su estado*, debido a que no puede ser modificado su estado interno directamente por otros. No importa que tan mal sea el código que un proceso esté ejecutando, simplemente no puede corromper el estado interno de los demás.

Cuando un proceso necesita intercambiar información con otro entonces le envía un mensaje. Ese mensaje es una copia de sólo lectura de los datos que el emisor posee. La comunicación de procesos en Erlang siempre trabaja como si el receptor recibiera una copia del mensaje, incluso si el emisor se encuentra en la misma computadora. Esta *distribución transparente* de procesos permite a los programadores Erlang ver a la red como un conjunto de recursos. No interesa mucho acerca de que si el proceso `X` se está ejecutando en un equipo distinto al proceso `Y`, debido a que el método de comunicación es exactamente el mismo sin importar dónde están ubicados. Una de las cosas que también Erlang hace, es que su máquina virtual balancea automáticamente la carga de trabajo sobre los procesadores disponibles. Como se ilustra en la Fig. 4.1 tomada de [LMC11], si se tiene disponible más de un procesador (o núcleos) entonces Erlang hace uso de ellos para ejecutar procesos simultáneamente. Si no, aprovecha al máximo la potencia que hay en el único procesador. Por lo que los programas Erlang automáticamente se adaptan a distinto hardware.

En la siguiente sección, se ofrece una visión general de los métodos de comunicación de procesos utilizados por los diversos lenguajes de programación y sistemas operativos, mencionando las ventajas y desventajas involucradas.

4.4. Paradigmas de comunicación de procesos

El problema central que se presenta en todos los sistemas concurrentes es la de compartir información. Por ello, a lo largo de los años muchos enfoques para solucionar este problema han sido propuestos, algunos han aparecido como características de un lenguaje de programación y otros como librerías separadas. Enseguida se describen tres paradigmas de comunicación de procesos que hoy en día los lenguajes y sistemas operativos están tomando, estos son el de *memoria compartida con bloqueo*, *memoria transaccional por software* y el de *paso de mensajes*.

4.4.1. Memoria compartida con bloqueo

Es el enfoque más antiguo pero que todavía sigue siendo muy popular, ya que actualmente es la principal técnica para comunicar procesos. En este paradigma uno o más celdas de memoria pueden ser leídas o escritas por dos o más procesos. Para hacer esto posible, cuando un proceso realiza una secuencia atómica de operaciones en unas celdas de memoria, ningún otro proceso es capaz de accesar a ninguna de esas celdas hasta que todas las operaciones han sido completadas. Esta podría ser una manera de bloquear procesos para que no accesen a las celdas hasta que hayan sido liberadas. Se realiza mediante un bloqueo (*lock*), que es un constructor que hace posible restringir el acceso para un único proceso a la vez.

La implementación de *locks* requiere del soporte del sistema de memoria, tipícamente de instrucciones especiales al hardware. También requiere de completa cooperación entre los procesos, todos deben asegurarse de preguntar al *lock* antes de pretender accesar a una región de memoria compartida, así como después liberar el bloqueo para que alguien más tenga oportunidad de usarlo. El más mínimo error puede causar estragos.

Generalmente construcciones de alto nivel como semáforos, monitores y exclusiones mutuas están basados en estos bloqueos simples. Se utilizan como llamadas al sistema operativo o en los lenguajes de programación para hacer esta tarea más fácil y garantizar que los bloqueos sean apropiadamente solicitados y liberados. Aunque todo esto evita muchos problemas, los bloqueos todavía tienen ciertos inconvenientes:

- Requieren de muchos recursos incluso cuando las probabilidades de colisiones son bajas.
- Son puntos de contención en el sistema de memoria.
- Pueden dejar estados bloqueados por procesos fallidos.

- Es muy difícil depurar problemas empleando los bloqueos.

Por otra parte, el bloqueo puede funcionar bien para la sincronización de dos o tres procesos, pero a medida que el número aumenta la situación se vuelve inmanejable. Existe una posibilidad real de terminar con un interbloqueo (*deadlock*) complejo que no haya sido previsto.

Algunos autores consideran que esta forma de comunicación es mejor emplearla para la programación de bajo nivel, como en el núcleo (*kernel*) del sistema operativo. Sin embargo se encuentra en la mayoría de los lenguajes de programación populares.

4.4.2. Memoria transaccional de software

El mecanismo STM (*Software Transactional Memory*) actualmente se puede encontrar en la aplicación GHC[3] del lenguaje de programación Haskell, así como en el lenguaje Clojure[4] basado en la máquina virtual de Java. STM trata a la memoria como una base de datos tradicional empleando transacciones para decidir qué se escribe y cuándo.

Normalmente la implementación trata de evitar el uso de bloqueos trabajando de una manera optimista: una secuencia de accesos de lectura y escritura son tratados como una única operación y si dos procesos intentan acceder a una región compartida al mismo tiempo, cada uno en su propia transacción, sólo uno de ellos tendrá éxito. Tras comprobar los nuevos contenidos producto de esa transacción, a los otros procesos se les dice que fallaron y que deben volver a intentarlo. Este modelo no exige a ningún proceso esperar a que alguien más mientras se libera un bloqueo.

El principal inconveniente de este paradigma es el que se tengan que volver a intentar las transacciones fallidas y que podrían volver a fallar en repetidas ocasiones. Otro es que existe sobrecarga significativa involucrada con el sistema de transacciones en sí, así como la necesidad de tener memoria adicional para almacenar los datos que se están tratando de escribir mientras se decide por el proceso que tendrá éxito.

Para los programadores el enfoque STM parece más manejable que el uso de bloqueos y puede ser una buena forma de tomar ventaja de la concurrencia. Se puede considerar este enfoque una variante de la memoria compartida con bloqueos, que puede ser de más ayuda a nivel de sistema operativo que a nivel de programación de aplicaciones, actualmente es un tema de investigación abierto.

[3] GHC es un compilador y entorno interactivo para el lenguaje funcional Haskell, www.haskell.org/ghc.

[4] Lenguaje funcional inspirado en Lisp, www.clojure.org.

4.4.3. Paso de mensajes

Como ya se ha mencionado, la comunicación de procesos en Erlang se realiza por medio de paso de mensajes, donde el *proceso receptor* obtiene una copia del dato y nada hace que la copia sea observable por el *proceso emisor*. La única manera de comunicar la información de vuelta al emisor es enviar otro mensaje, en dirección inversa. Una de las consecuencias más importantes es que la comunicación funciona de la misma manera si el emisor y el receptor están en el mismo equipo o si están separados por una red.

El paso de mensajes en general se da de dos formas: *síncrona* y *asíncrona*. En la forma síncrona el emisor no puede hacer nada más hasta que el mensaje haya llegado al receptor. Mientras que en la forma asíncrona el emisor pueden proceder inmediatamente después de enviar el mensaje. En el mundo real, la comunicación síncrona entre máquinas separadas sólo es posible si el receptor envía un acuse de recibo al emisor avisando que todo está bien para que continúe, pero este detalle se puede mantener oculto al programador.

En Erlang las primitivas de paso de mensajes son asíncronas, a menudo el emisor no necesita saber si el mensaje llegó. Este método de comunicación asíncrono implica que el emisor no tiene que ser suspendido mientras se está entregando el mensaje, en particular si se envía el mensaje a través de un enlace de comunicación lento.

Conseguir este nivel de separación entre emisor y receptor tampoco es fácil. La copia de datos puede ser costosa para las grandes estructuras y puede causar mayor uso de memoria si el emisor también necesita mantener una copia de los datos. En la práctica, esto significa que se debe tener en cuenta la administración del tamaño y la complejidad de los mensajes que se están enviando. En programas normales de Erlang la mayoría de los mensajes son pequeños y la sobrecarga de la copia es normalmente insignificante.

4.5. Trabajando con procesos en Erlang

Para construir un programa en Erlang que involucre procesos es necesario definir primero qué actividades son concurrentes, cuáles pueden suceder de forma independiente una de otra y entonces por cada instancia de esas actividades que se hayan identificado convertirlas en un proceso. En contraste con la mayoría de otros lenguajes de programación, la concurrencia en Erlang es barata en cuanto a recursos de hardware como memoria y procesador [LMC11].

Por ejemplo, si se utilizara Erlang para escribir un servidor de mensajería instantánea (IM, *Instant Messaging*) para el envío de mensajes entre los miles de usuarios tal como los

sistemas Google Talk o Facebook, debido a la filosofía de diseño de Erlang, se generaría un nuevo proceso por cada evento. En un sistema de IM, un evento podría ser la actualización de estado en línea, un mensaje enviado, un mensaje recibido o una solicitud de inicio de sesión, y esto por cada uno de los múltiples usuarios. Cada proceso daría servicio al evento y terminaría cuando la solicitud se haya completada. Es posible hacer lo mismo también en lenguajes como C o Java, pero se vuelve muy difícil manejar todo al escalar el sistema a cientos de miles de eventos simultáneos [CT09]. Erlang no utiliza subprocesos nativos para representar los procesos, sino que tiene su propio gestionador en su máquina virtual, haciendo la creación de procesos muy eficiente y al mismo tiempo reduce al mínimo su contacto con la memoria. Esta eficiencia se mantiene independientemente del número de procesos concurrentes en el sistema. El mismo argumento aplica para el paso de mensajes, donde el tiempo para enviar un mensaje es despreciable y constante, sin importar el número de procesos.

Los procesos en Erlang son ligeros, es decir, utilizan muy poca memoria, pertenecen al lenguaje de programación y no al sistema operativo, no comparten memoria y son totalmente independientes. La única manera que interactúan es através del paso de mensajes, por eso algunas veces Erlang es llamado un *lenguaje de paso de mensajes puro*. Permite crear y destruir procesos, enviar mensajes entre procesos y tener un gran número de procesos sin problemas. Para la programación de procesos en Erlang se necesita de las siguientes tres acciones: lanzar o crear(*spawn*) procesos, enviar(*send*) y recibir(*receive*) mensajes [Arm07].

4.5.1. Creación de procesos

Para ejecutar código concurrente es necesario crear varios procesos. Esto se hace [Arm07] usando la función incorporada (*BIF*) `spawn(mod, funcion, Args)`, la cual crea un nuevo proceso concurrente que evalúa la función `funcion` que fue declarada como exportada en el módulo `mod` y donde `Args` es de tipo lista que contiene como elementos los parámetros de la función `funcion`. Los argumentos `mod` y `funcion` son de tipo átomo.

El nuevo proceso creado se ejecuta simultáneamente con el proceso que lo lanzó. La función `spawn` retorna un tipo de dato identificador de proceso (*pid*), revisar la sección A.3.9. Lo anterior se visualiza en la Fig. 4.2 adaptada de [CT09] donde el proceso `Pid1` ejecuta la función `spawn` en algún punto del programa, esta llamada resulta en un nuevo proceso con identificador `Pid2` que a su vez ejecutará la función del módulo con los argumentos proporcionados en el proceso `Pid1`. Típicamente cuando se hace uso de la función `spawn`, su retorno se le vincula a una variable:

```
1 Pid2 = spawn(mod, funcion, [arg1, arg2, ...]).
```

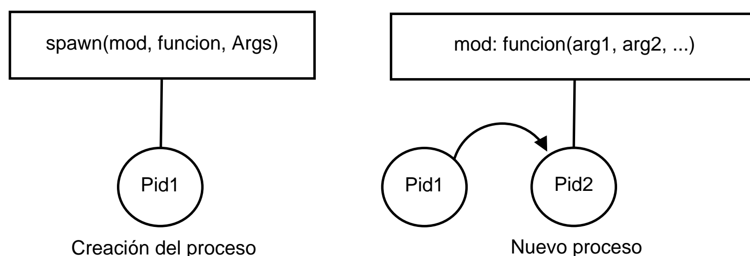

Figura 4.2: Lanzando un proceso Erlang

El nuevo proceso `Pid2` en este punto es sólo conocido dentro del proceso `Pid1`, ya que es una variable local que no ha sido compartida. Así, el proceso lanzado comienza la ejecución de la función exportada (segundo argumento) del módulo pasado como primer argumento y teniendo como argumentos el contenido de la lista pasada como tercer argumento en `spawn`. Una vez que el proceso es lanzado continuará ejecutándose y permanecerá vivo hasta que termine. Un proceso *termina* cuando no hay más código que ejecutar, entonces se dice que terminó de forma *normal*. Su memoria de trabajo, buzón de correo y otros recursos se reciclan. Si el propósito del proceso es producir datos para otro proceso, entonces debe enviar los datos de forma explícita en un mensaje antes de que termine su ejecución. En caso de haber un error en tiempo de ejecución se dice que terminó de forma *anormal* y otros procesos pueden ser informados de la excepción.

A diferencia de un hilo típico de un sistema operativo moderno que reserva algunos megabytes de espacio de direcciones de memoria y en caso de que llegué a necesitar más es muy probable que se produzca un error, los procesos Erlang comienzan con sólo un par de cientos de bytes de espacio de memoria, la cual crece o se contrae automáticamente según sea necesario. Como ya se mostró, la sintaxis para la creación de procesos en Erlang es sencilla, aquí el clásico "Hola Mundo" empleando un proceso de Erlang que consiste solamente en ejecutar la impresión de una cadena en consola [LMC11]:

```
1 Pid2= spawn(io,format,[``Hola Mundo'']).
```

Se invoca a la función `format` que se encuentra en el módulo de entrada y salida `io`, y recibe como parámetro la cadena `‘‘Hola Mundo’’`. Cabe señalar que la función `spawn` tiene otras variantes aceptando un sólo argumento. Este se puede tratar de la definición de una función *fun* (revisar la sección A.6) o simplemente otra forma de llamar a una función:

```
1 Pid3= spawn(fun() -> ... end) %definiendo una función
2 Pid4= spawn(fun modulo:funcion/n) %n es el número de argumentos
```

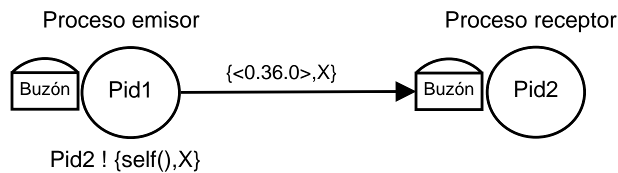

Figura 4.3: Paso de mensajes entre procesos Erlang

Para conocer los procesos que Erlang está ejecutando actualmente se utiliza la función BIF `processes()` que retorna una lista de ellos. Es muy útil en aplicaciones industriales donde se trabaja con millones de procesos que se ejecutan al mismo tiempo. El número predeterminado de procesos se encuentra en el rango de los cientos de millones, sin embargo puede modificarse este valor iniciando la consola Erlang con los siguientes parámetros: `erl + P MaxProcceses` donde `MaxProcesses` es un entero que representa el máximo número de procesos [CT09].

4.5.2. Envío de mensajes

Los procesos Erlang se comunican mediante el paso de mensajes. La construcción para enviar un mensaje en Erlang es con el *operador envío* binario infijo !/2 [Arm07]: `Pid ! Mensaje`, donde `Pid`—de tipo de dato *pid*—es el identificador del proceso receptor y `Mensaje` es el mensaje a enviar al receptor, el cual puede ser de cualquier tipo de dato[5] Erlang.

Cada proceso Erlang posee un buzón de correo en el que los mensajes entrantes se almacenan. Cuando se envía un mensaje, éste se copia desde el proceso emisor al buzón del receptor. Los mensajes se almacenan en el buzón en base al orden en el que llegan. En caso de envíar dos mensajes de un proceso a otro, los mensajes se reciben en el mismo orden en que se envían. En la Fig. 4.3 adaptado de [CT09] se ilustra el envío del mensaje `{self(),X}` del proceso `Pid1` a `Pid2`, el cual es una tupla con dos elementos: el primero de ellos es la función BIF `self()` que retorna el *pid* del proceso actual y el segundo elemento es una variable `X`.

El envío de un mensaje en Erlang está garantizado ya que no fallará; si se envía un mensaje a un proceso que no existe el mensaje se desecha sin generar ningún error. Recordar que el paso de mensajes en Erlang es asíncrono: un proceso emisor no se suspenderá después de enviar un mensaje, sino que continuará inmediatamente la ejecución de la siguiente expresión de su código independientemente del mensaje enviado.

La expresión `Pid ! Mensaje` devuelve un valor, tal como todas las expresiones válidas

[5]Puede ser de tipo átomo, número entero o real, cadena de texto, tupla, etc. Para más información revisar la sección A.3 donde se presentan los tipos de datos Erlang.

Erlang. En este caso es el mismo mensaje enviado, `Mensaje`. Así que si, por ejemplo, se necesita enviar el mismo mensaje a muchos procesos se puede escribir ya sea como una secuencia de envío de mensajes `Pid1!Msj, Pid2!Msj, Pid3!Msj` o en una sola expresión como `Pid3!Pid2!Pid1!Msj`, que es equivalente a escribir `Pid3!(Pid2!(Pid1!Msj))` donde `Pid1!Msj` retorna el mensaje para enviarlo a `Pid2`, que a su vez devuelve el mensaje que se enviará a `Pid3`.

La razón por la que el paso de mensajes y la creación de procesos siempre tienen éxito, incluso si el proceso receptor no existe o el proceso lanzado falla al crearse, tiene que ver con la *falta de dependencias entre los procesos*. Se dice que un proceso A *depende* de un proceso B cuando la terminación del proceso B puede impedir el funcionamiento correcto del proceso A. Las dependencias de procesos son muy importantes y a menudo influyen en el diseño. En sistemas masivamente concurrentes no se desea que los procesos dependan unos de otros a menos que se especifique de manera explícita y en tales casos siempre se busca que sea el menor número de dependencias posibles. Por ejemplo, considerar un servidor de IM manejando al mismo tiempo miles de mensajes que intercambian sus usuarios. Cada mensaje es un proceso generado para esa función en particular. Si debido a un error uno de estos procesos termina se perdería ese mensaje en particular. El asegurar la falta de dependencia entre este proceso y los demás procesos que se manejan como mensajes garantiza que estos últimos se procesen y entreguen a sus destinatarios de forma segura, independientemente del error [CT09].

4.5.3. Recepción de mensajes

Los mensajes se recuperan del buzón del proceso empleando la cláusula `receive`. Esta cláusula es una construcción delimitada por las palabras reservadas `receive` y `end`. Dentro contiene una serie de cláusulas que son similares a las que se utilizan en la definición de una función o en una estructura `case`: a la izquierda de la flecha un patrón (que puede o no tener guardias) y a la derecha una secuencia de expresiones, tomando la sintaxis siguiente [Arm07]:

```
1 receive
2     Patron1 [when Guardia1] -> Expresion11, .., Expresion1n;
3     Patron2 [when Guardia2] -> Expresion21, .., Expresion2n;
4     ...
5     Otro                    -> Expresion1, .., Expresionn
6 end
```

Cuando un proceso ejecuta la sentencia `receive`, en caso de que no haya ningún mensaje en el buzón, se está a la espera de la llegada de algún mensaje de otro proceso, es decir, la ejecución de más código no prosigue sino que se mantiene *escuchando*. Pero si hay más de un mensaje, toma el primer mensaje (el más antiguo) que haya llegado al buzón del proceso y se compara contra cada patrón de cada una de las cláusulas [CT09]:

- Si ocurre una *coincidencia correcta* entonces el mensaje se recupera del buzón de correo y las variables del patrón se vinculan con las partes correspondientes del mensaje, enseguida el cuerpo de la cláusula es ejecutada, las expresiones.

- Si ninguna de las cláusulas coinciden con el mensaje, los mensajes subsiguientes en el buzón son comparados uno a uno contra todas las cláusulas hasta que un mensaje coincide con una cláusula o que todos los mensajes hayan fracasado con todas las posibles coincidencias de patrones.

Considerar el siguiente ejemplo: el mensaje {`reset,151`} se envía a un proceso que ejecuta la instrucción `receive` y que coincide con la primera cláusula:

```
1 receive
2     {reset, N} -> reset(N);
3     Default    -> {error, aviso}
4 end
```

Esto hace que se vincule la variable `N` al número entero `151` y entonces se ejecuta la función `reset(151)`. Ahora suponer que dos nuevos mensajes—`restart` y {`reset, 151`}—se envían en ese orden al proceso. Tan pronto como el flujo de ejecución del proceso llegue a la sentencia `receive`, esta tratará de hacer coincidir el mensaje más antiguo en el buzón, `restart`, el cual no coincidirá con la primera cláusula pero sí con la segunda, vinculando la variable `Default` con el átomo `restart`. La sentencia `receive` retornará la última expresión evaluada en el cuerpo de la cláusula que haya coincidido. En este caso, con el primer mensaje retornará una tupla y con el segundo lo que devuelva la función `reset/1`. Para asegurar que la cláusula `receive` siempre recupere el primer mensaje del buzón se puede utilizar una variable no ligada, como `Default` en el ejemplo, esto hará que cualquier mensaje entrante si no coincide con ningun patrón entonces "caerá" en esta cláusula vinculándose a la variable.

Notar que el mensaje no precisamente siempre tiene que ser del mismo tipo de dato sino que puede tomar varias formas, en el ejemplo, la primera cláusula recibe un mensaje de tipo tupla y la segunda tiene una variable que esta abierta a aceptar cualquier tipo de dato Erlang, a esto se le denomina *recepción selectiva* [CT09]. En la programación de procesos es

muy común que los mensajes la mayoría de las veces sean de tipo tupla pero, como ya se vió, no es una restricción. El mensaje `restart` pudo ser enviado como una tupla con un único elemento y también encajaría con la segunda cláusula, sin embargo esto se considera una mala práctica de programación ya que consume más memoria y es más lento su manejo en el proceso. La directriz es que si la tupla tiene un solo elemento entonces utilizar el elemento por sí solo, sin la construcción de la tupla. En caso de que un mensaje no satisfaga a ninguna de las cláusulas de la sentencia `receive` el proceso se *suspende* hasta que un mensaje coincida, algo que no pasa en una estructura `case` ya que aquí se produce un error en tiempo de ejecución.

4.5.3.1. Recepción con tiempo de espera

Una instrucción `receive` podría quedarse esperando indefinidamente un mensaje que nunca llegue. Esto puede ser por varias razones. Por ejemplo, puede haber un error de lógica en el programa, o el proceso que va a enviar un mensaje podría haber fallado antes de hacerlo. Para evitar este problema, es posible añadir un tiempo de espera (*timeout*) a la sentencia `recieve`. Esto se hace con la construcción `after` como una cláusula más al final de todas. Con ello se establece un tiempo máximo de espera que el proceso tendrá para recibir un mensaje. La sintaxis es la siguiente:

```
1 receive
2     Patron1 [when Guardia1] -> Expresion11, .., Expresion1n;
3     Patron2 [when Guardia2] -> Expresion21, .., Expresion2n;
4     ...
5     after Tiempo -> Expresion1, .., Expresionn
6 end
```

La variable `Tiempo` es un número entero que denota el tiempo en milisegundos o en su caso utilizar el átomo `infinity`. Al usar `infinity` aquí es como si `after` no se incluyera en `receive` y el proceso esperaría indefinidamente. Cuando un proceso llega a la sentencia `receive` y no hay mensajes que coincidan con ningún patrón, entonces esperará los milisegundos indicados. Si después de pasar ese tiempo de espera no ha llegado ningún mensaje, entonces se ejecutarán las expresiones en `after`.

4.5.3.2. Modo de trabajo de la cláusula *receive*

Como ya se ha visto, en realidad el operador `!` no envía un mensaje directamente a un proceso sino al buzón de correo de este, para con la construcción `receive` se trata de remover

un mensaje del buzón. La única vez que se examina el buzón es cuando el programa evalúa la sentencia `receive`, que trabaja de la siguiente manera [Arm07]:

1. Cuando se comienza a ejecutar una sentencia `receive` un temporizador inicia, siempre y cuando una sección `after` esté presente en la construcción.

2. Se toma el primer mensaje en el buzón y se compara contra cada uno de los patrones de las cláusulas, en caso de coincidir con alguno, el mensaje se elimina del buzón de correo y se evalúan las expresiones de la cláusula.

3. Si ninguno de los patrones coincide con el primer mensaje del buzón entonces el mensaje se elimina del buzón y se guarda en una cola especial interna. El segundo mensaje en el buzón se toma ahora. Este procedimiento se repite hasta que un mensaje coincida o hasta que se hayan examinado todos los mensajes en el buzón.

4. Si ninguno de los mensajes del buzón coincide con ningún patrón entonces el proceso se suspende y se recalanderiza su ejecución hasta la próxima vez que un nuevo mensaje llegue al buzón. Cuando esto pase, únicamente se compara el nuevo mensaje contra todos los patrones y los mensajes que no hayan coincidido antes, que fueron colocados en la cola, no se comparan.

5. Tan pronto como un mensaje haya coincidido, entonces todos los mensajes que se han puesto en la cola son puestos otra vez en el buzón en el orden en que llegaron al proceso. En caso de que se haya programado un temporizador en la sentencia `receive` entonces los mensajes en la cola ya no se pasan al buzón sino se borran.

6. Si se tiene un temporazidor y el tiempo termina mientras se está a la espera de un mensaje, entonces se evalúan las expresiones de la sección `after` y los mensajes guardados en la cola de nueva cuenta se colocan en el buzón en el orden en que llegaron al proceso.

4.5.4. Registro de procesos

No siempre es práctico utilizar los *pids* para comunicar procesos, debido que para utilizar un *pid* en un proceso, este debe ser notificado de ello y almacenar ese valor. Es común *registrar* procesos que ofrecen servicios específicos con un *alias*: un nombre que puede ser utilizado en lugar del *pid*. Un proceso se registra con la función BIF `register(alias, Pid)`, donde `alias` es un átomo y `Pid` es un identificador de proceso.

Una vez que un proceso ha sido registrado, cualquier proceso puede enviar un mensaje al mismo sin tener en cuenta su identificador. Así, para enviar un mensaje a un proceso registrado lo único que se necesita es reemplazar el *pid* por el alias en el constructor !/2, lo cual quedaría: `alias!Mensaje`. Otras funciones BIFs que están relacionadas con el registro de procesos son: `unregister(Pid)`, elimina el alias con que se registro un proceso; `registered()`, devuelve una lista de nombres de procesos registrados y `whereis(alias)` que retorna el *pid* asociado al alias. Un ejemplo del uso de la función `register/2` en consola sería::

```
1> Pid = spawn(nomina, calcularsalario, []).
<0.51.0>
2> register(salario, Pid), salario!{Horas,SalarioMinimo}.
true
```

4.5.5. Ejemplos de procesos Erlang

A continuación se presentan dos ejemplos que muestran la creación de procesos y la interacción con otros a través del envío y recepción de mensajes. Estos ejemplos son el cálculo del área de varias figuras y el eco de un mensaje entre dos procesos.

4.5.5.1. Calcular el área de una figura

Un ejemplo sencillo ha considerar es el cálculo del área de distintas figuras. En un módulo llamado `geometria` se define la función `area/0` que implementa la construcción `receive`:

```
1  -module(geometria).
2  -export([area/0]).
3
4  area() ->
5     receive
6        {rectangulo, Alto, Ancho} ->
7           io:format(``El área del rectángulo es ~p~n'', [Alto * Ancho]),
8           area();
9        {circulo, Radio} ->
10          io:format(``El área del circulo es ~p~n'', [math:pi()*Radio*Radio]),
11          area();
12       finArea -> io:format(``El proceso que calcula areas termina.'');
13       Otro ->
14          io:format(``El área de ~p no se ha implementado.'', [Otro]),
15          area()
16    end.
```

Desde la consola se crea un proceso que evalúa la función `area/0` y se le envían mensajes:

```
1> PidArea= spawn(geometria, area, []).
<0.48.0>
2> PidArea ! {rectangulo,6,10}.
El área del rectángulo es 60.
{rectangulo,6,10}
3> PidArea ! {circulo,23}.
El área del circulo es 1661.9025137490005.
{circulo,23}
4> PidArea ! {triangulo,4,9}.
El área de {triangulo,2,9} no se ha implementado.
{triangulo,2,9}
5>  PidArea ! [o,t,r,a,c,o,s,a].
El área de [o,t,r,a,c,o,s,a] no se ha implementado.
[o,t,r,a,c,o,s,a]
6> PidArea ! finArea.
El proceso que calcula areas termina.
finArea
7> PidArea ! {rectangulo,2,4}.
{rectangulo,2,4}
```

En la línea 1 con la función `spawn` se crea un nuevo proceso y esta retorna un *pid* que es vinculado a la varibale `PidArea`. El nuevo proceso evaluará la función `area/0`. En las líneas 2 y 3 se envían dos mensajes que coinciden con la primera y segunda cláusula de la sentencia `receive`, que son el cálculo del área del rectángulo y circulo respectivamente. Ambas clásulas invocan a la función `area/0` otra vez en la última expresión por lo que el proceso se sigue ejecutando. El tercer mensaje (línea 4) encaja con la cuarta cláusula vinculándose con la variable `Otro`, ya que es una tupla que no coincide con los otros tres patrones. También se vuelve a llamar a la misma función en la última expresión.

En el cuarto mensaje, en lugar de un tipo tupla se envía uno de tipo lista el cual no coincide con ninguno de los primeros tres patrones y sí con el cuarto patrón. El quinto mensaje es el átomo `finArea` el cual coincide fielmente con el tercer patrón, esto hace que se imprima un aviso de terminación del proceso. Puesto que ya no se vuelve a llamar a la función recursivamente el proceso termina. Finalmente, en el sexto mensaje se envía un mensaje que encajaría con el primer patrón pero como el proceso ya ha terminado entonces ya no se evalúa, sin embargo tampoco se generá un error. Notar que aún así el operador de envío retorna el mensaje que se pretendió enviar.

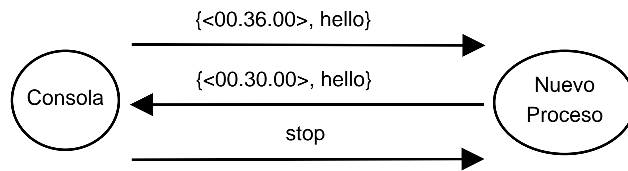

Figura 4.4: Eco de un mensaje

4.5.5.2. Eco de un mensaje

Este ejemplo consiste en el envío de un mensaje desde un proceso A a un proceso B, el proceso B toma el mensaje y lo regresa al proceso A. La consola de Erlang (por sí misma es un proceso) fungirá como el proceso A y el proceso B será creado para estar a la espera del mensaje. La Fig. 4.4 muestra la secuencia de envío del mensaje. El código es el siguiente:

```
-module(echo).
-export([go/0, loop/0]).
go()->
    Pid= spawn(echo,loop,[]),
    Pid ! {self(),hello},
    receive
      {Pid, Msg}-> io:format(``~w~n'',[Msg])
    end,
    Pid ! stop.

loop()->
    receive
      {From,Msg}-> From ! {self(),Msg},
                   loop();
      stop-> true
    end.
```

La función `go/0` será invocada desde la consola y lanzará un nuevo proceso que ejecute la función `loop/0`, enseguida le enviará un mensaje en forma de tupla con dos elementos: el *pid* de la consola y el átomo `hello`, para después entrar a una cláusula `receive` a la espera de un mensaje que únicamente imprimirá. La última instrucción es el envío del mensaje `stop` también al proceso creado. Por su parte la función `loop/0` estará a la espera de dos posibles mensajes: uno en forma de tupla con dos elementos y el otro el átomo `stop`. Para el caso del primer patrón, ejecutará el envío del segundo elemento de la tupla al proceso con *pid* `From` y llamará a la función `loop` para estar a la escucha. Si el mensaje es el átomo `stop` la ejecución de la función termina devolviendo `true`. Finalmente la ejecución del programa en la consola sería:

```
1> c(echo).        % compilación del módulo
{ok,echo}
2> echo:go().      % ejecución de la función
hello
stop
```

Con estos ejemplos se pretendió mostrar la forma en que los procesos se comunican y la importancia que tiene la coincidencia de patrones para que un proceso reciba mensajes, que pueden ser de cualquier tipo de dato Erlang.

4.6. Arquitectura Cliente/Servidor

El envío de mensajes entre los procesos de Erlang puede realizarse de diferentes maneras dependiendo del problema, sin embargo en varios casos habrá semejanzas en cómo se manejan estos. A esas similitudes se les llama *patrones de diseño*, tal como los hay también en el paradigma orientado a objetos [SM02]. Uno de los patrones de diseño comúnmente empleado [ÖV11, Som11, SGG12] en los sistemas distribuidos es el de la arquitectura Cliente/Servidor. En Erlang es posible implementarla encargando a los procesos que sean responsables de un *recurso*, por ejemplo una lista de habitaciones de un hotel y los servicios que se pueden aplicar sobre estos: reservación o verificación de disponibilidad de una habitación. Las solicitudes al *servidor* permitirán a los *clientes* acceder a los recursos y servicios; ambos son generalmente implementados como procesos Erlang.

De acuerdo a lo consultado en [Arm07, CT09, LMC11] enseguida se describe la implementación de una arquitectura Cliente/Servidor en Erlang. Un servidor puede ser una cola FIFO de una impresora, un gestor de ventanas o un servidor de archivos. Los recursos que podría manejar serían una base de datos, un calendario o una lista finita de elementos tales como boletos o libros. Los clientes utilizan estos recursos mediante el envío de peticiones al servidor para imprimir un archivo, actualizar una ventana, reservar un boleto o comprar un libro. El servidor recibe la petición, la maneja y entonces responde con una confirmación y retorna un valor si la solicitud se ha realizado exitosamente o con un error si la solicitud no tuvo éxito. En la Fig. 4.5 adaptado de [CT09] se ilustra este patrón representado por los procesos Erlang, que implica tener una red que separa a los clientes de un servidor el cual atiende sus solicitudes y les responde a cada uno de ellos. Como en la figura, muy a menudo se tienen varias instancias del cliente y un sólo servidor. El cliente y el servidor se implementan en Erlang como procesos separados y para que estos se comuniquen utilizan el

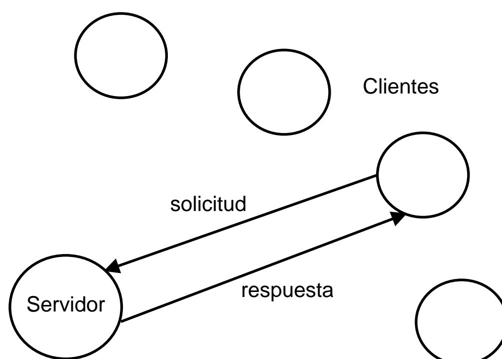

Figura 4.5: Arquitectura Cliente/Servidor mediante procesos Erlang

paso de mensajes de Erlang. Tanto el cliente y el servidor se pueden ejecutar en la misma máquina o en dos máquinas diferentes. Las palabras *cliente* y *servidor* se refieren a los roles que estos dos procesos tienen; el cliente siempre inicia mediante el envío de una solicitud al servidor, a su vez el servidor cómputa una respuesta y la envía al cliente.

El cliente tiene que estar preparado para afrontar un retraso en el tiempo de respuesta o tener en cuenta que las cosas pueden salir mal en la solicitud al servidor, ya que puede darse un fallo en la red, el proceso servidor puede colgarse o pueden haber tantas solicitudes que los tiempos de respuesta del servidor se vuelven inaceptables. Cuando un cliente utiliza un servicio o recurso manejado por el servidor entonces espera una respuesta a la petición, por lo que la llamada al servidor es síncrona. Pero si el cliente no necesita una respuesta, la llamada al servidor es asíncrona.

Para las llamadas de tipo síncrona se pueden esperar dos posibles respuestas del servidor: éxito y error. Para las solicitudes que se hayan hecho exitosamente se devolverá, además de la respuesta en sí que requiere el cliente, un identificador de éxito. Sin embargo para las solicitudes que por algún motivo fracasen se devolverá un identificador de error y el motivo de esto. Aunque esto no es mandatorio, se sugiere que la estructura de los mensajes se realicen mediante una tupla Erlang por lo que estos dos quedarían de la siguiente forma: `{ok, Respuesta}` donde el primer elemento de la tupla es el átomo `ok` indicando el éxito y el resto de elementos de la tupla sería la respuesta correspondiente que necesita el cliente, mientras que el átomo `error` indicaría fracaso al cual se le anexa el motivo, `{error, Motivo}`.

Retomando el ejemplo presentado en la sección 4.5.5.1, el cálculo del área de una figura, se implementará ahora empleando la arquitectura Cliente/Servidor. En ese ejemplo se envía una solicitud al proceso que ejecuta la función `area` que se encuentra en el módulo `geometria`. Esta función calcula el área de una figura en particular. Dicho proceso recibe los valores e

imprime el valor del área dentro de la misma función `area`. Ahora lo que se pretende es enviar como respuesta el valor calculado del área al proceso cliente que envió la solicitud original, y ya no imprimirla.

Se necesitará también agregar dos funciones más al módulo `geometria`, estas son `inicio` y `calcular`. La primera de ellas lanzará el proceso servidor registrando a este como `servidor`. Mientras que la función `calcular` hará las peticones al servidor. Dicha función será invocada desde la consola de Erlang, que fungirá como el proceso cliente. El código del módulo quedaría de la siguiente forma:

```
-module(geometria).
-export([inicio/0, area/0, calcular/1]).

inicio()-> register(servidor, spawn(geometria,area,[])).

area()->
   receive
     {PidCliente, {rectangulo, Alto, Ancho}} ->
         Respuesta=Alto*Ancho,
         PidCliente ! {ok, Respuesta},
         area();
     {PidCliente, {circulo, Radio}} ->
         Respuesta=math:pi()*Radio*Radio,
         PidCliente ! {ok, Respuesta},
         area();
     {PidCliente, Otro}->
         PidCliente ! {error, {''No implementado.'', Otro}},
         area()
   end.

calcular(Solicitud)->
   servidor ! {self(), Solicitud},
   receive
     {ok, Resultado}-> io:format(''Área=~w~n'',[Resultado]);
     {error, Motivo}-> io:format(''Error: ~p~n'',[Motivo])
   end.
```

Se modificó la función `area` para que con los parametros que le llegan haga el calculo, lo ligue a la variable `Respuesta` y enseguida lo retorne al cliente, esto para el caso de los primeros dos patrones. Además, junto con la respuesta envía el identificador `ok`. Cuando no coincida la petición con estos dos patrones (calcular el área de un rectángulo y de un circulo) entonces enviará un error con una descripción. Por otro lado la función `calcular/1` recibe

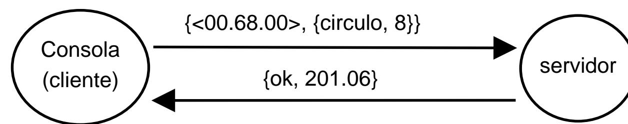

Figura 4.6: Ejemplo de petición y respuesta en la arquitectura Cliente/Servidor

un parámetro que debería ser una tupla, como lo espera el servidor, y enseguida lo envía al servidor adjuntando su *pid* obtenido de la función `self`. Con la cláusula `receive` se pone a la espera de la respuesta del servidor, que como se había mencionado antes generalmente son de dos tipos. Cuando la respuesta llega la imprime.

En la Fig. 4.6 se muestra el envío de un mensaje al servidor y la correspondiente respuesta de este. Para probar este ejemplo se hace a través de la consola de Erlang y algunos mensajes se pueden enviar de la siguiente manera:

```
1> c(geometria).
{ok,geometria}
2> geometria:calcular({circulo, 8}).
Área=201.06
ok
3> geometria:calcular({cuadrado, 105}).
Error: {"No implementado. ",{cuadrado, 105}}
ok
```

En el siguiente capítulo se tomará como ejemplo a la arquitectura Cliente/Servidor para dar una introducción a la notación del cálculo de eventos propuesta para la programación concurrente en Erlang.

Capítulo 5

Una notación para la programación concurrente

En este capítulo en primera instancia se presenta una aproximación de la notación del cálulo de eventos empleando la arquitectura Cliente/Servidor que sirva como preámbulo para conocer la forma de aplicarla. Enseguida se muestra la sintaxis de los elementos de la notación para la Modelación de Procesos Erlang (*ProME*, *notation for Process Modeling Erlang*) dando algunos ejemplos para cada uno de ellos. Para después presentar la gramática de la notación ProME en la forma Backus Naur extendida. La notación ProME se basa en las acciones principales que se pueden hacer con los procesos de Erlang: crear procesos y, enviar y recibir mensajes desde procesos. En el último apartado se presenta un caso de estudio de un sistema enseñanza-aprendizaje en línea que se modela con la notación ProME y se obtiene la implementación correspondiente en Erlang.

5.1. Las bases de la notación

En capítulos anteriores ya se han abordado los fundamentos y conceptos claves de temas como el cálculo de eventos, sistemas distribuidos, programación concurrente en Erlang y la necesidad de la especificación en el desarrollo de software. A manera de resumen, enseguida se presentan los temas de programación concurrente en Erlang y el cálculo de eventos resaltando aquellos elementos de interés en la tesis.

5.1.1. Programación concurrente

Reuniendo lo expuesto en el apéndice A y el capítulo 4 es posible mostrar las relaciones existentes entre los módulos, funciones, procesos y mensajes de Erlang. En primer lugar, la edición de programas en Erlang se lleva a cabo en archivos con extensión `erl` cuyo nombre tiene que ser el mismo al módulo que será creado dentro de este. Es decir, se requiere por cada módulo a programar crear su correspondiente archivo. Las funciones se crean dentro de los módulos. Por otro lado, cuando se crean procesos estos ejecutan una función que se encuentra en algún módulo; la comunicación entre procesos se da por medio del paso de mensajes. La Fig. 5.1 ilustra estas relaciones. Dado que estos conceptos son fundamentales para el tema de tesis y también claves para la programación concurrente en Erlang se da un breve resumen:

- *Módulos*–Los módulos son contenedores de código. Guardan el acceso a las funciones haciéndolas privadas o exportándolas para su uso público. Si un módulo se desea que se llame `test`, este debe residir en un archivo `test.erl` y cuando se compile se obtendrá un archivo objeto denominado `test.beam`.

- *Funciones*–Las funciones hacen todo el trabajo, cualquier código Erlang en un módulo debe ser parte de una función, lo que sería la parte secuencial de Erlang. Mientras la parte concurrente se da cuando las funciones son ejecutadas por procesos. Una función debe pertenecer a algún módulo.

- *Procesos*—Los procesos son la unidad fundamental de la concurrencia en Erlang. Se comunican entre sí a través de mensajes. Cualquier proceso puede crear (*spawn*) otro proceso, especificando la función que deberá ejecutar. El nuevo proceso ejecuta esa llamada y termina cuando la función no tenga más código que ejecutar. Por ejemplo, un proceso que ejecute la función `io:format/2` será de corta duración ya que después de imprimir algo terminará, mientras que un proceso que ejecute la función `timer:sleep(infinity)` durará para siempre[1].

- *Mensajes*—Los mensajes sirven para que los procesos se comuniquen, el operador de envío es `!`. Un mensaje puede ser cualquier tipo de dato Erlang. Los mensajes se envían de forma asíncrona de un proceso a otro y el receptor siempre se queda con una copia del mensaje. Los mensajes se almacenan en el buzón de correo del proceso receptor a la llegada y se pueden recuperar mediante la ejecución de la cláusula `receive`.

[1] La función `sleep(Time)` suspende la ejecución del proceso por `Time` milisegundos, el átomo `infinity` representa un tiempo infinito.

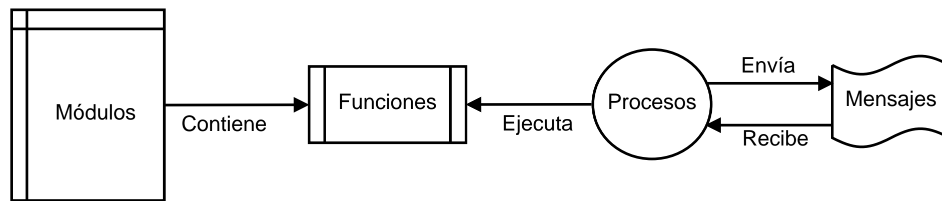

Figura 5.1: Conceptos claves de Erlang y sus relaciones

5.1.2. El cálculo de eventos

El cálculo de eventos es un formalismo lógico basado en la narrativa para el razonamiento acerca de acciones (eventos) y sus efectos (fluentes) respectivos. Una narrativa es una especificación de un conjunto de afirmaciones sobre la ocurrencia de eventos a través del tiempo. La afirmación de la ocurrencia de un evento en el cálculo de eventos se representa con el siguiente predicado:

$$Happens(e, t)$$

que indica que el evento e ocurre en el tiempo t. Ahora, para representar qué efecto tiene un evento se utiliza el predicado:

$$Initiates(e, f, t)$$

donde el fluente f inicia (es verdadero) a partir de que el evento e sucede en el tiempo t. Para expresar la terminación de un fluente (cuando sea falso) se utiliza el predicado siguiente:

$$Terminates(e, f, t)$$

lo cual afirma que la ocurrencia del evento e en el tiempo t finaliza la veracidad del fluente f. Por último, se muestra cómo se revisa en el cálculo de eventos la veracidad de un fluente:

$$HoldsAt(f, t)$$

este predicado es verdadero cuando el fluente f se mantiene en el tiempo t.

En este trabajo se argumenta que al utilizar el concepto principal de *narrativas* del cálculo de eventos se puede conseguir una herramienta adecuada y de alto nivel de abstracción: *la notación ProME*. Esto hará que al vincular las narrativas del cálculo de eventos con el diseño de sistemas con paso de mensajes en Erlang, se obtendrán las directrices para escribir implementaciones Erlang.

5.2. Acercando el cálculo de eventos a Erlang

Para poder llevar el cálculo de eventos a la sintaxis de Erlang se utilizará como ejemplo la arquitectura Cliente/Servidor presentada en la sección 4.6, que es útil para la construcción de sistemas distribuidos. Los puntos a considerar para realizar esto son:

1. hay un único servidor y se llama *Server*,
2. existe un cliente que se llama *Client* pero pueden haber varias instancias de este, y
3. la comunicación es síncrona entre el servidor y el cliente.

Sin embargo, antes de esto es necesario presentar los nombres de los eventos que utilizará la notación ProME del cálculo de eventos. Recordar que en la sección 4.5 se dijo que algunas de las acciones principales de la programación concurrente en Erlang eran las de enviar y recibir mensajes. Estos eventos son:

1. ***actuate***—involucrará la creación de una función Erlang.
2. ***receive***—involucrará un patrón de la cláusula `receive` de Erlang.
   a) ***end***—es un caso especial del evento *receive*, para establecer cuándo un proceso debe terminar. Involucrará la definición de un patrón en la cláusula `receive` que haga que termine el proceso Erlang en cuestión.
3. ***send***—involucrará el envío de un mensaje a un proceso con el operador de envío `!`.

5.2.1. Asignaciones a los eventos

Con asignaciones a los eventos se refiere a que los tres eventos mencionados arriba tendrán que estar relacionados con algunos valores. Por ejemplo, si un mensaje *Msg* se desea enviar a un proceso *Process1* entonces el evento *send* tendrá que conocer estos valores. La asignación quedaría como:

$$\mathit{send}(Process1, Msg) \tag{5.1}$$

otro ejemplo sería un mensaje de un proceso que llega a otro, donde este último lo recibe de acuerdo a los patrones definidos en su cláusula `receive`, la asignación sería:

$$\mathit{receive}(FromProcesss, Pattern1) \tag{5.2}$$

donde ese mensaje puede coincidir con el patrón *Pattern1* y que además contiene el identificador del proceso del que viene, este valor se establece a la variable no ligada *FromProcess*. El caso del evento *actuate* se aborda a continuación dado que su asignación es un tanto distinta a la de estos dos eventos.

5.2.2. El servidor como proceso

En esta sección se describirá paso a paso las expresiones de la notación ProME propuesta del cálculo de eventos y las correspondientes relaciones con las instrucciones Erlang que asisten al funcionamiento del proceso servidor para el ejemplo de la arquitectura Cliente/Servidor. Las funcionalidades serán las de lanzar el servidor, recibir peticiones, responder al cliente y cómo terminar la ejecución del servidor en caso de que se requiera.

5.2.2.1. Iniciar servidor

Para el evento *actuate*, aparte de relacionarse con algún valor se le tendrá que indicar qué es lo que tiene que accionar, valga la redundancia. En el caso del ejemplo *Server*, el nombre del servidor será el sufijo del nombre del evento *actuate* y quedará como:

$$actuateServer(Args) \tag{5.3}$$

donde *Args* serían los argumentos que necesite el servidor para arrancar. Estos argumentos pueden expresarse con la sintaxis de los tipos de datos que emplea Erlang o en lenguaje natural que represente algo que después en el código se pueda sustituir por tipos de datos Erlang. Con esta expresión lo que se tiene es que *actuateServer* es el evento que puede iniciar al proceso *Server* el cual necesita los argumentos *Args*. Empero hasta acá todavía el servidor no inicia, sólo se esta indicando qué servidor se iniciará y cómo se va a iniciar. Para poder iniciar el servidor se expresará con el predicado *Initiates* del cálculo de eventos, por lo que se tendría:

$$\mathit{Initiates}(actuateServer, liveServer, 0) \tag{5.4}$$

lo cual indica que se daría vida al servidor (fluente *liveServer*) con el evento *actuateServer* y es entonces cuando ya se pone en marcha. El tercer argumento se refiere al momento en que debe iniciar el servidor, no pueden haber tiempos negativos ni fraccionarios sólo enteros positivos. El valor cero representa que el servidor estará ejecuntándose desde el inicio del sistema y que a partir de entonces está preparado para poder recibir peticiones y responder. Estas dos expresiones juntas, 5.3 y 5.4, representarían en Erlang las siguientes instrucciones:

```
1  -module(server).
2  -export([actuateServer/0, functionServer/1]).
3
4  actuateServer()-> register(server,spawn(server,functionServer,[Args])).
5
6  functionServer(Args)-> functionServer(Args).
```

Este mapeo a código se lleva a cabo de la siguiente forma:

1. La línea 1 declara el uso de un módulo con nombre *server*. Este nombre se obtuvo del sufijo del evento *actuate*, que se formó como *actuateServer*.

2. La línea 2 declara las funciones `actuateServer` y `functionServer` que serán implementadas dentro del módulo. El nombre de las funciones se forma empleando también el sufijo *Server*. Los argumentos de la función `actuateServer` que lanza inicialmente al servidor, por default es cero, sin argumentos. En cambio los de la función `functionServer` será la cantidad de argumentos que se asignen en la expresión *actuateServer* del cálculo de eventos, para el caso del ejemplo se considerara `Args` como único argumento.

3. La línea 4 declara una función en la que se registra un proceso. El proceso que se registra es `server`, sufijo del evento *actuate*, y este proceso ejecutará la función `functionServer` del módulo `server`. Es válido coincidir el nombre del proceso y el del módulo, de hecho se considera una buena práctica de programación en Erlang [CT09].

4. La línea 6 declara la función `functionServer` tomando como argumentos lo que se le haya asignado al evento *actuateServer*. El predicado *Initiates* hace que la función se invoque a sí misma, esto no es un problema para Erlang ya que soporta que los procesos pueden estar ejecutándose siempre.

En este momento ya se tiene un módulo creado con dos funciones dentro. La función `actuateServer` puede quedarse como está o puede ser refinada en la fase de programación para agregarle instrucciones extras que se requieran. Sin embargo para la función principal del servidor, `functionServer`, falta todavía que permita el envío y recepción de mensajes, lo cual es posible con otras expresiones de la notación ProME que en las siguientes secciones se abordan. Antes, es importante recalcar que las expresiones 5.3 y 5.4 se tienen que utilizar al mismo tiempo para conseguir esas instrucciones Erlang presentadas ya que la primera indica qué proceso se desea levantar y en qué condiciones (los argumentos), mientras la segunda hace que eso se lleve acabo.

5.2.2.2. Recepción en el servidor

Con el módulo `server` ya creado se definirán en este todas las funciones que necesite el servidor, sin embargo las principales ya fueron declaradas: `actuateServer` y `functionSever`. Esta última será ejecutada por el proceso `server` y se le anexará las funcionalidades básicas del servidor como lo son las solicitudes que puede atender y las posibles respuestas para cada una de estas. Para el caso de las solicitudes que llegan, significa que se definirá una cláusula `receive` dentro de la función y se especificarán los patrones que tendrá. Con el evento *receive* de la notación ProME es posible representar esto de Erlang, similar a la expresión 5.2:

$$\textit{receive}(From, Pattern1) \tag{5.5}$$

para que este evento suceda se utiliza el predicado *Happens*:

$$\textit{Happens}(\textit{receive}, 1) \tag{5.6}$$

El segundo parámetro, valor 1, indica que debe realizarse el evento *receive* después del evento(s) que sucede(n) en el tiempo 0. La representación en código Erlang de esto consiste de un patrón tipo tupla con dos elementos, el primero de ellos es una variable no ligada que se encuentra a la espera de un *pid* de algún proceso y el segundo es el patrón mismo que puede ser de cualquier tipo de dato Erlang, dado que es la primera aparición del evento *receive* entonces se crea la constructiva `receive` de Erlang:

```
1 functionServer(Args)->
2     receive
3       {From, Pattern1}-> functionServer(Args)
4     end.
```

Notar que la llamada recursiva a la función persiste en el cuerpo de la cláusula `receive`, esto es porque se empleó el predicado *Initiates* y puesto que en el caso del ejemplo el servidor deberá seguir escuchando. Para agregar otro patrón, la notación en el cálculo de eventos quedaría de la siguiente forma:

$$\textit{receive}(From, Pattern2) \tag{5.7}$$

$$\textit{Happens}(\textit{receive}, 1) \tag{5.8}$$

el valor de 1 persiste en *Happens* porque puede suceder este o el anterior evento *receive* (expresiones 5.5 y 5.6), no los dos al mismo tiempo. Con las expresiones 5.7 y 5.8 se genera una cláusula más en `receive` y se coloca como el segundo patrón, este anexo se presenta en la línea 4 del siguiente código:

```
1 functionServer(Args)->
2     receive
3       {From, Pattern1}-> functionServer(Args);
4       {From, Pattern2}-> functionServer(Args)
5     end.
```

Las separaciones entre cláusulas dentro de `receive` se da por el punto y coma que significa la disyunción de las cláusulas de acuerdo a la concordancia de patrones. Puesto que las expresiones de la notación se están dando en forma narrativa, con 5.5 y 5.6 se crea la cláusula `receive ... end`, y cuando aparecen las expresiones 5.7 y 5.8 ya no es necesario crear otra sino que se agregan a la actual.

Por otro lado la variable `From` que aparece en ambos patrones será el valor del *pid* del cliente, quién solicite algo al servidor. Esto quiere decir que todos los clientes que envíen una petición tendrán que informar al servidor su identificador de proceso para que este pueda o no responderles, según lo amerite.

Por último, los identificadores `Pattern1` y `Pattern2` no son variables sino que representan la estructura del mensaje que recibe el servidor, es decir, la tupla `{From, PatternN}` en realidad puede tener muchos elementos, según sea lo que se requiera y no solamente dos, con el tipo de expresiones 5.5 y 5.7 el primer elemento de la tupla siempre será el identifcador del proceso emisor, mientras el segundo elemento puede ser de cualquier tipo de dato Erlang, el tercer el elemento también, etc. Esto es, que `PatronN` representa desde el segundo elemento de la tupla hasta el último. Quien haga la modelación con la notación ProME propuesta tiene la libertad de expresar la estructura real del patrón o nombrar un identificador con lenguaje natural a lo que necesite que represente para después detallarlo en la programación de la función `functionServer`, en este caso.

5.2.2.3. Responder desde el servidor

Como se mencionaba en la sección anterior, cada petición que llegue al servidor puede ser respondida por este. Entonces, es importante que por cada recepción se describa enseguida la respuesta al cliente; con el evento *send* y el predicado *Happens* es posible hacer esto. Para que el primer evento *receive* tenga su correspondiente evento *send*, después de las expresiones 5.5 y 5.6 deberían ir enseguida las siguientes:

$$send(\{ok, Response\}) \tag{5.9}$$

$$Happens(send, 2) \tag{5.10}$$

lo cual haría que se modificara el cuerpo de la primera cláusula `receive` para agregar una operación de envío hacia el proceso del que se recibió la petición, he ahí la importancia del proceso emisor *From* asignado en la expresión 5.5 del evento *receive* ya que en la expresión 5.9 del evento *send* no es necesario indicarlo porque se considera que está dentro del evento *receive*. La instrucción Erlang correspondiente a las expresiones 5.9 y 5.10 se muestra en la línea 3 del siguiente código:

```
1 functionServer(Args)->
2     receive
3       {From,Pattern1}-> From ! {ok, Response},
4                          functionServer(Args);
5       {From,Pattern2}-> functionServer(Args)
6     end.
```

Como se aprecia, el cuerpo de la primera cláusula es el que se modifica tomando la asignación del evento *send* y enviandolo al cliente a través de la variable `From`. Aquí hay una diferencia respecto a la expresión 5.1 presentada anteriormente como ejemplo y la expresión 5.9, ya que a la primera se le asignan dos valores y a la segunda sólo uno. Esto debido a que el evento *send* puede tomar esa dos formas. Ahora, respecto a las expresiones 5.7 y 5.8 si se desea la respuesta correspondiente entonces deben continuar las siguientes:

$$send(\{error, Reason\}) \tag{5.11}$$

$$Happens(send, 2) \tag{5.12}$$

Con esto se agregaría al código la instrucción `From ! {error, Reason}` en el cuerpo de la segunda cláusula de `receive`, línea 5 del código anterior mostrado. Al igual que en el caso de los patrones del evento *receive*, el mensaje del evento *send* puede ser de cualesquiera tipo de dato Erlang. En este caso siguiendo las respuestas empleadas en la arquitectura Cliente/Servidor se utilizan tuplas donde el primer elemento puede ser el átomo `ok` o el átomo `error`, mientras que el resto queda abierto a cualquier cantidad de elementos que pueden ser de cualquier tipo de dato Erlang.

Hasta aquí se tiene a `functionServer` con las funcionalidades ya establecidas de recepción y envío de mensajes. Sin embargo se puede agregar un evento más que haga que el servidor termine su ejecución, esto se traduce a que se construya una cláusula dentro de `receive` que ya no invoque a la función `functionServer`. Esto se aborda en el siguiente apartado.

5.2.2.4. Terminar el servidor

Pudiera necesitarse que el servidor termine de buena manera haciendo antes una serie de acciones que se requieran sin que el proceso como tal se ha forzado a parar. Para eso se utiliza el evento *end* junto con el predicado *Terminates*, que sería un caso especial del evento *receive* ya que dependerá de una petición que reciba y que encaje con un patrón que además de que ya no invocará a la misma función en este caso `functionServer`, se podrán agregar instrucciones que se necesiten antes de que el servidor se detenga. Esto queda expresado en el cálculo de eventos como:

$$end(Finish) \tag{5.13}$$

$$Terminates(end, liveServer, 1) \tag{5.14}$$

donde *Finish* será el patrón que tenga la sentencia `receive`, entonces cuando ocurre el evento *end* con el predicado *Terminates* hace que el fluente *liveServer* ahora sea falso, ya que el servidor dejará de estar ejecutándose. En código Erlang se agregará un patrón más en la función `functionServer`:

```
1 functionServer(Args)->
2     receive
3       {From,Pattern1}-> From ! {ok, Response},
4                         functionServer(Args);
5       {From,Pattern2}-> From ! {error, Reason},
6                         functionServer(Args);
7       {Finish}-> %to do some before to end the function
8                  ''Process is finished.''
9     end.
```

De esta manera es posible indicar en la notación cómo terminar un proceso Erlang que siempre se está ejecutando. Estas instrucciones que se agregan a la función deberían de ser modificadas después para personalizar el fin del proceso. Incluso es posible continuar con eventos *send* para avisar a otros procesos de que el servidor terminará o enviar alguna información antes de esto.

5.2.2.5. Modelación del servidor

Finalmente, en la Fig. 5.2 se muestra la narrativa completa de los eventos que ocurren del lado del servidor en esta arquitectura con el correspondiente código Erlang que representa.

Modelación del servidor con la notación ProME
$actuateServer(Args)$
$Initiates(actuateServer, liveServer, 0)$
$receive(From, Pattern1)$
$Happens(receive, 1)$
$send(\{ok, Response\})$
$Happens(send, 2)$
$receive(From, Pattern2)$
$Happens(receive, 1)$
$send(\{error, Reason\})$
$Happens(send, 2)$
$end(Finish)$
$Terminates(end, liveServer, 1)$

Código Erlang correspondiente

```
1  -module(server).
2  -export([actuateServer/0, functionServer/1]).
3
4  actuateServer()-> register(server, spawn(server,functionServer,[Args])).
5
6  functionServer(Args)->
7      receive
8        {From,Pattern1}-> From ! {ok, Response},
9                           functionServer(Args);
10       {From,Pattern2}-> From ! {error, Reason},
11                          functionServer(Args);
12       {Finish}-> % to do some before to end the function
13                   ''Process is finished.''
14     end.
```

Figura 5.2: Correspondencia entre la narrativa y Erlang para el servidor

5.2.3. El cliente como proceso

Después de haber modelado la parte del servidor ahora se abordara el proceso cliente que a diferencia del servidor, el tiempo de vida del cliente será breve ya que existirá en este caso sólo para hacer una solicitud al servidor, esperar la respuesta y terminará sin tener que más instrucciones ejecutar.

5.2.3.1. Iniciar el cliente

Por iniciar el cliente en este caso se entenderá como la modelación de la función que hará la petición al servidor, es decir, esta función cliente puede ser utilizada desde la consola Erlang u otro proceso para hacer las peticiones tomando como argumento la solicitud como tal, por lo tanto en este caso no se empleara el predicado *Initiates* sino *Happens* porque este permitirá indicar que la función pueda ser ejecutada por cualquier proceso, a diferencia de *Initiates* que se específica el nuevo proceso. El sufijo para el evento *actuate* será *Client* y las expresiones en la notación ProME quedarían como:

$$actuateClient(Request) \tag{5.15}$$

$$Happens(actuateClient, 1) \tag{5.16}$$

El código que generan las expresiones 5.15 y 5.16 es la declaración de una nueva función en el módulo `server`, se trata de `functionClient` la cual es agregada. Notar que no se crea un nuevo módulo para el cliente, esto es porque se está usando el predicado *Happens* y no *Initiates* que sí tiene ese efecto. La nueva función en el módulo queda de la siguiente manera:

```
1 -module(server).
2 -export([actuateServer/0, functionServer/1, functionClient/1]).
3 
4 actuateServer()-> ...
5 
6 functionServer(Args)-> ...
7 
8 functionClient(Request)-> .
```

Los puntos suspensivos en las dos primeras funciones se emplean para no describir toda la función ya que eso fue presentado en la Fig. 5.2; lo que se modificó en el módulo fue la línea 2 agregando la función `functionClient` y en la línea 8 se declaro la función tomando como argumento lo que se le asignó al evento *actuateClient*.

5.2.3.2. Enviar solicitud al servidor

Para modelar cómo el cliente tiene que enviar una solicitud al servidor dentro de la función `functionClient` se le tiene que indicar hacia qué proceso. Contrario a las expresiones 5.9 y 5.10 donde el servidor podía conocer de antemano a quién responder, aquí el cliente deberá conocer especificamente a quién hacer la petición. Para esto al evento *send* se le puede asignar tal valor además de la solicitud y después emplear el predicado *Happens*:

$$send(Server, Request) \tag{5.17}$$

$$Happens(send, 2) \tag{5.18}$$

Dado que se asume que esta narrativa es consecutiva a la presentada en la Fig. 5.2, la notación reconoce qué eventos han sido declarados y por eso se puede emplear como argumento a *Server* dentro del evento *send*. Es decir, el proceso *Server* de la narrativa pasada esta disponible para interactuar con otros procesos que puedan estar después de este y viceversa. Así, las expresiones 5.17 y 5.18 deben ir enseguida de 5.15 y 5.16 para agregar a la función `functionClient` una operación de envío:

```
1  -module(server).
2  -export([actuateServer/0, functionServer/1, functionClient/1]).
3
4  actuateServer()-> ...
5
6  functionServer(Args)-> ...
7
8  functionClient(Request)-> server ! {self(), Request}.
```

La línea 8 es modificada para agregar la operación de envío, donde el mensaje se forma como tupla teniendo como primer elemento el valor del identificador de proceso del cliente, através de la función `self`, y con el argumento *Request*. Este mensaje debería coincidir con alguno de los patrones modelados en la sección 5.2.2.2 respecto a las recepciones que hace el servidor. El proceso del *Servidor* es visible gracias a que se registró como proceso con el nombre de `server`, sección 5.2.2.1. Ahora el cliente debe estar preparado para recibir la respuesta del servidor, en el siguiente apartado se modela esto.

5.2.3.3. Recibir respuesta del servidor

Empleando el evento *receive* de la notación ProME, como ya se vió antes es posible crear la constructiva `receive` de Erlang, asignando el tipo de respuesta esperada. Para el ejemplo

se utilizarán las dos posibles respuestas que son éxito y fallo. La narrativa queda:

$$receive(\{ok, Response\}) \tag{5.19}$$

$$Happens(receive, 3) \tag{5.20}$$

$$receive(\{error, Reason\}) \tag{5.21}$$

$$Happens(receive, 3) \tag{5.22}$$

En caso de que se busque esperar más de una respuesta de éxito, sólo basta agregar las expresiones correspondientes de la notación indicando cómo debe ser esa respuesta, ya que puede ser de cualquier tipo de dato Erlang o expresarse en lenguaje natural. En la sección 5.2.2.3 se modeló cómo el servidor responde a las peticiones de los clientes y utiliza también la misma estructura de estas peticiones que se están describiendo ahora. Aunque cabe decir que ya queda a cuenta de quién modele hacer coincidir tanto los tipos de respuesta que se envían como las que se reciben. Las expresiones 5.19, 5.20, 5.21 y 5.22 dan paso a la siguiente representación en Erlang:

```
1 functionClient(Request)->
2     server ! {self(), Request},
3     receive
4       {ok, Response}-> ;
5       {error, Reason}->
6     end.
```

Las nuevas líneas son de la 3 a la 6. Si se llega a compilar esto causaría un error de sintaxis porque la cláusulas de las líneas 4 y 5 no tienen cuerpo. Aquí dependerá de lo que se quiera hacer con la respuesta del servidor (éxito o fallo) dentro de la función `functionClient`, ya sea imprimirla, guardarla o hacer cálculos con ella y después enviarla a otro proceso, etc. Suponer que existe otro proceso que también hace uso de la respuesta que da el servidor a este cliente y el encargado de pasarle esa respuesta es este cliente. Este otro proceso se llama *Log* que registra todas las respuestas exitosas del servidor. Para hacer este envío es necesario ubicar las expresiones correspondientes después del evento *receive* exitoso (expresiones 5.19 y 5.20) y antes de 5.21 y 5.22, la modelación sería:

$$send(Log, Response) \tag{5.23}$$

$$Happens(send, 4) \tag{5.24}$$

Lo cual signficaría formar el cuerpo de la primera cláusula de la constructiva `receive`. Es importante mencionar que este evento *send* sucederá en el tiempo 4, justo después de que coincida con el patrón del evento *receive*, lo cual auxilia la modelación de la ocurrencia de los

eventos. En la línea 2 del siguiente código se presenta la correspondencia de las expresiones 5.23 y 5.24:

```
receive
  {ok, Response}-> log ! {self(), Response};
  {error, Reason}->
end.
```

Puesto que el proceso cliente sólo se ejecutará una vez en contraste con el servidor que se mantiene escuchando, no tendrá un evento *end* que lo finalice.

5.2.3.4. Modelación del cliente

En la Fig. 5.3 se presenta la modelación completa del cliente con la notación ProME y el correspondiente código Erlang. Básicamente fue el anexo de `functionClient` para interactuar con el servidor. Notar que las expresiones 5.23 y 5.24 del evento *send* fueron colocadas enseguida del evento *receive* exitoso, por lo tanto si se deseara hacer unos envíos más se puede seguir utillizando el evento *send*, incluso antes del evento *receive* como lo atestiguan las expresiones 5.17 y 5.18.

5.3. ProME, una notación para la modelación de procesos Erlang

Después de tener un primer acercamiento con la notación propuesta del cálculo de eventos ahora se describen sus elementos a detalle y la sintaxis a la que obedecen. Ya se ha mencionado el acrónimo *ProME* para referirse a la notación propuesta en esta tesis, que se lee como modelación de procesos Erlang, en inglés *Process Modeling Erlang*.

5.3.1. Elementos de la notación ProME

Los elementos que componen la notación ProME son los siguientes:

1. **Eventos**. Los eventos para modelar procesos Erlang son cuatro: *actuate* para lanzar un proceso o modelar una función de un proceso, *send* para enviar mensajes a otro proceso, *receive* para recibir mensajes de un proceso y *end* para recibir un mensaje específico que finalice la ejecución de un proceso.

Modelación del cliente con la notación ProME
$actuateClient(Request)$
$Happens(actuateClient, 1)$
$send(Server, Request)$
$Happens(send, 2)$
$receive(\{ok, Response\})$
$Happens(receive, 3)$
$send(Log, Response)$
$Happens(send, 4)$
$receive(\{error, Reason\})$
$Happens(receive, 3)$

Código Erlang correspondiente

```
1  -module(server).
2  -export([actuateServer/0, functionServer/1, functionClient/1]).
3
4  actuateServer()-> ...
5
6  functionServer(Args)-> ...
7
8  functionClient(Request)->
9      server ! {self(), Request},
10     receive
11       {ok, Response}-> log ! {self(), Response};
12       {error, Reason}->
13     end.
```

Figura 5.3: Correspondencia entre la narrativa y Erlang para el cliente

2. **Predicados**. Los predicados que se utilizan en el cálculo de eventos son: *Initiates* para representar el comienzo de la ejecución de un proceso, *Happens* para indicar cuándo ocurre un evento y *Terminates* para finalizar la ejecución del proceso. Estos predicados trabajan con los eventos mencionados.

3. **Fluentes**. El único fluente que se emplea para indicar cuándo un proceso se está ejecutando es *live*. En otras palabras, este fluente es verdadero desde que se lanza un proceso y mientras se mantiene ejecutando, y falso cuando termina su ejecución. Se acompaña siempre del nombre del proceso en cuestión.

4. **Tiempos**. Los tiempos son muy importantes para la modelación porque sirven para saber cuándo suceden los eventos y así construir una narrativa que muestre como se da la interacción entre los procesos. El tiempo es mayor o igual a cero, $t \geq 0$. El valor cero generalmente indicará el arranque del sistema, del proceso o de la aplicación, es decir en ese punto comienzan su existencia para después operar con ellos.

5. **Procesos**. Es un elemento principal dentro de la notación debido a que los eventos trabajan sobre este. Un proceso existe desde el momento que se emplea el evento *actuate* ya que requiere saber sobre a quién accionará. El nombre del proceso tiene que comenzar con una letra mayúscula y puede estar formado por mayúsculas, minúsculas y números. Después este nombre puede ser utilizado en el resto de los eventos y en otras narrativas.

6. **Argumentos**. Con argumentos se refiere a aquellos parámetros que se pueden expresar en algún tipo de dato Erlang o en lenguaje natural. Por ejemplo el evento *send* cuando se le indica qué mensaje enviar, este mensaje se puede indicar con los tipos de datos Erlang o nombrar el mensaje por lenguaje natural para que en la fase de implementación el programador lo represente en el tipo de dato que aplique. La notación utiliza este tipo de argumentos en los eventos *actuate* para los posibles parámetros que puede tener la función, *send* para el mensaje y *receive* para el patrón.

7. **Narrativas**. Una narrativa en la notación ProME es una secuencia de eventos (los del punto 1) que ocurren. Siempre parte de un evento *actuate* y al volver a encontrar otra vez a este evento se considera como una segunda narrativa y así sucesivamente. Las narrativas se relacionan cuando una emplea el nombre de un proceso que es sufijo del evento *actuate* en otra narrativa. Cada narrativa puede representar en Erlang o un lanzamiento de un proceso Erlang y su correspondiente declaración de función o sólo la declaración de una función que interactuará con procesos. Así, una narrativa puede estar compuesta por un evento *actuate* y al menos un evento *send* o un evento *receive*.

5.3.2. Sintaxis de los elementos de la notación ProME

Para describir la sintaxis de cada elemento se comenzará desde los simples hasta los compuestos.

5.3.2.1. Palabras reservadas

Las palabras reservadas de la notación ProME son: *actuate*, *send*, *receive*, *end*, *live*, *Initiates*, *Happens* y *Terminates*.

5.3.2.2. Procesos

El nombre de los procesos puede formarse con letras mayúsculas, minúsculas y números, pero el primer caracter tiene que ser una letra mayúscula. Ejemplos: *ProcessDataBase*, *LogDay*, *ServerA7*, *Client101*, *P1*.

5.3.2.3. Argumentos

Como ya se dijo, los parámetros que reciben los eventos *actuate*, *send* y *receive* pueden expresarse en los tipos de datos que maneja Erlang o en lenguaje natural. Aquí algunos ejemplos de argumentos de función, argumentos como mensajes y argumentos como patrones respectivamente a esos tres eventos:

$$\begin{array}{l} actuateServer([true, -1, [\,]]) \\ actuateServer(ArgsInitials) \\ \hline send(Client, \{ok, false, Result\}) \\ send(Client, OkResult) \\ \hline receive(\{exit, Operation\}) \\ receive(OperationExit) \end{array}$$

5.3.2.4. Evento *actuate*

El evento *actuate* debe tener como sufijo el nombre del proceso y se le pueden asignar al evento argumentos o no. El evento aplica con uno de dos posibles predicados: *Initiates* o *Happens*. La sintaxis es la siguiente:

$actuateProcessName([Argument])$
$\langle Initiates(actuateProcessName, liveProcessName, Time) \;|\; Happens(actuate, Time)\rangle$

Cuando el evento es seguido por el predicado *Initiates* significa que se lanzará un nuevo proceso y se modelará la función que ejecutará ese proceso. A diferencia si es el predicado *Happens* ya que sólo se estaría modelando una función que llevará como nombre el sufijo del evento *actuate*. Ejemplos:

$$actuateAgent1()$$
$$Initiates(actuateAgent1, liveAgent1, 0)$$

Esto representa únicamente el siguiente código:

```
1 -module(agent1).
2 -export([actuateAgent1/0, functionAgent1/0]).
3
4 actuateAgent1()-> register(agent1, spawn(agent1,functionAgent1,[])).
5
6 functionAgent1()-> functionAgent1().
```

Mientras que con el predicado *Happens* pasa lo siguiente:

$$actuateAgent1()$$
$$Happens(actuate, 0)$$

En caso de que estas expresiones sean la primeras en toda la narrativa sólo se conseguiría:

```
1 functionAgent1()-> functionAgent1().
```

Pero si antes existe un evento *actuate* que aplicó con el predicado *Initiates* entonces también se modificaría la parte de exportación de funciones del módulo correspondiente:

```
1 -module(x).
2 -export([..., functionAgent1/0])
3
4 functionAgent1()-> functionAgent1().
```

En resumen, con el predicado *Initiates* se crea un nuevo módulo en el que se agregan funciones para ese proceso. Entretanto con el predicado *Happens* sólo se estaría modelando una función. Se desea que por cada proceso a modelar se emplee el predicado *Initiates* para la correspondiente creación del módulo Erlang. Enseguida se presentan narrativas que ejemplifican esto:

$$actuateProcessX()$$
$$Initiates(actuateProcessX, liveProcessX, 0)$$
$$actuateProcessY()$$
$$Happens(actuateProcessY, 1)$$
$$actuateProcessZ()$$
$$Happens(actuateProcessZ, 1)$$

Esta narrativa representa el siguiente código:

```
1 -module(processx).
2 -export([actuateProcessX/0, functionProcessX/0,
3          functionProcessY/0, functionProcessZ/0])
4
5 actuateProcessX()-> register(processx, spawn(processx,functionProcessX,[])).
6
7 functionProcessX()-> functionProcessX().
8
9 functionProcessY()-> .
10
11 functionProcessZ()-> .
```

Sin embargo esta otra narrativa genera un código distinto ya que se crearían dos módulos:

$$
\begin{array}{l}
actuateProcessX() \\
Initiates(actuateProcessX, liveProcessX, 0) \\
\\
actuateProcessY() \\
Initiates(actuateProcessY, liveProcessY, 0) \\
actuateProcessZ() \\
Happens(actuateProcessZ, 1)
\end{array}
$$

el primer módulo es:

```
1 -module(processx).
2 -export([actuateProcessX/0, functionProcessX/0])
3
4 actuateProcessX()-> register(processx, spawn(processx,functionProcessX,[])).
5
6 functionProcessX()-> functionProcessX().
```

y el segundo módulo:

```
1 -module(processy).
2 -export([actuateProcessY/0, functionProcessY/0 , functionProcessZ/0])
3
4 actuateProcessY()-> register(processy, spawn(processy,functionProcessY,[])).
5
6 functionProcessY()-> functionProcessY().
7
8 functionProcessZ()-> .
```

5.3.2.5. Evento *send*

El evento *send* se aplicará siempre con el predicado *Happens*. Necesariamente se le tiene que asignar un mensaje y es opcional especificar a quién será enviado. La sintaxis es:

$$send([ProcessName,]Message)$$
$$Happens(send, Time)$$

Cuando solamente se le asigna el mensaje significa que el evento *send* se esta utilizando "dentro" del evento *receive*, esto quiere decir que con el evento *receive* se conoce a quién enviar el mensaje ya que en el patrón se debió haber estructurado para recibir el identificador del proceso remitente, por lo que se puede asumir que hay un proceso que envió el mensaje y que se conoce su *pid*. Este caso se dio en la sección 5.2.2.3 donde se modela cómo responde el servidor ante la llegada de mensajes. Se dice que las expresiones *send* 5.9 y 5.10 están dentro o pertenecen a las expresiones *receive* 5.5 y 5.6. Un ejemplo que muestra las dos versiones de uso del evento *send* en una misma narrativa se muestra enseguida:

$$actuateRobot1()$$
$$Happens(actuateRobot1, 1)$$
$$send(Robot2, Msg1)$$
$$Happens(send, 2)$$
$$receive(PidRobot2, ok)$$
$$Happens(receive, 3)$$
$$send(Msg2)$$
$$Happens(send, 4)$$

que reprenta el siguiente código Erlang:

```
1 functionRobot1()->
2     robot2 ! Msg1,
3     receive
4       {PidRobot2, ok}-> PidRobot2 ! Msg2
5     end.
```

El primer evento *send* se le asigna el nombre del proceso al cual hay que enviar el mensaje, mientras el segundo evento *send* toma el nombre del proceso del evento *receive* que lo precede, el proceso *PidRobot2*. Notar que hay cuatro tiempos distintos y que ninguno de los eventos se pueden empalmar, el primero es cuando se invoca a la función, el segundo cuando se envía el primer mensaje, el tercero cuando llega un mensaje de `robot2` que coincide con el patrón de la cláusula `receive` y el cuarto tiempo es cuando se envía otro mensaje a `robot2`.

5.3.2.6. Evento *receive*

El evento *receive* también tiene como opcional asignarle dos argumentos a la vez. El argumento obligatorio es el del patrón y el opcional es el identificador del proceso emisor. Su sintaxis consiste en:

$$receive([ProcessName,]Pattern)$$
$$Happens(receive, Time)$$

El primer caso es cuando sólo se le asigna un patrón y este se pasa tal cual en la cláusula `receive`. El segundo caso es cuando se anexa como primer argumento un identificador del proceso emisor y es entonces que con el patrón se forma una tupla en `receive` teniendo como primer elemento al proceso y en el resto de los elementos al patrón. Un ejemplo del primer caso se da a continuación:

$$receive([\{true, a, 2\}, timeLogin, VarX])$$
$$Happens(receive, 1)$$
$$receive(MsgLogin)$$
$$Happens(receive, 1)$$

que representaría lo siguiente:

```
1    receive
2       [{true,a,2},timeLogin,VarX]-> ;
3       MsgLogin->
4    end.
```

El ejemplo correspondiente para el caso donde se asigna, además del patrón, el nombre de un proceso es:

$$receive(WebProcess, [10, 20, 30])$$
$$Happens(receive, 1)$$
$$receive(SystemProcess, \{x, y, z\})$$
$$Happens(receive, 1)$$

lo cual significa en Erlang esto:

```
1    receive
2       {WebProcess,[10,20,30]}-> ;
3       {SystemProcess,{x,y,z}}->
4    end.
```

Cabe recordar que en la sección 5.2.2.2 sobre la recepción que hace el servidor de las solicitudes y en 5.2.3.3 sobre cómo recibe el cliente la respuesta del servidor se emplearon estos dos casos del evento *receive* respectivamente.

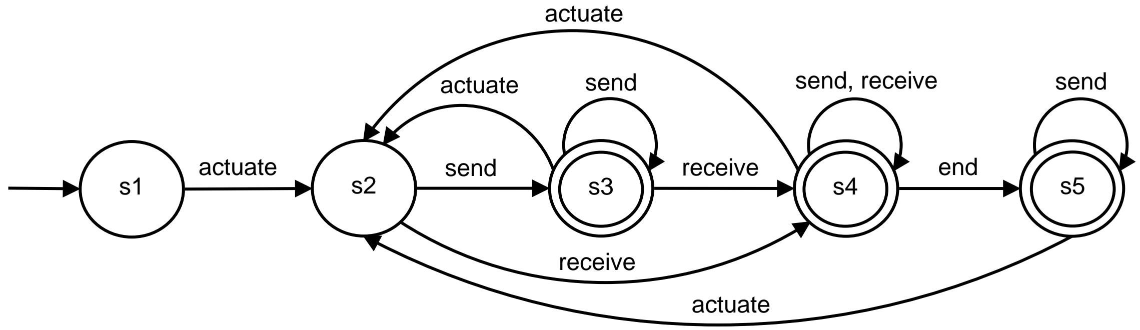

Figura 5.4: Autómata finito determinista de las narrativas de la notación ProME

5.3.2.7. Narrativa

Como ya se mencionaba anteriormente, es una secuencia de los eventos *actuate*, *send*, *receive*y *end*. Se asume que los predicados *Initiates*, *Happens* y *Terminates* acompañan a estos eventos según las estructuras descritas en los puntos previos. Una narrativa comienza con un evento *actuate* y le siguen a este al menos un evento *send* o un evento *receive*.

Haciendo uso de los autómatas finitos deterministas (AFD) que provee la teoría matemática de la computación [Bro89], se define enseguida el autómata que reconoce las estructuras sintácticas válidas de las narrativas de la notación ProME. Sea la quíntupla:

$$Narrative = (S, \Sigma, \delta, s, F)$$

donde S es el conjunto de estados, Σ el alfabeto de la máquina, δ la función de transición de $S \times \Sigma \to S$, $s \in S$ es el estado inicial y $F \subseteq S$ es el conjunto de estados de aceptación. Los valores de cada uno de ellos son los siguientes:

$$S = \{s_1, s_2, s_3, s_4, s_5\}$$
$$\Sigma = \{actuate, send, receive, end\}$$
$$\delta(s_1, actuate) = s_2 \quad \delta(s_2, send) = s_3 \quad \delta(s_2, receive) = s_4 \quad \delta(s_3, send) = s_3$$
$$\delta(s_3, receive) = s_4 \quad \delta(s_3, actuate) = s_2 \quad \delta(s_4, end) = s_5 \quad \delta(s_4, send) = s_4$$
$$\delta(s_4, receive) = s_4 \quad \delta(s_4, actuate) = s_2 \quad \delta(s_5, send) = s_5 \quad \delta(s_5, actuate) = s_2$$
$$s = s_1$$
$$F = \{s_3, s_4, s_5\}$$

La representación gráfica de la función de transición δ del autómata definidio para las narrativas de la notación ProME se muestra en la Fig. 5.4, la cual describe los tipos de estructuras sintácticas que puede tener (aceptar) una narrativa.

% reglas de producción principales

Narrative→ Actuate ⟨ Initiates | Happens ⟩ EventA [Narrative]

Actuate→ **actuate**ProcessName **(** [Argument] **)**

Initiates→ **Initiates (actuate**ProcessName **, live**ProcessName **,** Time **)**

Happens→ **Happens (** ⟨ **actuate** | **send** | **receive** ⟩ **,** Time **)**

EventA→ Send [EventA] | Receive [EventB]

EventB→ Send [EventB]
| Receive [EventB]
| End [EventC]

EventC→ Send [EventC]

Send→ **send (** [ProcessName **,**] Message **)** Happens

Receive→ **receive (** [ProcessName **,**] Pattern **)** Happens

End→ **end (** Pattern **)**
Terminates (end , live ProcessName **,** Time **)**

% reglas de producción auxiliares

ProcessName→ Uppercase Rest
Rest→ ⟨Uppercase | Lowercase | Digit⟩ [Rest]
Pattern→ Argument
Message→ Argument
Argument→ DataTypesErlang | NaturalLanguage
Time→ Digit [Time]
Uppercase→ ⟨A-Z⟩
Lowercase→ ⟨a-z⟩
Digit→ ⟨0-9⟩

Figura 5.5: Gramática de la notación ProME en la forma Backus Naur extendida

5.3.2.8. Gramática de la notación ProME

Por último, respecto a la sintaxis, se presenta en la Fig. 5.5 la gramática regular [ALSU07] de la notación ProME que establece todas las estructuras que puede tener la notación. La gramática se muestra en la forma Backus-Naur extendida [Lou97].

5.4. Metodología para emplear la notación ProME

Enseguida se describen las pautas para utilizar la notación ProME. Primero, un programa Erlang construido a partir de una narrativa es un modelo lógico de los axiomas y la narrativa del cálculo. Por eso se debe considerar siempre la consistencia en la narrativa con el envío y recepción de mensajes. Deberá existir una narrativa para cada componente del sistema distribuido y/o concurrente. Cada narrativa describirá los eventos que, o bien son concurrentes o funcionan como petición-respuesta como el caso de la arquitectura Cliente/Servidor. Con esto, de cada narrativa será posible extraer un esqueleto de programación Erlang mediante la inspección de qué mensajes son enviados o recibidos, además de tener la posibilidad de modelar un módulo y sus funciones, o solamente una función.

Siguiendo este marco conceptual, un mayor detalle en la modelación de las narrativas daría un mayor detalle a los programas Erlang y también a partir de estas por sí mismas se pueden sacar algunas conclusiones teóricas que los sistemas distribuidos deben satisfacer. Para alcanzar este objetivo, una caracterización adecuada de los eventos, de los predicados y la dependencia del tiempo es absolutamente obligatorio.

El método propuesto para programar un sistema distribuido basado en la notación ProME y conseguir la parte medular de su implementación en Erlang es el siguiente:

1. Identificar a los actores en el sistema y sus roles, que serán los futuros procesos.

2. Identificar qué eventos llevan a cabo los actores y cuáles ocurren de forma paralela, concurrente o acoplada.

3. Identificar qué actores interactúan y a través de qué eventos lo hacen.

4. Para cada actor ofrecer las narrativas correspondientes siguiendo un orden temporal intencionalmente, teniendo en cuenta la sintaxis de la notación ProME.

5. Verificar a nivel narrativa que cada una de ellas y el rol del los actores sean congruentes.

Con esto se tendría la implementación en código Erlang del sistema distribuido descrito en un modelo basado en una serie de narrativas.

5.5. Ejemplo de narrativas ProME

Ahora se aborda la creación de una narrativa del ejemplo que se presentó en la sección 4.6 del capítulo 4 sobre un pequeño servidor que atiende peticiones del cálculo del área de figuras geométricas. Los procesos clientes pueden ser muchos y habrá un único proceso servidor. Las funcionalidades del servidor son: cálcular el área de dos figuras y devolver el resultado, en caso de que llegué una petición para calcular el área de una figura que no exista devolverá un mensaje de error. Así, la narrativa correspondiente para el servidor es:

$$actuateGeometria()$$
$$Initiates(actuateGeometria, liveGeometria, 0)$$

$$receive(PidCliente, \{rectangulo, L1, L2\})$$
$$Happens(receive, 1)$$
$$send(ok, L1 * L2)$$
$$Happens(send, 2)$$

$$receive(PidCliente, \{circulo, R\})$$
$$Happens(receive, 1)$$
$$send(ok, pi * R * R)$$
$$Happens(send, 2)$$

$$receive(PidCliente, Otro)$$
$$Happens(receive, 1)$$
$$send(error)$$
$$Happens(send, 2)$$

Por otro lado los clientes harán las solicitudes al servidor con la siguiente narrativa:

$$actuateCliente(Solicitud)$$
$$Happens(Cliente, liveCliente, 1)$$
$$send(Geometria, Solicitud)$$
$$Happens(send, 2)$$
$$receive(\{ok, Resultado\})$$
$$Happens(receive, 3)$$
$$receive(error)$$
$$Happens(receive, 3)$$

Estas simples narrativas ya asumen todo el trabajo de comunicación de paso de mensajes entre los procesos. Significa, que el servidor una vez en ejecución entra en escucha de peticiones y una vez atendidas vuelve a estar a la espera de más peticiones. Recordar que los procesos tiene un buzón de mensajes, así aunque la carga de peticiones al servidor sea mucha siempre serán atendidas cuando llegue su turno, puesto que se toman en orden de llegada. Aprovechando la relación que este modelado tiene con Erlang, el código para la narrativa del servidor es:

```
1  -module(geometria).
2  -export([actuateGeometria/0, functionGeometria/0]).
3
4  actuateGeometria()-> register(geometria, spawn(geometria,functionGeometria,[])
       ).
5
6  functionGeometria()->
7     receive
8       {PidCliente, {rectangulo, L1, L2}} ->
9           PidCliente ! {ok, L1*L2},
10          functionGeometria();
11      {PidCliente, {circulo, R}} ->
12          PidCliente ! {ok, math:pi()*R*R},
13          functionGeometria();
14      {PidCliente, Otro}->
15          PidCliente ! error,
16          functionGeometria()
17    end.
```

La única línea que se modificó en la fase de implementación es la número 12, al agregar el número π con la instrucción `math:pi()`, lo demás fue resultado de la correspondencia. Asimismo, el código que generaría la narrativa para el proceso cliente es:

```
1  -module(geometria).
2  -export([actuateGeometria/0, functionGeometria/0, functionCliente/1]).
3
4  ...
5
6  functionCliente(Solicitud)->
7     geometria ! {self(), Solicitud},
8     receive
9       {ok, Resultado}-> ;
10      error->
11    end.
```

A las líneas 9 y 10 de este último código se les agrega una instrucción a cada una en el consecuente de la cláusula para imprimir el resultado que devuelve el servidor, serían las únicas líneas modificadas en la implementación:

```
1     receive
2       {ok, Resultado}-> io:format("Resultado=~w~n",[Resultado]);
3       error-> io:format("Error~n")
4     end.
```

Para poder simular la ejecución de la función cliente en tres procesos distintos se agrega una función en las que se harán las solicitudes a través de la creación de los procesos con la función `spawn/3`, cada proceso envía una petición distinta:

```
1  -module(geometria).
2  -export([actuateGeometria/0, functionGeometria/0,
3           functionCliente/1, lanzarProcesos/0]).
4
5  ...
6
7  lanzarProcesos() ->
8     spawn(geometria,functionCliente,[{cuadrado,10}]),
9     spawn(geometria,functionCliente,[{circulo, 5}]),
10    spawn(geometria,functionCliente,[{rectangulo, 3, 4}])
11 end.
```

Aunque a simple vista el código parecería ser de menos contenido que las narrativas originales, ese no es el fin de la especificación sino al contrario detallar lo más posible el sistema para evitar ambigüedades y si dicha especificación facilita la tarea de los programadores es una ventaja (*plus*). Recordar que los requisitos son documentados através de notaciones que sin llegar a la programación del sistema van describiendo lo que necesita y su comportamiento.

Por último se muestra la ejecución del sistema, comenzando por compilar el módulo, después levantar el servidor con la función `actuateGeometria()` y enseguida lanzar a los tres clientes procesos que harán las solicitudes al servidor, esto con la función `lanzarProcesos()`:

```
1> c(geometria).
{ok,geometria}
2> geometria:actuateGeometria().
true
3> geometria:lanzarProcesos().
Error
Resultado=78.53981633974483
Resultado=12
```

5.6. Caso de estudio: un sistema enseñanza-aprendizaje en línea

Se trata de un sistema enseñanza-aprendizaje en línea que ha sido reportado en los siguientes trabajos: [CGH$^+$09, CGH$^+$10], donde se describe cómo se emplea Erlang para realizar este sistema. Es posible emplear Erlang en la parte concurrente de un sistema distribuido que da soporte a las tareas de enseñanza-aprendizaje en línea, en particular la gestión del sistema y algunas tareas concurrentes. Entonces también es posible modelarlo con la notación ProME.

5.6.1. Descripción del problema

El problema consiste en implementar un sistema distribuido para apoyar los procesos concurrentes de enseñanza y aprendizaje. Con esta propuesta se pretende apoyar el ambiente del aula tradicional a través de una herramienta de enseñanza-aprendizaje asistida por computadora. Los usuarios, profesores y estudiantes, involucrados interactúan con el sistema usando computadoras y dispositivos móviles facilitando en gran medida sus tareas cotidianas. El sistema posee un recurso compartido: los examenes de evaluación para los estudiantes.

Este sistema está basado ampliamente en la arquitectura Cliente/Servidor (ver secciones 3.3.1 y 4.6). Consta de un sólo servidor y varios clientes. La comunicación entre los clientes y el servidor es asíncrona. Puede existir una comunicación entre los clientes pero debe ser a través del servidor. El servidor se activa en el tiempo t_0 y es a partir de entonces que los clientes pueden abrir un canal de comunicación con el servidor, ya que este decidirá aceptar o rechazar las solicitudes formuladas por los clientes.

Cabe mencionar que esta parte del sistema distribuido es la del *back-end*, ya que procesará las entradas del sistema y devolverá las salidas correspondientes sin preocuparse cómo fueron captadas las entradas ni cómo serán presentadas las salidas. Dado que Erlang posee *sockets* para la interacción con lenguajes de programación como Java o Ruby, el *front-end* puede ser implementado sobre aplicaciones móviles en sistemas como Android por ejemplo o en plataformas web para ser usada en un navegador de escritorio, y así interactuar con los usuarios. Incluso el mismo Erlang permite la creación de interfaces gráficas de usuario a través de la librería *wxErlang* [CT09, Eri14] implementada de la super librería *wxWidgets*[2].

[2] wxWidgets es una librería de C++ que permite a desarrolladores crear GUIs para Windows, Mac OS X, Linux y otras plataformas con una sola base de código. Cuenta con implementaciones en lenguajes como Python, Perl, Erlang, Ruby y muchos otros, da a las aplicaciones un rostro nativo del SO, http://www.wxwidgets.org/.

5.6.2. Tipos de usuarios

Se identifican dos tipos de usuarios en el sistema y que además tendrán el rol de clientes en la mencionada arquitectura: profesor y estudiante. Las funciones o actividades de cada uno de ellos que definen su comportamiento pueden ser descritas por un conjunto de acciones llevadas a cabo através de eventos. Estos eventos se dan cuando existe interacción con otros elementos del sistema. Enseguida se enumeran las actividades de estos usuarios donde las palabras que aparecen en inglés se utilizarán más adelante para la modelación.

Tanto los estudiantes y los profesores tienen algunas actividades en común, las cuales son:

1. Suscripción– **subscribe**: registrarse al sistema, esta petición se envía al servidor después de llenar un formulario, la respuesta del servidor puede ser aceptar (**accept**) o rechazar (**reject**).

2. Iniciar sesión– **logon**: inicio de sesión en el sistema, la respuesta del servidor puede ser: **accept** o **not_accepted**.

3. Invitación de conversación– **invitation(friend)**: es una petición de comunicación con un estudiante o profesor llamado **friend**. La respuesta puede ser aceptada (**accept**) o rechazada (**reject**);

4. Conversación en línea– **chat(friend,msg)**: poder platicar con un estudiante o profesor llamado **friend** si este no está ocupado, por ejemplo, si el estudiante no está haciendo un examen o el profesor diseñando un examen.

5. Mostrar usuario activos– **who**: revelar quiénes están dentro del sistema.

6. Terminar sesión– **logoff**: salir del sistema y así terminar la sesión.

7. Ver recursos– **resources**: poder ver los recursos disponibles como notas, libros, etc.

8. Historial de recursos– **history_resources**: permite conocer al estudiante y profesor el historial de los recursos obtenidos.

9. Actualizar recursos– **update_resources**: permite el acceso a un repositorio local para obtener los últimos recursos.

Los estudiantes tienen las siguientes actividades disponibles:

1. Solicitar un examen– **test**: la respuesta del servidor es: **Type of test?**, y entonces el estudiante puede seleccionar un tipo de examen de entre dos: simulados o reales; después, el examen del tipo seleccionado es mandado al estudiante desde el servidor.

2. Hacer examen– **do_test**: hacer un examen en el intervalo de tiempo $[t_1, t_2]$, $t_1 < t_2$; el tiempo puede ser monitoreado desde el servidor o directamente desde una computadora local.

3. Crear grupo de trabajo– **cgroup(name)**: los estudiantes pueden decidir unirse a este grupo; el líder del grupo es la persona que creó tal grupo con nombre **name**.

4. Mostrar grupos de trabajo– **group**: revelar los grupos existentes en el sistema.

Los profesores por su parte tienen estas actividades exclusivas:

1. Diseñar examen– **design_test**: un profesor puede diseñar un examen, este examen puede ser un recurso compartido.

2. Publicar examen de un profesor– **public(test)**: un profesor puede publicar un examen.

Por otro lado, el servidor administra cada petición y coordina la ejecución de todo el sistema. El servidor se llama *Minerva*. Otras tareas del servidor deben bloquear la comunicación con el mundo exterior, archivos log (es un archivo especial oculto) usados por los estudiantes y limitar el tiempo para resolver un examen. La idea es utilizar la computadora como un asistente entrenando para estudiantes y profesores.

De las actividades de los clientes descritas se pueden derivar las del servidor, ya que ahí se mencionan implícitamente:

1. Regitrar usuario– **subscribe**.

2. Permitir la sesión de un usuario– **logon**.

3. Enviar invitación de conversación a otro usuario– **invitation(friend)**.

4. Enviar mensaje de conversación a otro usuario– **chat(friend,msg)**.

5. Enviar lista de usuarios activos– **who**.

6. Terminar la sesión de usuarios– **logoff**.

7. Enviar lista de recursos disponibles– **resources**.

8. Mostrar historial de recursos del usuario– **history_resources**.

9. Mostrar los últimos recursos disponibles– **update_resources**.

10. Atender solicitud de examen de un estudiante– **test**.

11. Monitorar la realización de examen de un estudiante– **do_test**.

12. Crear grupo de trabajo– **cgroup(name)**.

13. Mostrar grupos de trabajo– **group**.

14. Atender solicitud de un profesor para el diseño de un examen – **design_test**.

15. Publicar un examen– **public(test)**.

5.6.3. Modelación con la notación ProME

Ahora para caracterizar este flujo de acciones se utiliza la notación ProME y con ello obtener la implementación correspondiente a Erlang, esto será para ciertas actividades del usuario estudiante y las correspondientes para el caso del servidor como muestra representativa del uso de la notación. El usuario profesor no se modelará. De la metodología presentada en la sección 5.4 del capítulo anterior, hasta este momento se puede considerar que los primeros tres puntos ya han sido cubiertos y toca ahora crear las narrativas.

5.6.3.1. Proceso cliente estudiante

De las trece actividades que tiene el usuario estudiante se descartarán siete para la modelación, son las siguientes: suscripción, invitación de conversación, ver recursos, historial de recursos, actualizar recursos, crear grupo de trabajo y mostrar grupos de trabajo. Las seis actividades a modelar serán: iniciar sesión, conversación en línea, mostrar usuarios activos, terminar sesión, solicitar examen y hacer examen.

El proceso cliente estudiante es un rol del que se harán todas aquellas instancias que se lleguen a necesitar de los estudiantes, es decir, podrán haber varios *usuarios estudiantes* con el mismo comportamiento pero sólo se modelará un *cliente estudiante*. Cuando un usuario estudiante entre al sistema será un nuevo proceso en ese momento, instancia de cliente estudiante. Así que lo que interesa es cómo será el cliente estudiante, mientras que la generación de procesos usuarios estudiantes puede ser definida en la etapa de implementación.

Enseguida se muestra la primera narrativa para el cliente estudiante donde se declara el inicio de sesión:

$$actuateStudentClient(StudentCredential) \tag{5.25}$$
$$Happens(actuateStudentClient, liveStudentClient, 1) \tag{5.26}$$
$$send(Minerva, \{self(), logon, StudentCredential\}) \tag{5.27}$$
$$Happens(send, 2) \tag{5.28}$$
$$receive(logon_accept) \tag{5.29}$$
$$Happens(receive, 3) \tag{5.30}$$
$$end(logon_not_accept) \tag{5.31}$$
$$Terminates(end, 3) \tag{5.32}$$

La narrativa indica que el cliente debe iniciar en el tiempo 1 y recibe como argumento las credenciales del usuario estudiante (nombre de usuario, contraseña, etc.), envía esto al sevidor *Minerva* y enseguida espera por una respuesta de dos posibles. Si hubo un error al iniciar sesión recibe del servidor el átomo **logon_not_accept**, significa que el proceso termina. En caso de que el usuario haya iniciado sesión exitosamente entonces recibirá como respuesta del servidor el átomo **logon_accept** y ahora el usuario estudiante ya puede realizar sus actividades, esto se modela en la siguiente narrativa:

$$actuateStudentClient() \tag{5.33}$$
$$Happens(actuateStudentClient, liveStudentClient, 4) \tag{5.34}$$
$$receive(test_request) \tag{5.35}$$
$$Happens(receive, 5) \tag{5.36}$$
$$send(Minerva, \{self(), test\}) \tag{5.37}$$
$$Happens(send, 6) \tag{5.38}$$
$$receive(do_test_request) \tag{5.39}$$
$$Happens(receive, 5) \tag{5.40}$$
$$send(Minerva, \{self(), do_test\}) \tag{5.41}$$
$$Happens(send, 6) \tag{5.42}$$
$$receive(\{chat_to, Friend, Msg\}) \tag{5.43}$$
$$Happens(receive, 5) \tag{5.44}$$
$$send(Minerva, \{self(), chat_to, Friend, Msg\}) \tag{5.45}$$
$$Happens(send, 6) \tag{5.46}$$

$$receive(who_request) \tag{5.47}$$
$$Happens(receive, 5) \tag{5.48}$$
$$\quad send(Minerva, \{self(), who\}) \tag{5.49}$$
$$\quad Happens(send, 6) \tag{5.50}$$

$$receive(\{test, TypeTest\}) \tag{5.51}$$
$$Happens(receive, 5) \tag{5.52}$$
$$receive(\{do_test, Test\}) \tag{5.53}$$
$$Happens(receive, 5) \tag{5.54}$$
$$receive(\{chat_from, Friend, Msg\}) \tag{5.55}$$
$$Happens(receive, 5) \tag{5.56}$$
$$receive(\{who, UserList\}) \tag{5.57}$$
$$Happens(receive, 5) \tag{5.58}$$

$$end(logoff_request) \tag{5.59}$$
$$Terminates(end, 5) \tag{5.60}$$
$$\quad send(Minerva, \{self(), logoff\}) \tag{5.61}$$
$$\quad Happens(send, 6) \tag{5.62}$$

Esta narrativa puede verse como una interfaz (contacto) entre los usuarios estudiantes y el servidor, ya que los usuarios estudiantes interactuarán con el cliente estudiante y este con el servidor *Minerva*. La narrativa comienza en el tiempo 4 debido a que es lo que continuaría despues del evento *receive* (expresiones 5.29 y 5.30) de la narrativa anterior. Después se pone a la escucha de peticiones de los usuarios estudiantes y de las respuestas del servidor, de las expresiones 5.35 – 5.50 representan las peticiones de los usuarios estudiantes y enconsecuencia se lo delega al servidor, y de las expresiones 5.51 – 5.58 son las respuestas que vendrán del servidor.

5.6.3.2. Proceso servidor Minerva

Las actividades que se modelarán del servidor *Minerva* son: permitir la sesión de un usuario, enviar mensaje de conversación a otro usuario, enviar lista de usuarios activos, terminar la sesión de usuarios, atender solicitud de examen de un estudiante y monitorar la

realización de examen de un estudiante. La narrativa es la siguiente:

$$actuateMinerva() \tag{5.63}$$

$$Initiates(actuateMinerva, liveMinerva, 0) \tag{5.64}$$

$$receive(FromStudent, \{logon, StudentCredential\}) \tag{5.65}$$

$$Happens(receive, 1) \tag{5.66}$$

$$send(logon_accept) \tag{5.67}$$

$$Happens(send, 2) \tag{5.68}$$

$$send(logon_not_accept) \tag{5.69}$$

$$Happens(send, 2) \tag{5.70}$$

$$receive(FromStudent, test) \tag{5.71}$$

$$Happens(receive, 1) \tag{5.72}$$

$$send(\{test, TypeTest\}) \tag{5.73}$$

$$Happens(send, 2) \tag{5.74}$$

$$receive(FromStudent, do_test) \tag{5.75}$$

$$Happens(receive, 1) \tag{5.76}$$

$$send(\{do_test, Test\}) \tag{5.77}$$

$$Happens(send, 2) \tag{5.78}$$

$$receive(FromStudent, \{chat_to, Friend, Msg\}) \tag{5.79}$$

$$Happens(receive, 1) \tag{5.80}$$

$$send(Friend, \{chat_from, FromStudent, Msg\}) \tag{5.81}$$

$$Happens(send, 2) \tag{5.82}$$

$$receive(FromStudent, who) \tag{5.83}$$

$$Happens(receive, 1) \tag{5.84}$$

$$send(\{who, UserList\}) \tag{5.85}$$

$$Happens(send, 2) \tag{5.86}$$

$$receive(FromStudent, logoff) \tag{5.87}$$

$$Happens(receive, 1) \tag{5.88}$$

El servidor *Minerva* se inicia en el tiempo 0 e inmediatamente entra a la escucha de peticiones, estas son enviadas desde las narrativas anteriores en las expresiones: 5.27, 5.28, 5.37, 5.38, 5.41, 5.42, 5.45, 5.46, 5.49, 5.50, 5.61 y 5.62 . Como ya se decía, las respuestas que envía el servidor en esta narrativa son recibidas en las expresiones 5.51 – 5.58 de la narrativa anterior.

Con esto queda cubierto cómo se da el flujo de actividades entre los usuarios estudiante y el sistema *Minerva*, en cuanto al envío y recepción de mensajes que requieren para comunicarse. Las instrucciones precisas que se tienen que hacer antes o después de cada expresión de las narrativas pueden ser completadas en la implementación.

5.6.4. Implementación en Erlang

Se presenta aquí el código correspondiente a las tres narrativas de arriba, dos del cliente estudiante y una del servidor *Minerva*.

5.6.4.1. Narrativas del cliente estudiante

Las dos narrativas del cliente estudiante tienen el mismo nombre de proceso, significa que se trata de la implementación de una misma función pero que difiere en el tipo y en el número de argumentos. El proceso *StudentClient* en la primera narrativa lleva un argumento mientras que en la segunda está sin argumentos. Así, la implementación Erlang para la primera narrativa del proceso cliente estudiante queda de la siguiente manera:

```
1 functionStudentClient(StudentCredential)->
2     minerva ! {self(),logon,StudentCredential},
3     receive
4       logon_accept-> functionStudentClient(StudentCredential);
5       logon_not_accept->
6     end.
```

La línea 4 del código tendrá que ser modificada para invocar a la misma función pero sin argumentos. Continuando con la segunda narrativa, las expresiones 5.33 y 5.34 hacen que se modifique el código anterior agregando una segunda declaración de `functionStudentClient` sin argumentos, línea 6:

```
1 functionStudentClient(StudentCredential)->
2     ...
3     logon_accept-> functionStudentClient();
4     ...
5
6 functionStudentClient()-> functionStudentClient().
```

El resto de las expresiones, 5.35 – 5.62, serán para el contenido de esta función sin argumentos. El código Erlang correspondiente es:

```
1 functionStudentClient()->
```

```
2     receive
3       test_request->
4           minerva ! {self(),test},
5           functionStudentClient();
6       do_test_request->
7           minerva ! {self(),test},
8           functionStudentClient();
9       {chat_to,Friend,Msg}->
10          minerva ! {self(),chat_to,Friend,Msg},
11          functionStudentClient();
12      who_request->
13          minerva ! {self (), who},
14          functionStudentClient();
15      {test,TypeTest}-> functionStudentClient();
16      {do_test,Test}-> functionStudentClient();
17      {chat_from,Friend,Msg}-> functionStudentClient();
18      {who,UserList}-> functionStudentClient();
19      logoff_request-> minerva ! {self(),logoff}
20    end.
```

5.6.4.2. Narrativa del servidor Minerva

La implementación Erlang para la narrativa del servidor Minerva es la siguiente:

```
1  -module(minerva).
2  -export([actuateStudentClient/0,functionMinerva/0]).
3
4  actuateStudentClient()-> register(minerva,spawn(minerva,functionMinerva,[])).
5
6  functionMinerva()->
7     receive
8       {FromStudent,logon,StudentCredential}->
9               % después de decidir, se enviará uno de estos dos mensajes:
10              FromStudent ! logon_accept,
11              FromStudent ! logon_not_accept,
12              functionMinerva();
13      {FromStudent,test}->
14              FromStudent ! {test,TypeTest},
15              functionMinerva();
16      {FromStudent,do_test}->
17              FromStudent ! {do_test,Test},
18              functionMinerva();
19      {FromStudent,chat_to,Friend,Msg}->
```

```
20              Friend ! {chat_from,FromStudent,Msg},
21              functionMinerva();
22      {FromStudent,who}->
23              FromStudent ! {who,UserList},
24              functionMinerva();
25      {FromStudent, logoff}->
26    end.
```

En este caso el servidor no posee ninguna instrucción para detenerse, ya que se asume que se estará ejecutando siempre. Notar que este código contempla el contenido de todo un módulo, en el cual no están presentes las funciones de cliente estudiante, por lo que el código para esas funciones pudiera ser agregado sin problemas a este módulo o estar en otro aparte.

Así la notación de las narrativas para modelar este tipo de sistemas puede ser leída y entendida sin conocer el lenguaje de programación Erlang y al mismo tiempo se estaría consiguiendo la construcción de código en el lenguaje.

Capítulo 6

Conclusiones y trabajo futuro

En este capítulo los resultados y las conclusiones finales de la investigación son mostrados y se destaca la contribución de la misma. Por último se mencionan los posibles trabajos futuros de investigación.

6.1. Resultados

Se distinguen explícitamente dos resultados claves en esta tesis, el primero de ellos es la notación ProME y el segundo es la correspondencia que existe de la notación hacia la sintaxis de programación concurrente de Erlang.

La notación ProME está basada en el formalismo cálculo de eventos y significa modelación de procesos Erlang. Contempla las acciones primordiales con las que trabajan los procesos de Erlang: la creación de procesos y, el envío y recepción de mensajes entre procesos. A estas acciones en la notación se les denomina eventos los cuales ocurren en determinado tiempo. Está notación sirve para modelar sistemas distribuidos con paso de mensajes. Si además el sistema se implementa en el lenguaje de programación Erlang, entonces se estaría consiguiendo una implementación parcial del mismo a la que sólo se le tendría que detallar un poco más la programación.

En este trabajo se presenta la constitución de la notación ProME, ya que se detallan y exponen sus componentes: una gramática regular que representa el lenguaje de la misma, la sintaxis correspondiente de cada uno de sus elementos, un autómata finito determinista que precisa la estructura de las narrativas de la notación y una metodología para aplicar la notación.

También aquí se ha desarrollado la correspondencia entre la sintaxis de cada elemento de la notación ProME y la sintaxis de programación concurrente Erlang. Los procesos, argumentos, los predicados *Happens*, *Initiates* y *Terminates*, los eventos *actuate*, *end*, *send* y *receive*, el fluente *live*, los tiempos y las narrativas en la notación ProME tienen su representación en Erlang, ya sea como átomo, variable, otro tipo de dato Erlang, lanzamiento de un proceso, registro de un proceso, módulo, función, operador o cláusula, según corresponda.

6.2. Conclusiones

De acuerdo a la hipótesis planteada en un inicio y a los resultados obtenidos de la tesis se concluye que la hipótesis es verdadera, esto es, que el cálculo de eventos sí es una herramienta apropiada para poder especificar sistemas concurrentes y distribuidos con paso de mensajes. Esto lo demuestra la obtención de la notación ProME que se puede usar para modelar este tipo de sistemas y además interpretarse hacia la programación concurrente Erlang.

Los objetivos de la investigación también se han cumplido, ya que no sólo se obtuvo una notación basada en el cálculo de eventos para especificar sistemas distribuidos con paso de mensajes, sino que también la correspondencia unidireccional a la sintaxis que emplea Erlang para la programación concurrente.

Así, la notación ProME es una especificación alternativa y complementaria para describir el comportamiento de un sistema distribuido con paso de mensajes, y que también puede auxiliar la programación de estos sistemas cuando se implementan con el lenguaje Erlang. Una ventaja de la notación es la modelación del tiempo ya que provee: la granularidad necesaria, el orden de los eventos y la consistencia. Esto permite gestionar en qué orden se ejecutan los procesos y cómo se comunican: el envío y recepción de mensajes. Otra ventaja es que con las narrativas que se vayan construyendo del sistema es posible anticiparse al comportamiento que tendrá este antes de su implentación, lo que daría lugar a corregir tempranamente la modelación, así evitar costos de tiempo y trabajo.

Como ya se ha documentado, la obtención de la notación ProME fue posible gracias a la identificación de una estrecha relación entre los eventos en el cálculo de eventos y un conjunto bien delimitado de acciones del paradigma de comunicación de procesos paso de ensajes. Entonces con esta herramienta se puede tratar la especificación no sólo de sistemas distribuidos completos sino además de programas Erlang. Incluso puede ser utilizada como parte de la documentación del desarrollo y en la verificación del código en las etapas de implementación y pruebas.

La primera contribución en este trabajo ha sido la de aplicar el cálculo de eventos para la especificación de sistemas distribuidos con paso de mensajes y hallar la correspondiente interpretación hacia Erlang. La segunda contribución es la obtención de herramientas para la especificación de estos sistemas: la notación ProME y la correspondencia con Erlang. Estas contribuciones se extienden a los campos de estudio de: aplicaciones de la lógica, sistemas distribuidos, ingeniería de software, ingeniería de requisitos, programación concurrente y a la programación en Erlang como tal.

6.3. Trabajo futuro

Los resultados encontrados en esta tesis tienen como consecuencia algunos trabajos a desarrollar en un futuro. Estos involucran varios aspectos para fortalecer la investigación presentada aquí, que van desde obtener una segunda versión de la notación ProME hasta dotarla de un método de verificación que asegure la correcta construcción del software, pasando por la búsqueda de reglas de transformación que faciliten la automatización hacia Erlang.

Como se menciona, la mejora y explotación de la notación ProME incluye la creación de la siguiente versión de esta. Esto sería modificaciones a la notación original para obtener una versión reducida, es decir, utilizar abreviaturas o símbolos que permitan que sea más concisa. Se puede pensar en que el nombre de los eventos y predicados sean sustituidos, respectivamente, por letras griegas minúsculas y mayúsculas adecuadas. Esto permitirá una lectura rápida de la especificación y resaltará en ella a los procesos y la forma de comunicarse entre estos: el paso de mensajes.

Otra modificación que puede haber en la notación ProME es la de otorgar la posibilidad a los predicados de aceptar como parámetros la declaración de asignación de los eventos. Por ahora sólo pueden recibir el nombre del evento, la asignación del evento tiene que ser hecha previamente. De ser necesario, tampoco se puede descartar la inclusión de nuevos eventos y predicados a la notación.

Dado que la correspondencia entre la notación ProME y la programación concurrente Erlang presentada aquí, independientemente de la sintaxis de ambas partes, posee varias reglas semánticas, esos puntos correlativos deben establecerse en forma de reglas de transformación de la notación ProME a Erlang. En el caso de los ejemplos y el caso de estudio presentados en este documento, la traducción de las narrativas a Erlang fue hecha de forma manual (mecánica). Tal vez en esto el enfoque de desarrollo de software empleando modelos: MDA (*Model Driven Architecture*) [MM03] pueda ser útil. Este enfoque fue aceptado en el

2001 como framework por el organismo OMG (*Object Management Group*) para proporcionar herramientas que permitan la especificación de un sistema independientemente de la plataforma de implementación, la elección de una plataforma en particular para el sistema y la transformación de la especificación del sistema a una plataforma en particular. Esto último es lo que sería de interés en la búsqueda de las reglas de transformación.

Derivado de las reglas de transformación, se puede conseguir la automatización de las narrativas ProME a código Erlang. La automatización involucra la creación de un compilador que permita la obtención de los archivos Erlang a partir de especificaciones ProME. Esta herramienta de software puede ser implementada como un módulo más que se pueda trabajar desde la consola Erlang, un complemento en algún IDE como Eclipse o poderse ejecutar en un navegador web sobre una simple página. La construcción automática de código Erlang agilizará la fase de implementación y cualquier modificación en la narrativa enseguida se vería reflejada en el código.

Por último, proveer de un método de verificación a la notación ProME que ayude a cerciorarse que se está construyendo correctamente el software, es decir, que la implementación se ajuste o cumpla con la especificación inicial: las narrativas ProME construidas. Por ejemplo, cuando se realicen las inspecciones al código, a los programas o a los módulos para verificar que cumplen con los requisitos planteados, este método servirá para ello y por ende asegurará aún más la calidad del software.

Apéndices

Apéndice A

Erlang básico

Este apéndice describe los elementos básicos del lenguaje de programación Erlang como son: interfaz de usuario, sintaxis, tipos de datos, tipos de operadores, estructuras, compilación, ejecución, entre otros. Cumple con la distribución y versión Erlang/OTP R16B01 revisada en la realización de esta tesis. El apéndice pretende servir de referencia rápida para la comprensión del lenguaje. También se puede encontrar información más detallada en los siguientes libros: [Arm07, CT09, LMC11], el primero de ellos fue publicado por Joe Armstrong, creador de Erlang. Dichas fuentes se consultaron para la realización de este apartado.

A.1. Programando en Erlang

Erlang es un lenguaje funcional basado en el lenguaje Prolog, tolerante a fallos, orientado al trabajo en tiempo real y a la concurrencia. Fue diseñado desde un inicio para que los programas Erlang se ejecuten de forma ininterrumpida, esto siginifica, que puede modificarse el código sin detener la ejecución.

Erlang no sólo proporciona un compilador para ejecutar el código, sino se considera una plataforma de desarrollo ya que posee una colección de herramientas para la programación como la interfaz de línea de comandos y una máquina virtual sobre la que se ejecuta el código compilado, la cual lo dota de la capacidad de manejar y distribuir procesos de manera independiente a los procesos que emplea el sistema operativo en el cual opera. Así en cada distribución de Erlang está incluido el marco de trabajo *Open Telecom Platform* (OTP) que proporciona maneras de estructurar sistemas Erlang, robustez, tolerancia a fallos y un conjunto de herramientas y bibliotecas para desarrollar sistemas grandes y completos.

Como ya se dijo, Erlang es un lenguaje de tipo funcional, por lo que su sintaxis está diseñada como si se tratará de la definición de una función matemática o de una proposición lógica, a diferencia de los lenguajes imperativos que se basan en la consecución de mandatos. Por lo que al momento de programar se piensa en el *qué* se quiere en lugar del *cómo* obtenerlo.

La última versión de Erlang así cómo las anteriores se pueden obtener de el sitio web oficial *www.erlang.org*, así también una basta documentación del lenguaje para comenzar a aprender o entrar en más profundidad en cierto tópicos que interesen.

A.2. El intérprete de comandos de Erlang

La interfaz de usuario que viene en una distribución Erlang es de tipo texto, es decir, una consola en la que se ejecutan líneas de comandos (*shell*). Para comenzar a utilizarla en sistemas operativos como MAC OS, Unix y Linux se invoca con el comando `erl` y en sistemas Windows se incia al lanzar el archivo ejecutable correspondiente. Una vez iniciado el shell es posible tipear expresiones y esperar el resultado a mostrarse después de presionar la tecla entrar (*enter*). Todas las expresiones a ser evaluadas deben terminar con un punto (`.`). Enseguida se muestra el inicio de ejecución de la línea de comandos de Erlang para luego ingresar un comentario que inicia con el caracter % por lo que no es tomado en cuenta por el intérprete, después se ingresa la expresión aritmética 10 + 20 y se muestra el resultado 30. Finalmente la consola está disponible en la línea 2 a la espera de más instrucciones Erlang.

```
Erlang R16B01 (erts-5.10.2) [64-bit] [smp:4:4] [async-threads:10]

Eshell V5.10.2  (abort with ^G)
1>%Esto es un comentario
1>10+20.
30
2>_
```

En caso de no terminar una expresión con un punto, el intérprete no evaluará lo que se haya ingresado y continuará recibiendo entradas hasta que se ingrese el punto:

```
2>3
2>*
2>4
2>+
2>8.
20
3>_
```

Para acceder a los resultados obtenidos previamente en la consola se hace uso de la función `v/1`, que recibe como parámetro un número entero positivo, y puede devolver los últimos 20 resultados arrojados por el intérprete. Así de los ejemplos previos, con `v(1)` se accede al primer resultado, 30, que arrojo de la ejecución de la línea 1 y con `v(2)` al resultado 20 de la línea 2. Por ejemplo, al sumar estos resultados se obtiene la suma total de 50:

```
3>v(1) + v(2).
50
4>_
```

Cuando se introduce una expresión inválida se obtendrá un error al respecto. Para terminar la ejecución de la consola se hace uso de la función `q/0`, como se muestra a continuación:

```
4>5-.
* 1: syntax error before: '.'
5>q().
```

A.3. Tipos de datos

Erlang emplea varios tipos de datos entre los que se pueden distinguir los simples y los complejos (o compuestos). En los simples entran los tipos de datos numéricos, átomos y booleanos. En los compuestos están las tuplas, la fecha, listas, cadenas y registros. A los datos en Erlang se les conoce generalmente como "términos".

A.3.1. Numéricos

Existen dos tipos de datos numéricos: los números enteros y los números reales. En las operaciones aritméticas la conversión se realiza automáticamente entre los tipos de números, esto hace que normalmente no se tenga que hacer ningún tipo de conversión explícita.

A.3.1.1. Números enteros

Los números enteros en Erlang pueden ser de tamaño arbitrario. Sean números pequeños o grandes el espacio necesario en memoria se asigna automáticamente, por lo que es completamente transparente para el programador, eso significa que nunca hay que preocuparse por el truncamiento de las cantidades o desbordes de memoria, en el ejemplo siguiente se muestra esto. También es posible emplear números de base 2, 16 y 32:

```
1>1234567890 * 9876543210 * 9999999999.
121932631100442005888736473100
2>2#1010.
10
3>16#FF.
255
4>36#ZZ.
1295
```

A.3.1.2. Números reales

A los números reales en Erlang se les conoce como “flotantes”(tipo *float*), la precisión de estos números está definido por una representación de 64 bits. Acepta la notación decimal con exponente representado por `E` o `e`. Aquí algunos ejemplos:

```
1>9.9 * 20  * -1.5666666 + 100.
-210.19998679999998
2>1.234E-10 * 10.
1.234e-9
```

A.3.2. Átomos

En Erlang un átomo es un tipo de dato que se identifica sólo por los caracteres que lo forman, por lo que dos átomos siempre se consideran exactamente iguales si tienen la misma representación de caracteres. En analogía con otros lenguajes vendrían siendo las constantes `#define` en el caso de C y C++, los valores `static final` en Java y `enums` en Ruby. Las únicas operaciones que se pueden realizar con los átomos son las comparaciones. Letras, números, el símbolo arroba (@), el punto (.) y guiones bajos (_) son caracteres válidos para formar un átomo siempre y cuando se inicie con una letra minúscula. Cualquier caracter está permitido dentro de un átomo si es encapsulado por comillas simples. Ejemplos de átomos válidos que comienzan con una letra minúscula son:

```
noviembre idReg var20 x_y1 nodo@local true false
```

Y átomos encapsulados por comillas simples:

```
'Noviembre' 'un espacio' 'Cualquier cosa {}#@ \n\012' '_nodo@local.erl'
```

A.3.3. Booleanos

No hay valores o caracteres booleanos en Erlang. En lugar de un tipo booleano, los átomos `true` y `false` son los que se utilizan junto con los operadores relacionales, que más adelante se presentan, para conseguir expresiones lógicas. Aquí algunos ejemplos:

```
1>1<1.
false
2>0.999<1.
true
3>true and false.
false
4>true or false.
true
```

A.3.4. Tuplas

Una tupla es un tipo de dato compuesto que se utiliza para almacenar una colección de elementos, los cuáles son valores de tipos de datos Erlang, pero que no tienen que ser todos del mismo tipo. Las tuplas son delimitados por llaves, $\{\ldots\}$, y sus elementos están separados por comas. Algunos ejemplos de tuplas incluyen:

```
{123, bcd} %Con dos elementos: el entero 123 y el átomo bcd.
{abc, {def, 123}, ghi} %Con tres elementos, el segundo es una tupla.
{}    %Es una tupla vacía, sin elementos.
{100} %Con un elemento, es válido pero no tiene sentido utilizarlo así.
```

Cuando el primer elemento de una tupla es un átomo se dice que este es la etiqueta (*tag*) de la tupla, se considera una buena práctica de programación. Esta buena práctica se utiliza para representar diferentes tipos de datos y usualmente tienen un significado en el programa que se declara. Se puede usar en aquellos casos en que es más fácil acceder al elemento por un identificador conocido que por un índice que podría ser desconocido, por ejemplo:

```
{persona, ‘Joe’, ‘Armstrong’, 50, M}
%El átomo persona es la etiqueta de la tupla e informa que dicha tupla
%contiene datos de una persona como su nombre, edad y sexo.
```

A.3.5. Fecha y hora

El manejo del tiempo en Erlang no se realiza con un tipo de dato estándar, sino que se establece como un término encerrado en una tupla. Una fecha es una tupla con tres

elementos enteros destinados al año, mes y día, respectivamente. La función `date/0` retorna este formato. De la misma manera que la fecha, la hora se maneja en una tupla de tres elementos: hora, minutos y segundo. La función `time/0` devuelve esta tupla. Por lo que un tipo de dato completo que incluya la fecha y hora sería una tupla que contiene a las dos tuplas anteriores. Ejemplos:

```
1>date(). %Devuelve la fecha local
{2013,11,20}
2>time(). %Devuelve la hora local
{10,30,20}
3>erlang:localtime(). %Devuelve la fecha y hora local
{{2013,11,20},{10,31,20}}
```

A.3.6. Listas

Las listas y las tuplas se utilizan para almacenar colecciones de elementos, en ambos casos, los elementos pueden ser de diferentes tipos y las colecciones pueden ser de cualquier tamaño. Sin embargo, las listas y tuplas son muy diferentes en la forma que pueden ser procesadas. Las listas son delimitadas por corchetes, [. . .], y sus elementos están separados por comas. Al igual que en una tupla, los elementos de las listas no tienen que ser del mismo tipo de dato y se pueden mezclar libremente. Algunos ejemplos de listas incluyen:

```
[enero,febrero,marzo,abril,mayo] %Con cuatro elementos.
[a,[b,[1,2,3],f],g] %Con tres elementos, el segundo es de tipo lista.
[] %Es una lista vacía, sin elementos.
[{persona,‘Juan’,22}, {persona,‘Pedro’,24}, {persona,‘Miguel’,20}]
%Es una lista que contiene tres elementos que son de tipo tupla.
```

Para poder agregar o sustraer elementos a una lista se utilizan los operadores ++ y - - respectivamente. Ejemplo:

```
1> [1,2,3] ++ [4,5,6].
[1,2,3,4,5,6]
2> [1,2,3] -- [2,3].
[1]
```

Las listas y tuplas poseen varias características similares, sin embargo la diferencia entres estas radica en cómo se procesan. Una tupla puede utilizarse únicamente para la extracción de elementos particulares, mientras que una lista es posible partirla en una cabecera (*head*) y una cola (*tail*), siempre y cuando la lista no esté vacía. La cabecera se refiere al primer elemento de la lista y la cola es otra lista que contiene todos los elementos restantes. Por

ejemplo si se toma la lista [1,2,3] la cabcera será 1 y la cola la lista [2,3]. Al utilizar el operador | la lista puede ser representada como $[1|[2,3]]$, de lado izquierdo del operador se coloca la cabecera y en la derecha la cola. Con esta misma notación se puede seguir partiendo la lista para tener otras representaciones más: [1|[2|[3]]] y [1|[2|[3|[]]]]. Sin embargo Erlang también permite tener en la cabecera más de un elemento separados por coma: $[1, 2|[3|[]]]$. Todas estas listas son equivalentes a la lista original [1,2,3]. Aquí un ejemplo más donde todas las listas son equivalentes semánticamente:

```
[uno, dos, tres, cuatro]
[uno, dos, tres, cuatro|[]]
[uno, dos|[tres, cuatro]]
[uno, dos|[tres|[cuatro|[]]]]
[uno|[dos|[tres|[cuatro|[]]]]]
```

A.3.7. Cadenas

En Erlang los caracteres son representados por números enteros y una cadena de caracteres se puede representar por una lista que contiene enteros. Para conocer el número correspondiente a cada caracter se utiliza el símbolo $:

```
1> $A.
65
2> $A + 32.
97
3> $a.
97
4> $1. %Al intentar hacerlo con un número se obtiene un error.
syntax error
```

En realidad, no hay un tipo de dato cadena en Erlang. Las cadenas son un tipo específico de lista que tiene como elementos valores enteros ASCII. Las cadenas también se pueden denotar como texto encerrado entre comillas dobles ("). La cadena vacía "" es equivalente a una lista vacía [ ]. La diferencia entre un átomo declarado entre comillas simples y una cadena de texto encerrada entre comillas dobles es el uso que se dará. Los átomos sólo se pueden comparar, es la única operación con la que se puede trabajar con ellos, mientras que las cadenas pueden procesarse de diferentes maneras como separarse, unirse, agregar más texto. A continuación se ejemplifica todo lo anterior con el texto "Hola Mundo":

```
1> [65,66,67]. %Valores ASCII
"ABC"
2> 'Hola Mundo'. %No confundir, esto es un átomo, no una cadena.
```

```
'Hola Mundo'
3> [72,111,108,97,32,77,117,110,100,111]. %Con valores ASCII
"Hola Mundo"
4> [$H,$o,$l,$a,$ ,$M,$u,$n,$d,$o]. %Con la notación del símbolo $.
"Hola Mundo"
5>"". %Cadena vacía
[]
6> "Hola " ++ "Mundo". %Cadenas de texto, concatenación de listas.
"Hola Mundo"
```

A.3.8. Registros

Un registro es una estructura de datos con un número fijo de campos que son accedidos por su nombre, similar a una estructura en el lenguaje C o un registro en Pascal. Difiere de las tuplas donde los campos son accedidos por su posición. Los campos en un registro son definidos como átomos. Por ejemplo, para el caso de datos de una persona se define un tipo registro de la siguiente manera y una instancia del registro respectivamente:

```
-record(persona, {nombre,telefono,edad}). %Declaración del registro

#persona{nombre="Joe Armstrong", %Una unstancia del registro persona
         telefono="128-5699-56",
         edad=50}

#persona{edad=30,         %Otra instancia del registro persona con un
         nombre="María", %orden de campos distinto a la declaración.
         telefono="998-4109-67"}
```

Para declarar una instancia del registro persona se requiere de un constructor, esto es utilizando el símbolo # seguido del nombre del registro, para luego asignar valores a los campos como si fueran elementos de una tupla. Es posible establecer valores por defecto a los campos en la definición del registro, siguiendo el mismo ejemplo sería:

```
-record(persona, {nombre,phone="",edad=0}).
#persona{nombre="Juan"} %Instancia válida, con teléfono vacío y edad cero.
```

Así, la definición general de un registro es de la siguiente manera:

```
-record{nombre_registro, {campo1[ = default1],
                          campo2[ = default2],
                          ...
                          campon[ = defaultn]}
```

Las partes encerradas en corchetes son opcionales. Cabe mencionar que el nombre de un campo puede ser usado en más de un registro, sin embargo el nombre del registro puede ser empleado solamente en una definición, ya que con ello se identifica al registro.

Es muy importante mencionar que la palabra reservada `record` no es un comando de la consola de Erlang. Para declarar un registro con esta palabra únicamente se puede hacer en módulos (archivos) de código fuente Erlang (que más adelante se aborda) con extensión `erl` o en archivos con extensión `hrl` que son para definir registros. Para cargarlos (poder leerlos) en la consola se hace uso del comando `rr(`nombre_archivo`)`. Para crear registros directamente en la consola se utiliza el comando `rd(`nombre_registro,{campo1, campo2,...}`)`, esto se suele emplear para realizar pruebas y depuraciones de programas. Finalmente, la función `rl()` muestra en consola el listado de registros que se han definido.

A.3.9. Identificador de procesos

Debido a que Erlang fue concebido principalmente para soportar programación con miles de procesos a la vez es importante conocer este tipo de dato, que en realidad es informativo pero muy útil en la programación concurrente. Cualquier código ejecutable en Erlang es un proceso, por lo que cada cada vez que se crea un proceso se le asigna un identificador (PID, *Process Identifier*), usualmente se les conoce como *pid*. Los *pids* son un tipo de dato especial en Erlang. Tienen el aspecto de <*Entero.Entero.Entero*>, por ejemplo <0.35.0>, es decir tres enteros separados por un punto y encerrados entre corchetes angulares. Esta sintaxis no es posible introducirla en la consola y crear un *pid*, sino que sólo se muestra y existe para fines de depuración e información acerca de los procesos en ejecución. Por ejemplo en la ejecución de código concurrente es muy común que algunas funciones retornen el *pid* de algún proceso como información a otro proceso o quizás al programa principal.

La función `self()` retorna el *pid* del proceso que actualmente se está ejecutando, incluso se puede probar con la consola, ya que está es también un proceso en Erlang:

```
1> self().
<0.93.0>
```

A.3.10. Variables

En Erlang las variables se utilizan para almacenar valores de tipos de datos simples y compuestos, todos los que ya se han visto. Comienzan con una letra mayúscula o con un

guión bajo, seguidas por letras mayúsculas, minúsculas, números enteros y guiones bajos. No pueden contener otros caracteres especiales. Ejemplos de variables son:

```
Una_variable Bandera Nombre1 NumCta _var
```

Las variables en Erlang difieren de las variables de la mayoría de los lenguajes de programación convencionales, ya que una vez que se le haya asignado un valor a una variable no se puede cambiar dicho valor, es decir, cada variable durante su tiempo de vida sólo puede contener un valor. Esto se llama *asignación única*. Por lo tanto, si se tiene que hacer un cálculo manipulando el valor de una variable, el resultado se debe almacenar en una nueva variable. Por ejemplo, al escribir lo siguiente resultaría un error en tiempo de ejecución:

```
1> X=2.
2
2> X=X*X.
** exception error: no match of right hand side value 4
3> Y=X * X.   %Creando la varibale Y a partir de X.
4
```

debido a que X ya tiene asigando el valor de 2. Todas las llamadas a variables en Erlang son *llamadas por valor*: todos los argumentos de la llamada de una función se evalúan antes de evaluar el cuerpo de la función. El concepto de *llamada por referencia* no existe. También, todas las variables en Erlang son consideradas locales a la función en las que fueron creadas. No existen las variables globales, por lo que es más fácil depurar los programas Erlang y reducir el riesgo de errores y malas prácticas de programación.

Otra característica útil de las variables en Erlang es que no hay necesidad de declararlas: sólo utilizarlas. La razón para no declarar el tipo de dato es que Erlang posee un *sistema de tipo dinámico*. Los tipos de datos se determinan en tiempo de ejecución, al igual que la viabilidad de la operación que se está intentando ejecutar sobre la variable. El siguiente código intenta multiplicar un átomo por un entero, lo cual en tiempo de ejecución se obtendrá un error, pero no en compilación ya que sintácticamente es correcto:

```
1> Var= uno.
uno
2> X= Var * 2.
** exception error: an error occurred when evaluating an arithmetic
   expression
   in operator  */2 called as uno * 2
```

Para desvincular una variable de su valor existen dos funciones que se pueden utilizar. La función `f()` que borra todas las asignaciones de variables que se hayan hecho previamente,

mientras que `f(Variable)` desvincula una variable específica de su valor. Estas operaciones sólo se pueden utilizar en la consola:

```
1> A= (1+2)*3.
9
2> f(A).
ok
3> B= A * 2.
** 1: variable ‘A’ is unbound **
4> A= 11.
11
5> B= A / 2.
5.5
6> f().
ok
7> A.
** 1: variable ‘A’ is unbound **
8> B.
** 1: variable ‘B’ is unbound **
```

A.4. Tipos de operadores

En esta sección se listan los operadores básicos que Erlang maneja: los aritméticos, lógicos, relacionales y aritméticos a nivel de bits. En la parte final se presenta la prioridad de los operadores en Erlang.

A.4.1. Operadores aritméticos

Las operaciones con números enteros y reales en Erlang incluyen la suma, resta, multiplicación y división. Los operadores + y - también se pueden utilizar como operadores unarios para representar números positivos y negativos respectivamente, como -12 o +10.5. Las operaciones que se realicen unicamente con números enteros siempre resultará en un número entero, excepto en el caso de la división de punto flotante, donde el resultado siempre es un flotante. En la tabla A.1 se enumeran los operadores aritméticos.

Operador	Descripción	Operadores
+	Unario +X	Númerico
-	Unario -X	Númerico
*	Multiplicación X*Y	Númerico
/	División de punto flotante X/Y	Númerico
div	División entera X div Y	Entero
rem	Residuo entero X rem Y	Entero
+	Suma X+Y	Númerico
-	Resta X-Y	Númerico

Tabla A.1: Operadores aritméticos en Erlang.

A.4.2. Operadores lógicos

Los operadores lógicos en Erlang son binarios, a excepción de la negación que es unaria. En la tabla A.2 se presentan estos operadores, los cuales evaluan valores booleanos o expresiones que devuelvan un valor booleano, valga la redundancia.

Operador	Descripción
and	Retorna `true` sólo si ambos argumentos son verdaderos.
andalso	Retorna `false` si el primer argumento es falso, sin evaluar el segundo.
or	Retorna `true` si cualquiera de los argumentos es verdadero.
orelse	Retorna `true` si el primer argumento es verdadero, sin evaluar el segundo.
xor	Retorna `true` si uno de sus argumentos es verdadero y el otro falso.
not	Retorna `true` si el argumento es falso, y viceversa

Tabla A.2: Operadores lógicos en Erlang.

A.4.3. Operadores relacionales

También llamados en Erlang comparadores de términos, toman dos expresiones, uno de cada lado del operador. El resultado de la expresión es uno de los dos átomos booleanos `true` o `false`. Los operadores *igual a* (`==`) y *diferente a* (`/=`) comparan los valores sin prestar atención a los tipos de datos. Ejemplos: `1==uno` retorna `false`, mientras que `uno==uno` retorna verdadero. Así, `1==1.0` retorna verdadero y `1/=1.0` devuelve `false`. En las dos últimas

operaciones los enteros se convierten a flotantes antes de ser comparados.

Los operadores *exactamente igual a* (`=:=`) y *exactamente diferente a* (`=/=`) no sólo comparan los dos valores de ambos lados sino también sus tipos de datos. Por ejemplo, `1=:=1.0` y `1=/=1` retornan `false` y `1 =/= 1.0` devuelve `true`. Además de las comparaciones de igualdad y desigualdad también es posible comparar el orden entre los valores utilizando los operadores *menor que*, *menor o igual que*, *mayor que* y *mayor o igual que*. En la tabla A.3 se muestran todos estos operadores de comparación.

Operador	**Descripción**
==	Igual a
/=	Diferente a
=:=	Exactamente igual a
=/=	Exactamente diferente a
<	Menor que
=<	Menor o igual que
<	Mayor que
>=	Mayor o igual que

Tabla A.3: Operadores relacionales en Erlang.

A.4.4. Operadores a nivel de bits

Los operadores artiméticos a nivel de bits en Erlang se presentan en la tabla A.4. Los operandos solamente pueden ser de tipo entero. Aquí algunos ejemplos de sus usos:

```
1> 3 band 1.
1
2> 3 bor 1.
3
3> 3 bxor 1.
2
4> bnot 3.
-4
5> 2 bsl 1.
4
6> 2 bsr 1.
1
```

Operador	**Descripción**	**Operadores**
bnot	Unario, not a nivel de bits.	Entero
band	and a nivel de bits.	Entero
bor	or a nivel de bits.	Entero
bxor	xor a nivel de btis.	Entero
bsl	Desplazamiento a la izquierda.	Entero
bsr	Desplazamiento a la derecha.	Entero

Tabla A.4: Operadores aritméticos a nivel de bits en Erlang.

A.4.5. Precedencia de operadores

Cuando se evalúa una expresión, el operador con mayor precedencia es quien se evalúa primero. En caso de operadores con la misma prioridad entonces se evalúa de acuerdo a la asociatividad. En la tabla A.5 se muestra la prioridad de los operadores en Erlang, siendo el de mayor prioridad el de la primera fila y de menor prioridad el de la última fila.

Operador	**Descripción**
:	Ejecución de funciones
#	Resolución de registros
+ - bnot not	Unarios
/ * div rem band and	Asociatividad a la izquierda
+ - bor bxor bsl bsr or xor	Asociatividad a la izquierda
++ - -	Asociatividad a la derecha
== /= =< < >= > =:= =/=	Operadores relacionales
andalso	Comprobación simple de `and`
orelse	Comprobación simple de `or`
= !	Asociatividad a la derecha
catch	Captura de errores

Tabla A.5: Precedencia de operadores en Erlang.

A.5. Coincidencia de patrones de Erlang

La coincidencia de patrones (*pattern matching*) en Erlang es utilizada para:

- Asignar valores a variables.
- Controlar la ejecución del flujo del programa.
- Extraer valores desde tipos de datos compuestos.

La combinación de estas características permite escribir programas concisos, legibles y potentes, particularmente cuando se usa para manejar los argumentos de una función.Una coincidencia de patrón se escribe como `Patron=Expresion`, que en realidad es una generalización de la creación de variables que se vió anteriormente. El `Patron` consiste de estructuras de datos que contiene tanto variables ligadas y no ligadas, así como valores literales: átomos, enteros o cadenas. Una variable *ligada* es una variable que ya tiene un valor y una *no ligada* es una que todavía no tiene un valor. Ejemplos de patrones:

```
Doble
{Doble, 34}
{Doble, Doble}
[true, Doble, 23, {34, Triple}]
```

Por otro lado, la `Expresion` consiste de estructuras de datos, variables ligadas, operaciones matemáticas y llamadas de funciones. Podría no contener valores no ligados. Cuando una coincidencia de patrón es ejecutada, dos resultados son posibles:

- Éxito (*succeed*), esto se traduce en que las variables no ligadas se convierten en ligadas, esta ejecución retorna el valor de la expresión.
- Fallo (*error*), las variables no pudieron ser ligadas.

Lo que determina el éxito en una coincidencia de patrones es la comparación entre `Expresion` y `Patron`, se evalúa primero la expresión y después este valor es comparado con el patrón:

- La expresión y el patrón necesitan ser de la misma forma: una tupla de tres elementos puede coincidir únicamente con una tupla de tres elementos, una lista de la forma [X|Xs] puede coincidir sólo con una lista vacía, etcétera.
- Las *literales* en el patrón tienen que ser iguales a los valores ubicados correspondientemente en la expresión.
- Las variables no ligadas que se encuentran en el patrón son ligadas con el valor correspondiente en la expresión.

En la mayoría de los lenguajes de programación el símbolo = denota una sentencia de asignación, en Erlang sin embargo denota una operación de coincidencia de patrones. La expresión LI=LD realmente significa: evaluar el lado derecho (LD) y entonces el resultado compararlo con el patrón del lado izquierdo (LI). El caso de las variables, es una forma simple de patrón. Cuando se tiene por vez primera una expresión como `X=valor1`, debido a que X todavía no está vinculada a un valor entonces sí se puede vincular con la expresión `valor1`.

A.6. Funciones como datos: *fun*

Como Erlang es un lenguaje de programación funcional y una característica esperada de un lenguaje de este tipo es que debe ser capaz de manejar funciones como datos, es decir, pasar una función como entrada a otra función, retornar una función como resultado de otra función, poner una función en una estructura de datos y recogerla después. En Erlang tal objeto *función como dato* se denomina *fun* o a veces *expresión lambda* o cierre (*closure*). Se dice también que son funciones "anónimas" porque no tienen nombre. Se definen entre las palabras reservadas `fun` y `end`. El caso más sencillo posible de esta función es no recibir ningún argumento y siempre devolver cero:

```
1>fun() -> 0 end.
```

Enseguida otro ejemplo donde se define una función *fun* que recibe un parámetro y se vincula a una variable `Z`:

```
1> Z= fun(X) -> 2*X end.
#Fun<erl_eval.6.56006484>
```

Cuando se define este tipo de función la consola Erlang imprime `#Fun`< ... > donde los puntos representan información de depuración para el lenguaje. La variable a la que se vincula `fun` ahora se puede tratar como si fuera de una función y a su vez se puede asignar a otra variable:

```
2> Z(3).
6
3> Mult2=Z.
#Fun<erl_eval.6.10732646>
4> Mult2(4).
8
```

En realidad las funciones *fun* pueden tener cualquier número de argumentos así como contener varias cláusulas:

```
%Función que calcula la hipotenusa de un triángulo
```

```
Hipotenusa = fun(X,Y) -> math:sqrt(X*X + Y*Y) end.

%Más de una cláusula en la definición de la función
Area= fun({circulo,Radio})-> Radio*Radio*math:pi();
         ({cuadrado,Lado})-> Lado*Lado;
         ({rectangulo,Alto,Ancho})-> Alto*Ancho
      end.
```

La palabra reservada `fun` también se utiliza para llamar funciones definidas dentro de un mismo módulo, es decir, una función definida con un nombre y con (o sin) parámetros puede vincularse a una variable o pasarse directamente como parámetro a otra función usando esta palabra reservada, por ejemplo:

```
%Definición de la función hipotenusa y su
%corresponiente vinculanción a la variable H:
hipotenusa(X,Y) -> math:sqrt(X*X + Y*Y) end.
...
H= fun hipotenusa/2.

%Pasando la función hipotenusa como argumento
imprimir_resultado(fun hipotenusa/2)
```

Bibliografía

[Ale95] Vladimir Alexiev. The event calculus as a linear logic program, 1995.

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, y Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Pearson, Addison-Wesley, 2 edition, 2007.

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf, 2007.

[BK82] Jan A Bergstra y Jan Willem Klop. Fixed point semantics in process algebras. *Mathematical Centre*, 1982.

[Bou14] Clint Boulton. Facebook WhatsApp Messaging Service Written in Exotic Erlang. http://blogs.wsj.com/cio/2014/02/24/facebook-whatsapp-messaging-service-written-in-exotic-erlang/, 2014. The Wall Street Journal. [Accesado 28-Feb-2014].

[Bro86] Rodney A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, 1986.

[Bro89] J. Glenn Brookshear. *Theory of computation: formal languages, automata, and complexity*. Benjamin-Cummings Publishing Co., Inc., 1989.

[CDKB11] G.F. Coulouris, J. Dollimore, T. Kindberg, y G. Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley, 5 edition, 2011.

[CGH+09] Hugo Cortés, Mónica García, Jorge Hernández, Manuel Hernández, Esperanza Pérez-Cordoba, y Erik Ramos. Development of a distributed system applied to teaching and learning. En *Proceedings of the 8th ACM SIGPLAN workshop on ERLANG*, ERLANG '09, pages 41–50. ACM, 2009.

[CGH+10] Hugo Cortés, Mónica García, Jorge Hernández, Manuel Hernández, Esperanza Pérez-Cordoba, y Erik Ramos. Un framework para desarrollar sistemas distribuidos aplicados a enseñanza y aprendizaje, usando erlang como componente principal. *Temas de Ciencia y Tecnología*, 14(42):3–18, 2010.

[CT09] F. Cesarini y S. Thompson. *Erlang Programming*. O'Reilly Media, 2009.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[DK02] Marc Denecker y C. Kakas, Antonis. Abduction in logic programming. En *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, Lecture Notes in Computer Science, pages 402–436. Springer Verlag, 2002.

[Eri14] Ericsson. wxErlang Reference Manual. http://www.erlang.org/doc/apps/wx/, Enero 2014. [Accesado 17-Feb-2014].

[Gho07] Sukumar Ghosh. *Distributed Systems: An Algorithmic Approach*. Chapman & Hall/CRC Computer & Information Science Series. Taylor & Francis, 2007.

[Ham82] A.G. Hamilton. *Numbers, Sets and Axioms: The Apparatus of Mathematics*. Cambridge University Press, 1982.

[Hen07] Matthew Hennessy. *A Distributed Pi-Calculus*. Cambridge University Press, 2007.

[HJD05] M. Elizabeth C. Hull, Ken Jackson, y Jeremy Dick. *Requirements Engineering*. Springer, 2 edition, 2005.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., 1985.

[HV09] Krystof Hoder y Andrei Voronkov. Comparing Unification Algorithms in First-Order Theorem Proving. En Bärbel Mertsching, Marcus Hund, y Muhammad Zaheer Aziz, editors, *KI 2009: Advances in Artificial Intelligence*, Lecture Notes in Computer Science, pages 435–443. Springer, 2009.

[Inf10] Infoq.com. The way GitHub helped Erlang and the way Erlang helped Github. www.infoq.com/interviews/erlang-and-github, 2010. [Accesado 21-Ene-2014].

[JBFT05] Bart Jacob, Michael Brown, Kentaro Fukui, y Nihar Trivedi. Introduction to Grid Computing. *IBM Redbooks*, 2005.

[Kes12] Zachary Kessin. *Building Web Applications with Erlang*. O'Reilly Media, 2012.

[Kow92] Robert Kowalski. Database Updates in the Event Calculus. *Journal of Logic Programming*, pages 121–146, 1992.

[KS86] R Kowalski y M Sergot. A logic-based calculus of events. *New Gen. Comput.*, 4(1):67–95, January 1986.

[KS08] A.D. Kshemkalyani y M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, Julio 1978.

[Let09] Eugene Letuchy. Erlang at Facebook. Facebook, 2009.

[LMC11] Martin Logan, Eric Merritt, y Richard Carlsson. *Erlang and OTP in Action*. Manning, 2011.

[Lou97] Kenneth C. Louden. *Compiler construction: principles and practice*. Computer Science Series. PWS-KENT Publishing Company, 1997.

[Mar03] Robert Cecil Martin. *UML for Java programmers*. Prentice Hall, 2003.

[MH69] John Mccarthy y Patrick J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. En *Machine Intelligence*, pages 463–502. Edinburgh University Press, 1969.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The Pi-calculus*. Cambridge University Press, 1999.

[MM82] Alberto Martelli y Ugo Montanari. An Efficient Unification Algorithm. *TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS (TOPLAS)*, 4(2):258–282, 1982.

[MM03] J. Miller y J. Mukerji. Mda guide version 1.0.1. Reporte Técnico, Object Management Group (OMG), 2003.

[MS02] Rob Miller y Murray Shanahan. Some alternative formulations of the event calculus. En *Computer Science; Computational Logic; Logic programming and Beyond*, pages 452–490. Springer Verlag, 2002.

[Mue08] Erik Mueller. Event calculus. En Frank van Harmelen, Vladimir Lifschitz, y Bruce Porter, editors, *Handbook of Knowledge Representation*, pages 671–708. Elsevier, 2008.

[NCdW97] S.H. Nienhuys-Cheng y R. de Wolf. *Foundations of Inductive Logic Programming*. Lecture Notes in Artificial Intelligence. U.S. Government Printing Office, 1997.

[NR05] Thomas Noll y Chanchal Kumar Roy. Modeling Erlang in the Pi-calculus. En *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, ERLANG '05, pages 72–77. ACM, 2005.

[OMG11] OMG. OMG Unified Modeling Language (OMG UML), Infrastructure, V2.4.1. Reporte Técnico, Object Management Group, 2011.

[ÖV11] Tamer Özsu y Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, 2011.

[Pre09] Roger Pressman. *Software engineering: a practitioner's approach*. Mc Graw Hill, 7 edition, 2009.

[RNRC06] Chanchal Kumar Roy, Thomas Noll, Banani Roy, y James R. Cordy. Towards Automatic Verification of Erlang Programs by Pi-Calculus Translation. En *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, pages 38–50. ACM, 2006.

[RR06] Suzanne Robertson y James Robertson. *Mastering the Requirements Process*. Addison-Wesley, 2 edition, 2006.

[SFS09] Issam Souilah, Adrian Francalanza, y Vladimiro Sassone. A formal model of provenance in distributed systems. En James Cheney, editor, *Workshop on the Theory and Practice of Provenance*. USENIX, 2009.

[SGG12] Abraham Silberschatz, Peter B. Galvin, y Greg Gagne. *Operating System Concepts*. Wiley, 9 edition, 2012.

[Sha00] Murray Shanahan. An abductive event calculus planner. *Journal of Logic Programming*, 44:207–239, 2000.

[Sky05] Lars Skyttner. *General Systems Theory: Problems, Perspectives, Practice*. World Scientific, 2005.

[SM02] S. Stelting y O. Maassen. *Applied Java Patterns*. Sun Microsystems Press, 2002.

[Sol12] Erlang Solutions. Erlang- powered WhatsApp exceeds 200 million monthly users. www.erlang-solutions.com/about/news/erlang-powered-whatsapp-exceeds-200-million-monthly-users, 2012. [Accesado 10-Ene-2014].

[Som11] Ian Sommerville. *Software Engineering*. Pearson Education, 9 edition, 2011.

[SS94] Mukesh Singhal y Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, Inc., 1994.

[TvS07] A.S. Tanenbaum y M. van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2 edition, 2007.

[Vin11] Steve Vinoski. Yaws: Yet Another Web Server. *IEEE Internet Computing*, pages 90–94, 2011.

Índice alfabético

agente, 11
álgebra de procesos, 37
aplicaciones de sistemas distribuidos, 30
aridad del functor, 13
arquitectura de un sistema distribuido, 32
arquitecturas de redes, 30
arquitecturas de sistemas distribuidos, 31
  Cliente/Servidor, 32
  Peer-to-peer, 33
átomo, 13
autonomía de sistemas distribuidos, 30

cálculo de cómputo, 27
cálculo de eventos, 11, 16
cálculo de eventos clásico, 17
cálculo de eventos simplificado, 18
cálculo de situaciones, 16
cómputo distribuido geográficamente, 27
cabeza de la cláusula, 13
cláusula de Horn, 12
cláusulas, 12
cliente, 32
compartir recursos, 27
comunicación asíncrona, 35, 46
comunicación entre procesos, 28
  modelos de comunicación, 35
  paradigmas de comunicación, 44
comunicación síncrona, 35, 46
conclusiones, 100
concurrencia, 30, 36, 41, 42
contribución de la tesis, 9
cuerpo de la cláusula, 13

diagrama espacio-tiempo, 36
distribución física, 28
distribución lógica, 28
distribuido, 28

Erlang, 7, 8, 37, 39
  aplicaciones, 40
  características, 40
  elementos básicos, 105
  función, 62
  historia, 39
  módulo, 62
  paso de mensajes, 42, 46
  unidad de concurrencia, 42
especificación de requisitos, 37
especificación de sistemas distribuidos, 37
especificación del software, 3
eventos, 16, 36
  actuate, 64, 78
  asignación a eventos, 64
  end, 64
  receive, 64, 82
  send, 64, 81

fórmulas lógicas, 12
fluentes, 16, 22

grid computing, 30

heterogeneidad de sistemas distribuidos, 30

ingeniería de requisitos, 3
ingeniería de software, 2
instantes, 20

lógica formal, 16
lógica temporal, 16
literal, 13
lock, 44

memoria disjunta, 28, 30
meta colectiva, 28
modelado, 5
modelo de Herbrand, 13

narrativa, 16
nodos, 32
notaciones, 5

objetos, 32

paradigmas de comunicación, 44
- exclusión mutua, 44
- lock, 44
- memoria compartida, 44
- memoria transaccional, 45
- monitores, 44
- paso de mensajes, 46
- semáforos, 44

paso de mensajes, 29, 36, 42, 46
peers, 33
predicados, 17, 77
- happens, 63
- holdsat, 63
- initiates, 63
- terminates, 63

proceso, 32, 39, 42
- aislamiento, 42
- buzón, 42
- distribución transparente, 43
- emisor, 46
- encapsulación, 43
- paso de mensajes, 42
- programación en Erlang, 46
- receptor, 46

procesos múltiples, 28
programa lógico, 13
programación concurrente, 8, 39
programación lógica, 12
ProME, 61, 75
- bases de la notación, 61
- caso de estudio, 89
- elementos de la notación, 75
- gramática, 84
- metodología, 85
- narrativa, 83
- notación, 63
- sintaxis, 78

reloj físico común, 29
respuesta del servidor, 33
resultados de la investigación, 99

separación geográfica, 30
servidor, 32
sistema, 28
sistemas distribuidos, 29, 41
- entidades, 32

solicitud al servidor, 33

término, 13
threads, 32, 42
tolerancia a fallas, 28, 41
trabajo futuro, 101

UML, 5