COMPROBACIÓN DE MODELOS EN SISTEMAS CONCURRENTES A PARTIR DE SU SEMÁNTICA EN MAUDE

GORKA SUÁREZ GARCÍA

MÁSTER EN INGENIERÍA EN INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Máster en Ingeniería en Informática

27/02/2017

Calificación: 6

Director:

Adrián Riesco Rodríguez

Autorización de difusión

Gorka Suárez García

27/02/2017

El abajo firmante, matriculado en el Máster en Ingeniería en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "COMPROBACIÓN DE MODELOS EN SISTEMAS CONCURRENTES A PARTIR DE SU SEMÁNTICA EN MAUDE", realizado durante el curso académico 2015-2016 bajo la dirección de Adrián Riesco Rodríguez en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

La comprobación de modelos (*model checking*) es una técnica automática para verificar si una propiedad se cumple en un sistema concurrente. Maude es un marco lógico de alto rendimiento donde se puede especificar, modelar, ejecutar y analizar —de forma sencilla— otros sistemas. Además, este entorno incluye un comprobador de modelos para verificar propiedades expresadas en lógica temporal lineal. Sin embargo, cuando una propiedad aplicada a un programa —escrito en un lenguaje de programación modelado para Maude— no se cumple, el contraejemplo —generado por el propio sistema— está basado en la semántica del propio Maude, dificultando la tarea de poder seguirlo a la hora de entender el resultado.

En esta memoria presentamos la herramienta Selene, un marco genérico que maneja sistemas concurrentes asíncronos de modo que el usuario pueda obtener una versión simplificada de los contraejemplos generados por el comprobador de modelos en Maude tras la realización del análisis sobre programas escritos en otros lenguajes. Para lograrlo se ofrece un kernel para manejar la memoria y los mensajes, elementos que se emplearán en el "informe" final obtenido del contraejemplo. Sobre dicha arquitectura el usuario podrá especificar los detalles de la semántica del lenguaje a manejar.

Por último, se analizará cuáles fueron los objetivos iniciales, los resultados obtenidos, los problemas encontrados durante el desarrollo, así como las propuestas y líneas futuras de trabajo que serían deseables para la mejora del proyecto.

Palabras clave

Maude, comprobación de modelos, semántica formal, Erlang, JSON.

Abstract

Model checking is an automatic technique for verifying whether some properties hold in a concurrent system. Maude is a high-performance logical framework where other systems can be easily specified, executed, and analyzed. Moreover, Maude includes a model checker for checking properties expressed in Linear Temporal Logic. However, when a property on a program written in a programming language specified in Maude does not hold the counterexample generated by this system refers to the Maude semantics, which might be difficult to follow.

In this Master's Thesis we present Selene, a generic framework for dealing with asynchronous concurrent systems that allows users to receive a simplified version of the counterexample generated by the Maude model checker to relate it to the program being analyzed. This is achieved by providing a kernel for dealing with messages and memory, which are later handled in the counterexample; the user can specify the details of his semantics on top of this kernel.

Finally, it will be analyzed which were the initial objectives, the final results, the problems found in the development, as well as the proposals and future lines of work in the project to improve it.

Keywords

Maude, model checking, formal semantics, Erlang, JSON.

Índice general

Índice de figuras III

1. Introducción 1

2. Introduction 3

3. Objetivos 5

4. Estado del arte 7

4.1. Especificar lenguajes de programación . . . 7
4.2. El sistema Maude . . . 11
4.3. Especificar lenguajes en Maude . . . 15
4.4. Comprobación de modelos en Maude . . . 17
4.5. Comprobadores de modelos para lenguajes . . . 22

5. Selene 26

5.1. Pasos iniciales . . . 26
5.2. Proyectos, fuentes y líneas . . . 31
5.3. Las referencias . . . 36
5.4. Los valores . . . 37
5.5. Las expresiones . . . 39
5.6. La memoria . . . 42
5.7. El mundo . . . 45
5.7.1. El objeto `Project` . . . 46
5.7.2. El objeto `Config` . . . 46
5.7.3. El objeto `Status` . . . 47
5.7.4. El objeto `Node` . . . 47
5.7.5. El objeto `Process` . . . 48
5.7.6. Los mensajes . . . 48
5.8. La máquina . . . 49
5.8.1. El núcleo de ejecución . . . 49

5.8.2. El estado de ejecución 51
5.8.3. Las expresiones temporales 52
5.9. El motor de ejecución de Erlang 52
5.10. Comprobación de modelos en Selene 62
5.10.1. Por un puñado de propiedades 63
5.10.2. La transformación del contraejemplo 64
5.10.3. La representación visual del contraejemplo 70

6. Trabajo futuro **71**

7. Conclusiones **73**

8. Conclusions **76**

Bibliografía **89**

Índice de figuras

4.1. Órdenes condicionales usando semánticas denotacionales. 9
4.2. Órdenes condicionales usando semánticas estructuradas. 10
4.3. Órdenes condicionales usando semánticas axiomáticas. 11
4.4. Los naturales con la notación de Peano y el operador suma. 13
4.5. La reducción de una suma usando el módulo de los naturales. 14
4.6. Un módulo de sistema que usa los naturales para una cuenta atrás. 14
4.7. La reescritura de un término usando el módulo de la cuenta atrás. 15
4.8. Un módulo de sistema con propiedades para comprobar el modelo. 19
4.9. Un módulo de sistema con un estado inicial para comprobar el modelo. . . . 20
4.10. Comprobando si la cuenta atrás eventualmente termina. 20
4.11. Comprobando si la cuenta atrás siempre termina eventualmente. 20
4.12. Comprobando si la cuenta atrás siempre no termina. 20

5.1. Un ejemplo de aplicación escrita en el lenguaje Erlang. 27
5.2. La jerarquía de tipos en Selene, parte 1. 29
5.3. La jerarquía de tipos en Selene, parte 2. 30
5.4. Usando las listas del preludio de Maude. 31
5.5. Código usando el tipo `InputFile`. 32
5.6. Salida por consola del código usando el tipo `InputFile`. 32
5.7. Código usando el tipo `SourceFiles`. 32
5.8. Salida por consola del código usando el tipo `SourceFiles`. 33
5.9. Código usando el tipo `ProjectObject`. 33
5.10. Salida por consola del código usando el tipo `ProjectObject`. 33
5.11. Código usando los índices de línea. 34
5.12. Salida por consola del código usando los índices de línea. 34
5.13. Código usando el tipo `SourceLines`. 35
5.14. Salida por consola del código usando el tipo `SourceLines`. 35
5.15. Las estructuras en el lenguaje genérico de árboles sintácticos. 39
5.16. El ejemplo "hola mundo" en Erlang transformado al LGAS. 42
5.17. El ejemplo "servidor echo" en Erlang transformado al LGAS. 43

5.18. El diagrama de estados de las sentencias. 53
5.19. Salida por consola de la reescritura del "servidor echo", parte 1. 54
5.20. Salida por consola de la reescritura del "servidor echo", parte 2. 55
5.21. Código de la regla `world.init`. 56
5.22. Código de la regla `statement.init`. 57
5.23. Código de la regla `statement.work`. 57
5.24. Código de la regla `statement.exec`. 58
5.25. Código de la regla `statement.afcl`. 58
5.26. Código de la regla `statement.afam`. 59
5.27. Código de la regla `statement.error`. 59
5.28. Código de la regla `statement.next`. 60
5.29. Código de la regla `ambit.finish`. 60
5.30. Código de la regla `call.finish`. 61
5.31. Código de la regla `process.finish`. 61
5.32. Código de la regla `process.msg.add`. 61
5.33. Código de la regla `process.msg.get`. 62
5.34. Resultado del comprobador de modelos con el "servidor echo". 65
5.35. Resultado transformado del comprobador de modelos con el "servidor echo". . 66
5.36. Algoritmo que sigue la operación `modelCheckTransformer`. 69

Agradecimientos

Quiero dar mi agradecimiento a mi tutor el Prof. Adrián Riesco Rodríguez de la Facultad de Informática de la Universidad Complutense de Madrid, así como a mi familia y mis amigos.

Capítulo 1

Introducción

Depurar aplicaciones es una tarea compleja y más si añadimos a la ecuación la programación concurrente. Por ello, a lo largo de los últimos años se han desarrollado herramientas para facilitar la depuración y el análisis de código fuente.

Uno de los campos de análisis es la *verificación formal*[60], dentro de la cual se encuentra la *comprobación de modelos*[16], una técnica automática para verificar si una propiedad —expresada en lógica modal— se cumple en un sistema concurrente.

En 1977, Amir Pnueli propuso la *Lógica Lineal Temporal*[76] (LTL por sus siglas en inglés), una variante de lógica modal para alcanzar la verificación formal mediante comprobación de modelos. En LTL se pueden definir propiedades atómicas sobre los estados de una aplicación y a partir de estas escribir fórmulas temporales con las que realizar las comprobaciones del modelo.

Desarrollar comprobadores de modelos fiables y eficientes no es una tarea trivial. Debido a ello, existen pocas herramientas de este tipo, lo que lleva a tener que realizar traducciones entre lenguajes para permitir su uso. Estas traducciones requieren demostraciones formales para validar su fiabilidad. Adicionalmente, los contraejemplos obtenidos son difíciles de entender[35].

Entre los comprobadores de modelos existentes se encuentra el implementado para el sistema *Maude*[19], un marco lógico de alto rendimiento donde se pueden especificar, modelar, ejecutar y analizar otros sistemas. Maude nos permite definir la sintaxis y la semántica de

otros lenguajes de programación, lo que nos lleva a poder utilizar su comprobador de modelos sobre los mismos.

El objetivo de este proyecto es estudiar el uso de Maude para aplicar comprobación de modelos sobre lenguajes de programación concurrentes. Para ello se ha elegido como candidato *Erlang*[1], un lenguaje funcional concurrente de uso general que se comunica mediante paso de mensajes.

El resto del documento está estructurado de la siguiente manera:

- En el capítulo 3 hablaremos de los objetivos iniciales del proyecto.
- En el capítulo 4 hablaremos de los métodos disponibles para especificar lenguajes, como Maude. A continuación, hablaremos sobre el estado del arte en la comprobación de modelos sobre lenguajes de programación.
- En el capítulo 5 hablaremos de las ideas que hay detrás del estado actual del proyecto. También hablaremos de los problemas que hemos encontrado durante el desarrollo, así como las decisiones que se tomaron.
- En el capítulo 6 hablaremos de las posibles líneas de trabajo futuro para el proyecto.
- Finalmente, en el capítulo 7 y 8 compartiremos unas conclusiones finales sobre el proyecto, tanto en castellano como en inglés.

Capítulo 2

Introduction

Debugging applications is a complex task, even more if we add concurrent programming into the equation. For this reason, in the last few years, tools have been developed to ease debugging and the analysis of source code.

An area of analysis is *formal verification*[60] and inside it we find *model checking*[16], an automatic technique for checking whether some properties—expressed in modal logic—hold in a model of a concurrent system.

In 1977, Amir Pnueli proposed the *Linear Temporal Logic*[76], a type of modal logic to achieve formal verification with model checking. In LTL we can define atomic properties over the states of an application and with them write temporal formulae to perform model checking.

Developing an efficient and reliable model checker is not trivial. So the number of model checkers that exists is low and hence translations are usually required. These translations require to formally prove their correctness. Moreover, the obtained counterexamples are difficult to understand[35].

Among the current existing model checkers we have the one developed for *Maude*[19], a high-performance logical framework where other systems can be specified, modeled, executed, and analyzed. In Maude the syntax and semantics of other systems and logics can be defined; hence, we can use the model checker for analyzing programs for those systems.

The aim of this project is to study how to use Maude for model checking other con-

current programing languages. To do so, the chosen language candidate was *Erlang*[1], a general-purpose concurrent functional programming language that uses asynchronous message passing as communication.

The rest of the document is structured as follows:

- In Chapter 3 we will talk about the initial objectives of the project.
- In Chapter 4 we will talk about some of the tools available for specifying programming languages, like Maude. Then, we will talk in second place about the current state of the art of the model checking tools of programming languages.
- In Chapter 5 we will talk about the ideas behind the current state of the project. We will also discuss about the problems encountered during the development and the choices that were taken.
- In Chapter 6 we will discuss about the possible lines of future work in the Selene project.
- Finally, in Chapter 7 (in Spanish) and 8 (in English) we will share the final conclusions of the project.

Capítulo 3

Objetivos

Inicialmente el nombre del proyecto era “Comprobación de modelos en sistemas concurrentes parametrizada por la semántica en Maude”. Esto implica implementar *semánticas operacionales estructuradas*[75] (SOS) mediante *lógica de reescritura*[94] y usar dichas semánticas como parámetro del sistema a construir. Este enfoque permitiría aplicar comprobación de modelos sobre cualquier lenguaje definido mediante SOS.

Adicionalmente, existía la necesidad de encontrar un medio de expresar los contraejemplos en términos similares a los del lenguaje origen de la aplicación a analizar, ya que por defecto los contraejemplos que obtenemos están expresados con la semántica de Maude.

Tras un breve análisis, se fijaron los siguientes objetivos:

1. Desarrollar una herramienta que tome el código fuente de un proyecto, una fórmula expresada en LTL y genere un contraejemplo con información útil para el desarrollador.

2. Desarrollar una herramienta que tome el contraejemplo obtenido y muestre al usuario la ejecución del proyecto, de modo que el desarrollador pueda inspeccionar aquella información relevante a la ejecución de forma visual.

El plan de trabajo para desarrollar la herramienta del punto 1, que utilizara la semántica de Erlang fue:

- Implementar la representación de la sintaxis del lenguaje.

- Implementar el funcionamiento de la semántica del lenguaje.
- Implementar las propiedades que serán usadas en la LTL.
- Implementar un algoritmo de transformación para el contraejemplo generado por el comprobador de modelos.

El plan de trabajo para desarrollar la herramienta del punto 2, incluía desarrollar una herramienta gráfica en Erlang usando *wxWidgets*[100] mediante el módulo `wx`[26]. Pero la idea terminó por ser descartada debido a la gran complejidad que la idea central del proyecto conformaba ya de por sí.

Al no llegar a desarrollarse la parametrización que se deseaba en primera instancia, se cambió el título del trabajo por el actual.

Capítulo 4

Estado del arte

4.1. Especificar lenguajes de programación

La especificación de un lenguaje se trata de aquella documentación que lo define, siendo una guía —que permita comprender el significado de cualquier programa escrito en dicho lenguaje— para los usuarios y los implementadores del mismo.

Existen varias formas para *especificar un lenguaje de programación*[59]. Por un lado están las formas explícitas de definir la sintaxis y la semántica de un lenguaje. Otra forma es la que va implícita a las descripciones del comportamiento del compilador.

Para nuestro proyecto nos centraremos en las formas explícitas de especificar un lenguaje. La sintaxis habitualmente se define mediante una *gramática formal*[14], como por ejemplo la *Forma Backus–Naur*[62] (BNF) y sus *variantes*[53], que se tratan de notaciones para *gramáticas libres de contexto*[15].

En cuanto a las semánticas tenemos dos principales opciones para definirlas:

1. Usar el *lenguaje natural* como es el caso de *C++*[56,58], *C*[57] o *JavaScript*[23]. Aunque es una opción bastante común, conlleva el problema de no tener necesariamente siempre una única posible interpretación, como fue el caso de los *hilos en Java*[77].

2. Usar *semánticas formales* mediante el uso de las matemáticas para definir el comportamiento de las semánticas del lenguaje en cuestión. Tenemos como ejemplo de ello los lenguajes *ML*[67] y *Scheme*[61]. Esta forma es más precisa y menos ambigua que el

lenguaje natural. Pero al ser menos popular algunos lenguajes incluyen ambas formas, como es el caso de *Modula-2* [52,54,55].

Para nuestro caso de estudio usaremos semánticas formales, pues —al estar fundamentadas en las matemáticas— nos permiten lo siguiente:

- Verificar de forma matemática la corrección de programas.
- Facilitar el diseño de tipos y demostrar la fiabilidad de los mismos.
- Sentar las bases de un estándar no-ambiguo y uniforme para la implementación.

Dado que nuestro interés se encuentra en la verificación formal, el primer punto —verificar la corrección de programas— hace que resulten más interesante las semánticas formales que el uso del lenguaje natural para nuestro caso.

Dentro de las semánticas formales nos encontramos tres enfoques distintos:

- Las *semánticas denotacionales* [72,78,89,99] son un enfoque que formaliza el significado de los lenguajes mediante el uso de *denotaciones*. El objetivo de este enfoque es encontrar un dominio para representar aquello que hace un *programa* o una *frase* del mismo. Este objetivo puede llevar a que los programas y frases sean representados por funciones parciales. Es importante en este enfoque que las semánticas sean composicionales, que significa que la *denotación* de una *frase* se construye con las *denotaciones* de sus *subfrases*.
- Las *semánticas estructuradas* [40,41,72,75] son un enfoque usado para verificar ciertas propiedades de un *programa*, mediante la construcción de demostraciones a partir de declaraciones lógicas sobre su ejecución. Este enfoque se divide en dos categorías:
    1. *Semánticas operacionales estructuradas* o *semánticas de paso corto*, usadas para describir formalmente cada paso individual en la ejecución de una *frase* de un programa.

2. *Semánticas naturales* o *semánticas de paso largo*, usadas para describir de forma general el resultado obtenido de la ejecución de una *frase* de un programa.

- Las *semánticas axiomáticas* [72,92] son un enfoque basado en la lógica matemática para dotar a los *programas* de corrección. Este enfoque está muy relacionado con la *lógica de Hoare* [43].

Podemos encontrar un ejemplo para cada enfoque en las figuras 4.1, 4.2 y 4.3, que representan la misma declaración condicional. Primero se muestra la sintaxis genérica de la orden `if` y después un *programa* de ejemplo usando la orden en sí; donde comprobamos que si la variable `x` es distinta de cero restaremos una unidad dicha variable, si no añadiremos `x` a la variable `y`.

$$[\![\texttt{if } \mathrm{b} \texttt{ then } \mathrm{c}_0 \texttt{ else } \mathrm{c}_1]\!]_{comm}\sigma = \texttt{if } [\![\mathrm{b}]\!]_{boolexp}\sigma \texttt{ then } [\![\mathrm{c}_0]\!]_{comm}\sigma \texttt{ else } [\![\mathrm{c}_1]\!]_{comm}\sigma$$

$$\begin{aligned} & [\![\texttt{if } \mathsf{x} \neq 0 \texttt{ then } \mathsf{x} := \mathsf{x} - 1 \texttt{ else } \mathsf{y} := \mathsf{y} + \mathsf{x}]\!]_{comm}\sigma \\ & \quad = \texttt{if } [\![\mathsf{x} \neq 0]\!]_{boolexp}\sigma \texttt{ then } [\![\mathsf{x} := \mathsf{x} - 1]\!]_{comm}\sigma \texttt{ else } [\![\mathsf{y} := \mathsf{y} + \mathsf{x}]\!]_{comm}\sigma \\ & \quad = \texttt{if } \sigma\mathsf{x} \neq 0 \texttt{ then } [\sigma|\mathsf{x} : \sigma\mathsf{x} - 1] \texttt{ else } [\sigma|\mathsf{y} : \sigma\mathsf{y} + \sigma\mathsf{x}] \end{aligned}$$

Figura 4.1: *Órdenes condicionales usando semánticas denotacionales.*

El ejemplo denotacional de la figura 4.1 fue tomado del libro “Theories of Programming Languages” [78] de Reynolds, que está centrado en el estudio de los lenguajes mediante el uso de esta técnica. En este podemos ver que la semántica de la expresión `if` elige una rama u otra dependiendo de la semántica de su condición. A su vez, el resultado de cada rama de la expresión depende de la semántica de cada subexpresión. Para que la expresión tenga algún valor final debe de haber un contexto o estado que es representado con σ. Este estado es un entorno que relacionará variables con valores, pues del mismo sacaremos los valores asociados a las variables que encontremos en nuestras expresiones. El resultado final de

nuestro programa será un estado final que hemos ido obteniendo con las modificaciones que las expresiones del programa han ido ejerciendo. Como se puede ver en la figura, en la rama verdadera del `if` tenemos que el estado resultante es $[\sigma|\mathsf{x} : \sigma\mathsf{x} - 1]$, esto significa que x será modificada en el entorno tomando el valor previo de x y restando a ese valor una unidad.

$$\frac{\langle \mathrm{b}, S\rangle \Rightarrow \top}{\langle \texttt{if}\ \mathrm{b}\ \texttt{then}\ \mathrm{c}_0\ \texttt{else}\ \mathrm{c}_1, S\rangle \rightarrow \langle \mathrm{c}_0, S\rangle} \qquad \frac{\langle \mathrm{b}, S\rangle \Rightarrow \bot}{\langle \texttt{if}\ \mathrm{b}\ \texttt{then}\ \mathrm{c}_0\ \texttt{else}\ \mathrm{c}_1, S\rangle \rightarrow \langle \mathrm{c}_1, S\rangle}$$

$$\frac{\langle \mathsf{x} \neq \mathsf{0}, S\rangle \Rightarrow \top}{\langle \texttt{if}\ \mathsf{x} \neq \mathsf{0}\ \texttt{then}\ \mathsf{x} := \mathsf{x} - \mathsf{1}\ \texttt{else}\ \mathsf{y} := \mathsf{y} + \mathsf{x}, S\rangle \rightarrow \langle \mathsf{x} := \mathsf{x} - \mathsf{1}, S\rangle}$$

$$\frac{\langle \mathsf{x} \neq \mathsf{0}, S\rangle \Rightarrow \bot}{\langle \texttt{if}\ \mathsf{x} \neq \mathsf{0}\ \texttt{then}\ \mathsf{x} := \mathsf{x} - \mathsf{1}\ \texttt{else}\ \mathsf{y} := \mathsf{y} + \mathsf{x}, S\rangle \rightarrow \langle \mathsf{y} := \mathsf{y} + \mathsf{x}, S\rangle}$$

Figura 4.2: *Órdenes condicionales usando semánticas estructuradas.*

Mientras que las semánticas denotacionales siguen un enfoque de tomar el programa completo hasta reducirlo a un estado final, las operacionales se centran en los pasos que dan —largos o cortos— hasta ejecutar el programa por completo, dando un estado final S. En esta notación el equivalente al entorno recae sobre S, representado por σ en la semántica denotacional. En el ejemplo de la figura 4.2 tenemos dos reglas de transición, que dependiendo del valor que obtengamos ejecutando la expresión booleana b, si es cierto se realizará una transición en la que ejecutar la rama c_0, si es falso se ejecutaría la rama c_1. En la notación utilizamos la flecha $\Rightarrow$ para indicar que estamos evaluando una expresión para obtener su valor final y $\rightarrow$ para mostrar las transiciones entre los pasos de ejecución del programa.

Las semánticas axiomáticas se centran en verificar si las precondiciones y las poscondiciones se cumplen para probar la corrección del programa. En ocasiones podemos vernos en la necesidad de comprobar que un programa tenga ciertas invariantes, para realizar demostraciones. El estado que nos da la información, sobre las variables de nuestro programa,

$$\frac{\{P \wedge \mathrm{b}\}\ \mathrm{c}_0\ \{Q\}, \{P \wedge \neg \mathrm{b}\}\ \mathrm{c}_1\ \{Q\}}{\{P\}\ \texttt{if}\ \mathrm{b}\ \texttt{then}\ \mathrm{c}_0\ \texttt{else}\ \mathrm{c}_1\ \{Q\}}$$

$$\frac{\{P \wedge (\mathsf{x} \neq 0)\}\ \mathsf{x} := \mathsf{x} - 1\ \{Q\}, \{P \wedge \neg(\mathsf{x} \neq 0)\}\ \mathsf{y} := \mathsf{y} + \mathsf{x}\ \{Q\}}{\{P\}\ \texttt{if}\ \mathsf{x} \neq 0\ \texttt{then}\ \mathsf{x} := \mathsf{x} - 1\ \texttt{else}\ \mathsf{y} := \mathsf{y} + \mathsf{x}\ \{Q\}}$$

Figura 4.3: *Órdenes condicionales usando semánticas axiomáticas.*

lo encontraremos en las precondiciones y las poscondiciones. Por ejemplo, si creamos una nueva variable con un valor, esta aparecerá en la poscondición.

Debido a las investigaciones realizadas con Maude en este campo, el enfoque usado con el proyecto es el de las semánticas operacionales.

4.2. El sistema Maude

Maude [18,19,50] es un marco lógico de alto rendimiento. Permite el uso de lógica ecuacional de pertenencia y de reescritura. Una de las influencias principales de este lenguaje es *OBJ3* [32], que proviene de la familia OBJ [33] y se puede considerar como un sublenguaje de lógica ecuacional.

La *reescritura* [3] —en matemáticas, informática y lógica— cubre un amplio espectro de métodos para substituir subtérminos en una fórmula con otros términos de forma no-determinista. Esta mecánica, de forma resumida, consiste en un conjunto de objetos más las relaciones que definen como se transforman. El no-determinismo permite que una regla se aplique a un término de diferentes formas, así como que más de una regla se pueda aplicar al mismo término.

Por ello la lógica de reescritura nos permite manejar de forma natural la computación concurrente. Además, como marco general semántico, trae consigo propiedades que otorgan

semánticas de ejecución a una amplia selección de lenguajes y modelos de concurrencia.[1] Estos motivos —que hacen de la lógica de reescritura un buen marco semántico— la convierten también en un buen marco lógico, lo que implica una metalógica sobre la que otras lógicas pueden ser representadas y ejecutadas de un modo natural.

Además, la lógica de reescritura es reflexiva, y Maude implementa de manera eficiente esta reflexión a través de su metanivel, lo que permite extender al propio lenguaje. El uso avanzado de metaprogramación es factible gracias a, entre otras cosas, las operaciones que Maude dispone para el algebra composicional de módulos. Todo ello convierte al metanivel en una de las características más interesantes del lenguaje, pues nos permite crear entornos de ejecución para diferentes tipos de lógicas, demostradores de teoremas y otros lenguajes y modelos de computación.

Antes de proseguir, echemos un vistazo a la apariencia que el lenguaje Maude tiene con la figura 4.4. La unidad básica en Maude son los *módulos* y en el ejemplo vemos el uso de un *módulo funcional* declarado con `fmod`. Los módulos funcionales sirven para definir tipos de datos y operaciones aplicables a los mismos mediante el uso de teorías ecuacionales. El contrapunto lo tenemos en los *módulos de sistema*, que son empleados para especificar teorías de reescritura a través de reglas de transición.

En el ejemplo vemos una especificación para los números naturales y la operación de la suma. Para lograrlo primero hay que definir el tipo `Nat` con la palabra reservada `sort`. Una vez declarado el tipo, podemos definir operaciones relacionadas con el mismo usando `op`. Las operaciones sin argumentos también son conocidas como constantes; este es el caso de `zero`, definido como el caso base dentro de los números naturales. Las operaciones con argumentos son utilizadas para representar estructuras complejas de datos o para crear funciones matemáticas que definan las capacidades que podemos expresar para los tipos relacionados con dicha operación. En el ejemplo tenemos a `s_` como la función sucesor —

[1] Hay que tener en cuenta que —entre toda la selección de lenguajes y modelos— la reescritura da un excelente soporte a los modelos de concurrencia orientados a objetos, algo que no es baladí dados los paradigmas de programación que habitualmente se emplean en el día a día de la informática.

```
fmod SIMPLE-NAT is
   sort Nat .
   op zero : -> Nat .
   op s_ : Nat -> Nat .
   op _+_ : Nat Nat -> Nat .
   vars N M : Nat .
   eq zero + N = N .
   eq s N + M = s (N + M) .
endfm
```

Figura 4.4: *Los naturales con la notación de Peano y el operador suma.*

dentro de los axiomas de Peano a la hora de representar los naturales— y a `_+_` como la función suma.[2] Entonces, para dotar de semántica a las operaciones, necesitamos definir ecuaciones con `eq`. En el ejemplo especificamos que `zero` es el elemento identidad de la función suma y que la suma de dos naturales, `s N` y `M`, es el sucesor de la suma del predecesor de `s N` con `M`.[3]

Las conclusiones que podemos sacar del ejemplo son que los operadores nos ayudan a definir la sintaxis de los tipos y las ecuaciones a definir las semánticas de los operadores de modo denotacional (usando otros operadores de forma composicional).[4] Con estas herramientas podemos expresar frases de un tipo determinado y reducir sus términos a *expresiones base*[45] gracias al motor de reescritura del que dispone Maude.[5]

Tomando el módulo anterior, podemos expresar `s s s zero + s s zero` y reducir dicha expresión a `s s s s s zero` (este ejemplo sería equivalente a tener la expresión 3 + 2 y

[2]Dentro de la sintaxis de Maude el guión bajo se utiliza para representar la posición de los argumentos para una operación.

[3]El predecesor de `s N` es el propio `N`.

[4]Las ecuaciones en Maude se utilizan para el encaje de igualdades entre términos. Desde el punto de vista operacional, la ejecución de estas se realiza de izquierda a derecha.

[5]En lógica matemática una *expresión base* de un sistema formal es un *término base* o *fórmula base*. Un *término base*, o *término cerrado*, consiste en aquel que no contiene variables libres en su interior. Una *fórmula base* es aquella fórmula que no contiene variables libres en su interior, lo que conlleva a que los términos contenidos por la fórmula no pueden tener a su vez variables libres en su interior. Dentro de Maude una *expresión base* es aquel término que no contiene variables, solo constantes y operadores, como por ejemplo el caso del término `zero`.

```
Maude> red s s s zero + s s zero .
reduce in SIMPLE-NAT : s s s zero + s s zero .
rewrites: 4 in 6729318537ms cpu (0ms real) (0 rewrites/second)
result Nat: s s s s s zero
```

Figura 4.5: *La reducción de una suma usando el módulo de los naturales.*

```
mod SIMPLE-COUNTDOWN is
   pr SIMPLE-NAT .
   var N : Nat .
   rl s N => N .
endm
```

Figura 4.6: *Un módulo de sistema que usa los naturales para una cuenta atrás.*

reducir la suma al valor 5). La salida por consola en Maude de este ejemplo de reducción está en la figura 4.5.

Pero esto tan solo es un ejemplo mínimo de las capacidades completas de Maude. Con los módulos de sistema podemos definir reglas de transición para nuestro modelo, permitiendo al motor de reescritura darnos un estado final dentro del árbol de búsqueda de reescritura. Para mostrarlo tenemos el ejemplo más simple posible en la figura 4.6.

El módulo en cuestión representa una cuenta atrás de naturales. Primero se incluye el módulo `SIMPLE-NAT` para poder utilizar los naturales que hemos definido anteriormente. A continuación definimos una sola regla de transición con `rl`. Esta regla toma un natural y lo transforma en su predecesor. La salida de la reescritura de este ejemplo se puede ver en la figura 4.7.

Animamos al lector interesado en aprender más sobre Maude a leer su manual[18]. En el mismo se encuentra todos los detalles del lenguaje explicados de manera apropiada y con bastantes buenos ejemplos.

```
Maude> rew s s s s s zero .
rewrite in SIMPLE-COUNTDOWN : s s s s s zero .
rewrites: 5 in 1628036047000ms cpu (0ms real) (0 rewrites/second)
result Nat: zero
```

Figura 4.7: *La reescritura de un término usando el módulo de la cuenta atrás.*

4.3. Especificar lenguajes en Maude

Ahora que ya conocemos el lenguaje Maude, hablaremos sobre el trabajo realizado en el campo de la especificación de lenguajes sobre esta herramienta.

Por un lado podemos definir las semánticas de nuestra sintaxis de forma denotacional usando ecuaciones o podemos hacerlo de forma estructurada usando reglas, ya sean de paso corto o paso largo. En el caso de las semánticas estructuradas de paso largo, Maude permite incorporar el uso de condiciones de reescritura en reglas, para los pasos intermedios especificados en las premisas que necesite nuestro paso actual.

El primer trabajo[91] a tener en cuenta —por parte de Şerbănuţă, Roşu y Meseguer— estudia cómo obtener semánticas operacionales mediante el uso de la lógica de reescritura, dándonos una introducción a las semánticas de paso largo y a las de paso corto. En la sección A.1, de dicho trabajo, el lector puede encontrar un ejemplo de cómo definir reglas de paso largo en Maude.

Otro notable trabajo es el de Verdejo y Martí–Oliet[94], donde se demuestra que las premisas de las reglas de inferencia se expresan como reescrituras en las condiciones. Esto demuestra de nuevo que Maude está preparado para utilizar semánticas estructuradas ya sean de paso largo o de paso corto.[6] Los autores nos muestran varios casos de estudio que cubren diferentes paradigmas de programación. Entre los casos se empieza por mostrar versiones simplificadas de lenguajes, siendo *Fpl* muestra de lenguaje funcional dentro de la

[6]En el trabajo a las semánticas de paso largo también las denominan como semánticas de evaluación y a las de paso corto de computación.

programación declarativa, así como *WhileL* y *GuardL* son muestra de lenguajes de programación imperativa. Otros casos implementado —de lenguajes existentes— son el lenguaje funcional *Mini-ML*[20] de Gilles Kahn, así como los lenguajes de modelado *CCS*[65,66] de Milner y *Full LOTOS*[24,51] (incluyendo la especificación de tipos de datos *ACT ONE*).

En el trabajo de Meseguer y Roşu[64] podemos ver que la lógica de reescritura es un marco lógico que unifica las álgebras de las semánticas denotacionales y las SOS. Esto es posible, ya que los axiomas —de la teoría de la lógica de reescritura— incluyen tanto ecuaciones como reglas de reescritura; permitiendo que las definiciones de semánticas realizadas puedan ser ejecutadas —mediante la interpretación— con lenguajes como Maude, que dispongan de lógica de reescritura. Como resultado de lo anterior, un efecto colateral es que se puede dotar a estos intérpretes con capacidades de análisis de programas, como es el caso de la comprobación de modelos. Para demostrar todo lo anterior se implementó el lenguaje SIMPLE, un lenguaje imperativo con *sintaxis estilo C*[57]. En la sección 3.2 del trabajo, se habla también de otros lenguajes, como es el caso de la herramienta *JavaRL*[28] que permite interpretar una especificación de Java 1.4. JavaRL está desarrollado sobre *JavaFAN*[27], que es una aproximación basada en lógica de reescritura usando Maude; para permitir definiciones formales del lenguaje Java, así como el análisis de programas escritos en Java.

Estos tres trabajos guardan una estrecha relación con el trabajo aquí presentado, ya que el uso de semánticas estructuradas de paso corto ha sido la base sobre la que desarrollar el proyecto Selene. Sin embargo las semánticas de paso largo no fueron utilizadas, ello fue debido a ciertas peculiaridades del lenguaje Erlang con sus expresiones destinadas a recibir mensajes. Para poder lograr el cierre transitivo, de nuestra semántica de paso corto, se utilizó una máquina de estados.

Los trabajos desarrollados por Roşu usando Maude como marco —por ejemplo, enseñar diseño de lenguajes de programación[86]— son notables. En la sección 3.3 del trabajo de Meseguer y Roşu[64] se introduce $\mathbb{K}$, un marco para definición de dominios específicos para lógica de reescritura. El marco semántico $\mathbb{K}$ es explicado en trabajos posteriores de Roşu[87],

centrándose en cómo definir de forma genérica cualquier tipo de lenguaje, para dar soporte al no-determinismo y la concurrencia. También se explica en otros artículos[2,90] cómo el marco $\mathbb{K}$ interactúa con Maude, dado que el propio $\mathbb{K}$ está implementado sobre Maude. Alguna de estas interacciones son compilar las definiciones del lenguaje $\mathbb{K}$ en teorías de reescritura de Maude, para así poder ejecutar programas sobre el motor de reescritura de Maude.

La relación de Selene con los trabajos del párrafo anterior es algo menos estrecha, ya que el marco $\mathbb{K}$ no es utilizado en el proyecto. No obstante el marco $\mathbb{K}$ y Selene comparten la intención de crear una capa intermedia entre Maude y la definición de lenguajes.

La realización de intérpretes es solo una de las aplicaciones posibles a la hora de especificar un lenguaje en Maude. Otros ejemplos de análisis son el uso de Maude para la *depuración declarativa*[82–85], que permite depurar los módulos escritos en Maude, lo que a su vez nos permite realizar depuración declarativa sobre programas o frases usando la especificación de un lenguaje de programación en Maude[79].

Por último, otro ejemplo de análisis de programas es el *slicing estático*[97,98], que se utiliza para obtener un fragmento rebanado de un programa que contenga solo el código que afecte a un valor seleccionado dentro del mismo. Sobre este tema también se ha investigado dentro de Maude[80,81] en los últimos años con éxito.

Estos trabajos son muestras de análisis que nos permite una herramienta, como Maude, una vez tenemos la semántica descrita para ser ejecutada por un motor de reescritura. Selene comparte ese tipo de enfoque para poder realizar análisis sobre programas concurrentes.

4.4. Comprobación de modelos en Maude

La *comprobación de modelos*[16,17] es una técnica automática —dentro de la *verificación formal*[60]— para verificar si propiedades —expresadas en lógica modal— se cumplen dentro del modelo de en un sistema concurrente. Maude dispone de un *comprobador de modelos*[25] basado en LTL como parte del marco que ofrece.

Históricamente, esta metodología estaba centrada en la verificación de hardware median-

te el uso de *máquinas de estados finitas*[49,63] (MEF) como representación de los programas a ejecutar. Sin embargo estos conceptos son aplicables al software también, al ser también representable por una MEF. Las MEF son grafos dirigidos y la ejecución de los mismos puede ser transformada en un árbol con todos los posibles caminos de ejecución. Cada nodo del árbol contiene el estado de la ejecución del programa en ese punto. Esto permite comprobar si alguna vez el programa llega a algún estado donde se dé un error de ejecución.

La *LTL*[76] es una lógica modal cuyas modalidades hacen referencia al tiempo. Podemos codificar fórmulas —que pueden ser satisfechas durante la ejecución de un sistema— con los siguientes operadores lógicos: $\neg$ (negación), $\vee$ (disyunción), $\wedge$ (conjunción), $\rightarrow$ (implicación), $\leftrightarrow$ (si y solo si), *true*, *false*, $\bigcirc$ (siguiente), $\Box$ (global), $\Diamond$ (alguna vez/eventualmente), $\mathcal{U}$ (hasta) y $\mathcal{R}$ (liberación). Los últimos cinco son operadores modales temporales y su significado es el siguiente:

Símbolo	Explicación
$\bigcirc\ \phi$	ϕ ha de ser cierto en el siguiente estado de todas las ramas.
$\Box\ \phi$	ϕ ha de ser cierto siempre en todas las ramas.
$\Diamond\ \phi$	ϕ será cierto en algún momento de todas las ramas.
$\psi\ \mathcal{U}\ \phi$	ψ ha de ser cierto hasta que ϕ sea cierto en todas las ramas.
$\psi\ \mathcal{R}\ \phi$	ψ ha de ser cierto hasta que (incluyendo ese estado) ϕ sea cierto por primera vez; si ϕ nunca es cierto, ψ será cierto para siempre. Esta propiedad debe cumplirse en todas las ramas.

Volviendo al ejemplo de los números naturales —mostrado en las figuras 4.4 y 4.6— vamos a escribir una propiedad para comprobar si se cumple durante la ejecución de nuestro programa. El resultado se puede ver en las figuras 4.8 y 4.9. El módulo `SIMPLE-PROPS` importa a los módulos `SIMPLE-COUNTDOWN` —donde se encuentra la teoría de reescritura definida— y `SATISFACTION`, este último incluye los tipos necesarios en Maude para la definición del estado y de proposiciones atómicas, así como para definir la relación de satisfacción. Lo primero a resolver es qué tipo en nuestro modelo representará el estado de ejecución, para ello hacemos un `subsort` de `Nat` con respecto al tipo `State` (definido en `SATISFACTION` para definir el estado), de este modo le decimos que `Nat` es un subtipo de `State` y así podemos usar los naturales de la lógica temporal de la comprobación de modelos.

```
mod SIMPLE-PROPS is
   pr SATISFACTION .
   pr SIMPLE-COUNTDOWN .
   subsort Nat < State .
   var N : Nat .
   op cdfinished : -> Prop [ctor] .
   eq N |= cdfinished = (N == zero) .
endm
```

Figura 4.8: *Un módulo de sistema con propiedades para comprobar el modelo.*

El siguiente paso es definir la propiedad en sí misma. En el ejemplo se ve que hay una operación y una ecuación para definir una sola propiedad, llamada `cdfinished`, que comprueba si la cuenta atrás ha finalizado. Nótese la aparición de un nuevo elemento de sintaxis de Maude, el atributo `ctor`, que declara las operaciones como *constructores*.[7] Como el propio manual[18] de Maude explica, los constructores se utilizan para definir la forma canónica de las operaciones. La forma canónica es una forma simplificada de término base (un término sin variables). Existen más atributos que pueden ser utilizados con las operaciones, ecuaciones, reglas, etcétera; todos ellos explicados en detalle dentro del manual de Maude.

De vuelta a la propiedad `cdfinished` en sí misma, notaremos que la sintaxis para definirlas tiene sus peculiaridades. El lado izquierdo de la ecuación contiene una variable para representar el estado, que puede satisfacer o no la propiedad definida a la derecha del operador `_|=_`. En el lado derecho de la ecuación nos encontramos una expresión Booleana para comprobar si el estado ha llegado al valor `zero`; si la igualdad devuelve cierto, la propiedad ha sido satisfecha. El motivo de definir las propiedades como constructores se debe a que de este modo las fórmulas atómicas no se pueden reducir usando ecuaciones y por lo tanto están en la forma canónica.

El módulo del ejemplo en la figura 4.9 es mucho más sencillo, incluimos los módulos

[7] El módulo `SIMPLE-NAT` —para ser mejorado— puede ser modificado de modo que los operadores `zero` y `s_` sean constructores. De hecho la propia implementación de los naturales, dentro del preludio de Maude, contiene dichos operadores como constructores.

```
mod SIMPLE-MCTEST is
   pr SIMPLE-PROPS * (sort Nat to SmpNat) .
   pr MODEL-CHECKER .
   pr LTL-SIMPLIFIER .
   op initial : -> SmpNat .
   eq initial = s s s s s zero .
endm
```

Figura 4.9: *Un módulo de sistema con un estado inicial para comprobar el modelo.*

```
Maude> red modelCheck(initial, <> cdfinished) .
reduce in SIMPLE-MCTEST : modelCheck(initial, <> cdfinished) .
rewrites: 34 in 13129332125ms cpu (9ms real) (0 rewrites/second)
result Bool: true
```

Figura 4.10: *Comprobando si la cuenta atrás eventualmente termina.*

```
Maude> red modelCheck(initial, [](<> cdfinished)) .
reduce in SIMPLE-MCTEST : modelCheck(initial, []<> cdfinished) .
rewrites: 39 in 13129332125ms cpu (24ms real) (0 rewrites/second)
result Bool: true
```

Figura 4.11: *Comprobando si la cuenta atrás siempre termina eventualmente.*

```
Maude> red modelCheck(initial, [](~ cdfinished)) .
reduce in SIMPLE-MCTEST : modelCheck(initial, []~ cdfinished) .
rewrites: 25 in 6236687736ms cpu (0ms real) (0 rewrites/second)
result ModelCheckResult: counterexample(
    {s s s s s zero, unlabeled}
    {s s s s zero, unlabeled}
    {s s s zero, unlabeled}
    {s s zero, unlabeled}
    {s zero, unlabeled},
    {zero, deadlock})
```

Figura 4.12: *Comprobando si la cuenta atrás siempre no termina.*

relacionados con nuestro programa (`SIMPLE-PROPS`, que incluye a `SIMPLE-COUNTDOWN`) y aquellos que implementan el comprobador de modelos (`MODEL-CHECKER` y `LTL-SIMPLIFIER`; el primero nos permite usar el comprobador de modelos y el segundo simplifica las fórmulas de LTL bajo ciertas circunstancias, para optimizar el algoritmo de comprobación de modelos). El único detalle especial a comentar es la forma de importar `SIMPLE-PROPS`, renombrando el tipo `Nat` como `SmpNat`. La necesidad de hacer esto se debe a que el preludio de Maude ya tiene definido sus propios números naturales, que son usados por los diferentes módulos de la biblioteca de Maude y por lo tanto entramos en una colisión de nombres. Resuelto este pequeño problema, lo último que queda del módulo es definir una constante como estado inicial de la comprobación en nuestro programa de cuenta atrás.

Teniendo preparadas la especificación de nuestro sistema y el estado inicial del mismo, podemos comenzar a crear las fórmulas que necesitemos comprobar sobre el modelo. Ejemplos de fórmulas y sus resultados se pueden ver en las figuras 4.10, 4.11 y 4.12. El primer ejemplo comprueba si la cuenta atrás llega eventualmente a su final en algún punto de la ejecución. El segundo comprueba que siempre se llegue de forma eventual en algún punto de la ejecución al final de la misma, que dicho de otra manera sería que siempre se llega alguna vez al final de la ejecución. El tercero comprueba que siempre ocurra que no termina la ejecución, ejemplo en el que recibimos un contraejemplo[8] que nos indica qué camino se ha seguido para que la fórmula falle.

Estos ejemplos son una demostración mínima de las características de Maude. En la sección 4.1 del trabajo de Meseguer y Roşu[64], podemos ver el uso de Maude para realizar comprobación de modelos sobre la implementación de la especificación del lenguaje de programación SIMPLE usando lógica de reescritura.

Otro trabajo[71] de ejemplo, implementa en Maude una especificación de *Core Erlang*[11] (una versión reducida y básica del lenguaje Erlang), para luego poder utilizar el comprobador de modelos con dicha especificación. Este trabajo comparte con el nuestro el objetivo de

[8]El contraejemplo ha sido formateado a mano para mostrar de forma más vistosa el camino seguido para que la fórmula fallara.

analizar programas escritos en Erlang, solo que en vez de usar el lenguaje completo se emplea Core Erlang.[9] Lo que no comparte es que no busca generalizar la infraestructura sobre la que interpretar el código de entrada, como tampoco se preocupa de representar el contraejemplo de formas alternativas.

También dentro del trabajo relacionado con Java PathExplorer[37,38] —por parte de Havelund y Roşu— está Maude presente como una de las opciones para poder utilizar comprobación de modelos sobre el lenguaje Java.

4.5. Comprobadores de modelos para lenguajes

Maude no es la única herramienta que trae consigo un comprobador de modelos. Entre los diferentes tipos que existen, nos vamos a centrar sobre todo en aquellos orientados a analizar lenguajes de programación, más que en aquellos que usan lenguajes de modelado. Empezaremos primero con los lenguajes imperativos:

- *BLAST*[6,42] es un comprobador de modelos para el lenguaje C, centrado en comprobar propiedades de comportamiento sobre interfaces. Usa refinamiento de abstracciones automáticas dirigidas por contraejemplos, para construir un modelo abstracto sobre el que hacer comprobación de modelos. Fue finalmente sustituido por CPAchecker.

- *CPAchecker*[5,7–9] es un marco y una herramienta para la verificación formal y el análisis de programas escritos en el lenguaje C. Hereda conceptos de BLAST, como la abstracción perezosa. Realiza análisis de accesibilidad a estados que hayan violado una especificación dada a comprobar. Uno de los usos de esta herramienta es la verificación de los drivers de Linux.

- *REDLIB*[95,96] es una biblioteca para acceder a la herramienta RED, un comprobador de modelos para autómatas temporizados, que está basado en técnicas simbólicas so-

[9] El compilador de Erlang tiene —entre sus opciones— el transformar un módulo a su versión en Core Erlang. Para ello se utiliza en la consola de Erlang el comando: `c(módulo, to_core)`, o se puede por consola usar: `erlc +to_core módulo.erl`.

bre diagramas parecidos a los diagramas de decisión binarios. Se puede controlar la ejecución del modelo mediante una API escrita en el lenguaje C. Además la herramienta permite un lenguaje de modelado extendido con construcciones de C, lo que ayuda a poder convertir los programas escritos en C al modelo usado por REDLIB.

- *ISP* [34,73] es una herramienta para la verificación formal de programas que usan la Interfaz de Paso por Mensajes.[10] Igual que otros comprobadores de modelos, ISP verifica el completo espacio de estados para validar propiedades de seguridad y fiabilidad. Pero a diferencia de otros comprobadores de modelos, ISP realiza verificación a nivel del código, para lograrlo utiliza un algoritmo de reducción de orden parcial dinámico modificado que se llama *POE* [93].

- CHESS [22,68,69] es una herramienta diseñada para encontrar y reproducir, en programas concurrentes, *Heisenbugs* [70,74]. Las pruebas se realizan asegurando que cada ejecución tiene un patrón de intercalación diferente. Si uno de los patrones de intercalación desencadena un error, la herramienta puede reproducir ese patrón para realizar la depuración. CHESS está disponible para lenguajes manejados como C# o lenguajes nativos como C/C++.

- *Java PathFinder* [13,36] es un entorno de ejecución configurable para la verificación de programas en bytecode de la JVM (Java Virtual Machine). Su principal uso es la comprobación de modelos sobre programación concurrente, para encontrar fallos como carreras de acceso a datos, interbloqueos y excepciones no atrapadas.

- MoonWalker [21,88] es una herramienta similar a Java PathFinder, pero que verifica programas en bytecode del CIL (Common Intermediate Language). Su principal uso es detectar interbloqueos y assertos fallidos en programas escritos en C# y compilados con Mono.

[10] MPI (Message Passing Interface) es un estándar y sistema portable de paso por mensajes, diseñado por investigadores de la industria y académicos para trabajar con una amplia variedad de computadoras paralelas. Existen implementaciones de este estándar en C, C++ y FORTRAN.

En cuanto a los lenguajes declarativos, en concreto los funcionales, no se ha podido encontrar más que una sola herramienta. McErlang[29–31] es una herramienta centrada en la comprobación de modelos sobre programas concurrentes, para buscar errores durante la ejecución de dichos programas.[11] Para lograr estos objetivos, reemplaza partes del entorno estándar de ejecución de Erlang —aquellos relacionados con la distribución, la concurrencia y la comunicación— con un nuevo entorno de ejecución que simula los procesos dentro del comprobador de modelos, ofreciendo un acceso sencillo al estado del programa. El usuario no puede definir fórmulas en LTL para el comprobador de modelos de McErlang, pero puede seleccionar los algoritmos disponibles por el entorno, cada uno con una serie de propiedades definidas que se usarán en la comprobación.

El caso de McErlang está relacionado con nuestro proyecto, ya que ambos trabajan con Erlang. Sin embargo, McErlang solo centra el foco en Erlang como lenguaje de entrada y para facilitar su implementación utiliza el propio entorno de ejecución de Erlang, modificando aquellas partes que son objeto de análisis (distribución, concurrencia, comunicación). Por ello la capacidades de comprobación de modelos están restringidas a una serie de algoritmos, en vez de enfocadas a probar fórmulas cualesquiera en LTL.

El tipo de lenguajes donde es más popular el uso de comprobadores de modelos, son aquellos para modelar sistemas como algebras para procesos, redes Petri, etcétera. Ejemplos de estos lenguajes son: *CCS*[65,66] (Calculus of Communicating Systems), *CSP*[44] (Communicating Sequential Processes), *ACP*[4] (Algebra of Communicating Processes) y *LOTOS*[24,51].

Un caso particularmente famoso, dada su larga trayectoria, es el comprobador de modelos *SPIN*[10,46–48]. Creado en los laboratorios Bell durante los años 80, se utiliza para comprobar modelos de aplicaciones multi-hilo, descritas mediante *PROMELA*, el lenguaje propio de esta herramienta.

Aunque también es un tema bastante interesante este tipo de lenguajes de modelado, no están tan relacionados con el objeto de la investigación de este proyecto máster, como sí lo

[11]Se puede encontrar un ejemplo sobre cómo usar McErlang en el siguiente tutorial: https://babel.ls.fi.upm.es/trac/McErlang/attachment/wiki/midTermWorkshop/paper.pdf

están los casos anteriores.

Las conclusiones que podemos sacar de todo esto son:

1. No hay muchas herramientas para la comprobación de modelos que sean realmente conocidas y de los pocos casos populares que hay, se suele tratar de alguna herramienta de modelado de sistemas. Por lo que el desarrollador tiene que —además de programar su aplicación— codificar la especificación del modelo de dicha aplicación, si quiere dar uso de la comprobación de modelos.

2. Las herramientas que sí están centradas en analizar código fuente, escrito en lenguajes de programación, son todavía rudimentarias y por lo general están restringidas a un repertorio de propiedades a comprobar. En nuestro trabajo buscamos una forma de flexibilizar este aspecto utilizando el uso de fórmulas de LTL.

3. Tampoco hay APIs, ni marcos de trabajo, lo suficientemente populares y completamente desarrollados como para facilitar la creación de comprobadores de modelos para lenguajes de programación de forma genérica. En nuestro trabajo buscamos una forma de tener un marco de trabajo con el que no limitarse a tener que reimplementar un comprobador de modelos para cada lenguaje, intentando ofrecer una base general que facilite el uso de varios lenguajes.

Capítulo 5

Selene

5.1. Pasos iniciales

Como hemos explicado previamente, Selene[1] busca ser un marco donde desarrollar las semánticas de lenguajes concurrentes, para utilizar la comprobación de modelos sobre programas que le pasemos. Para ello se empezó tomando Erlang como lenguaje de partida y generalizar a partir de él.

En la figura 5.1 tenemos un ejemplo de aplicación escrita en Erlang, que crea N procesos, a cada uno de ellos se le envía el identificador de proceso del siguiente y luego se le envía un mensaje al primero. Entonces de forma cíclica se van enviando ese mensaje unas M veces y, al llegar a cero ese contador, el proceso termina su ejecución. Es un ejemplo básico para ver cómo se crean procesos (con la función `spawn`), enviar mensajes (con el operador `!`) y recibirlos (con la expresión `receive`). Otra funcionalidad importante del ejemplo es la función `io:format` que permite al programador mandar texto a la consola, obteniendo así una salida de datos por pantalla.

Teniendo en cuenta los elementos básicos para la concurrencia y la comunicación de procesos en Erlang, el objetivo de Selene es el encontrar una forma de generalizar esas funcionalidades para interpretar y ejecutar algoritmos concurrentes con el objetivo de poder analizar dicha ejecución. Pero aun existiendo buenos libros para aprender el lenguaje Erlang,

[1] El código fuente de la última versión de la herramienta se puede encontrar en: https://github.com/gorkinovich/Selene

```
-module(dalek).
-author("Gorka Suárez García").
-export([start/3, dalek_start/2, dalek_fight/3]).

start(0, _, _) -> ok;
start(_, 0, _) -> ok;
start(M, N, Msg) ->
  Davros = self(),
  Daleks1 = [spawn(?MODULE, dalek_start, [Davros, M]) || _ <- lists:seq(1, N)],
  Daleks2 = tl(Daleks1) ++ [hd(Daleks1)],
  SkaroDaleks = lists:zip(Daleks1, Daleks2),
  [Dalek ! {Davros, NextDalek} || {Dalek, NextDalek} <- SkaroDaleks],
  hd(Daleks1) ! {Davros, hd(Daleks1), Msg}.

dalek_start(Davros, Shots) ->
  receive {Davros, Victim} ->
    dalek_fight(Davros, Victim, Shots)
  end.

dalek_fight(Davros, Victim, Shots) ->
  ThisDalek = self(),
  Ammo = receive
    {Davros, ThisDalek, Msg} ->
      io:format("Davros called you to fight for the glory of Skaro!~n"),
      io:format("[~p] Dalek ~w received: ~p~n", [Shots, ThisDalek, Msg]),
      Msg;
    {OtherDalek, ThisDalek, Msg} when is_pid(OtherDalek) ->
      io:format("[~p] Dalek ~w received from ~w this message: ~p~n",
                [Shots, ThisDalek, OtherDalek, Msg]),
      Msg
  end,
  Victim ! {ThisDalek, Victim, Ammo},
  if Shots > 1 -> dalek_fight(Davros, Victim, Shots - 1);
     true      -> auto_exterminate
  end.
```

Figura 5.1: *Un ejemplo de aplicación escrita en el lenguaje Erlang.*

como el "Learn You Some Erlang for Great Good!"[39], la especificación oficial del lenguaje como tal no está disponible. De hecho Erlang traduce su código a una versión simplificada llamada Core Erlang, de la que sí existe una especificación[12] como tal. Esta limitación

lleva a tener que ir sacando el comportamiento de Erlang un poco por ensayo y error, dando prioridad a aquellos elementos que consideramos fundamentales para nuestro trabajo (concurrencia y comunicación).

En un proyecto anterior del autor —a la hora de especificar la sintaxis del lenguaje a implementar— se optó por hacer un sistema de tipos altamente complejo, que limitara las frases expresables en Maude únicamente a las que eran sintácticamente correctas. La idea era evitar cosas como sumar un Booleano a un número, por ejemplo. La idea fue descartada como opción para el proyecto Selene, pues añadía una complejidad innecesaria, implementando una característica que no hacía falta, ya que partiremos de la base de que los programas que se van a analizar son programas listos para ejecutar, por lo tanto son programas cuya sintaxis debería estar aprobada por el compilador del lenguaje.

En vez de comprobar si una expresión está bien escrita, antes de ejecutar nada con la especificación, asumiremos que el código fuente que recibe la herramienta es correcto en los términos sintácticos del lenguaje. De modo que la jerarquía de tipos de nuestra herramienta es bastante sencilla, como se muestra en las figuras 5.2 y 5.3.[2] Por ejemplo, las listas (ListValue) y las tuplas (TupleValue) son subtipos del concepto de contenedor (ContainerValue).

Una decisión, que ayudó bastante a simplificar la implementación, fue utilizar el tipo nativo para las listas en Maude. Pero en vez de usarlo directamente, se hizo un módulo derivado de `LIST` para extender algunas funcionalidades del tipo `List` que íbamos a necesitar:

- `elemAt`: Devuelve el elemento en la posición N de la lista.
- `hasAnyElem`: Comprueba que la lista no está vacía.
- `subtract`: Elimina los elementos que se encuentra en la segunda lista de la primera. Esta operación fue implementada para seguir el mismo funcionamiento que el operador de listas `--` de Erlang.

[2]Los tipos con un asterisco, en los diagramas, corresponden a tipos del preludio de Maude.

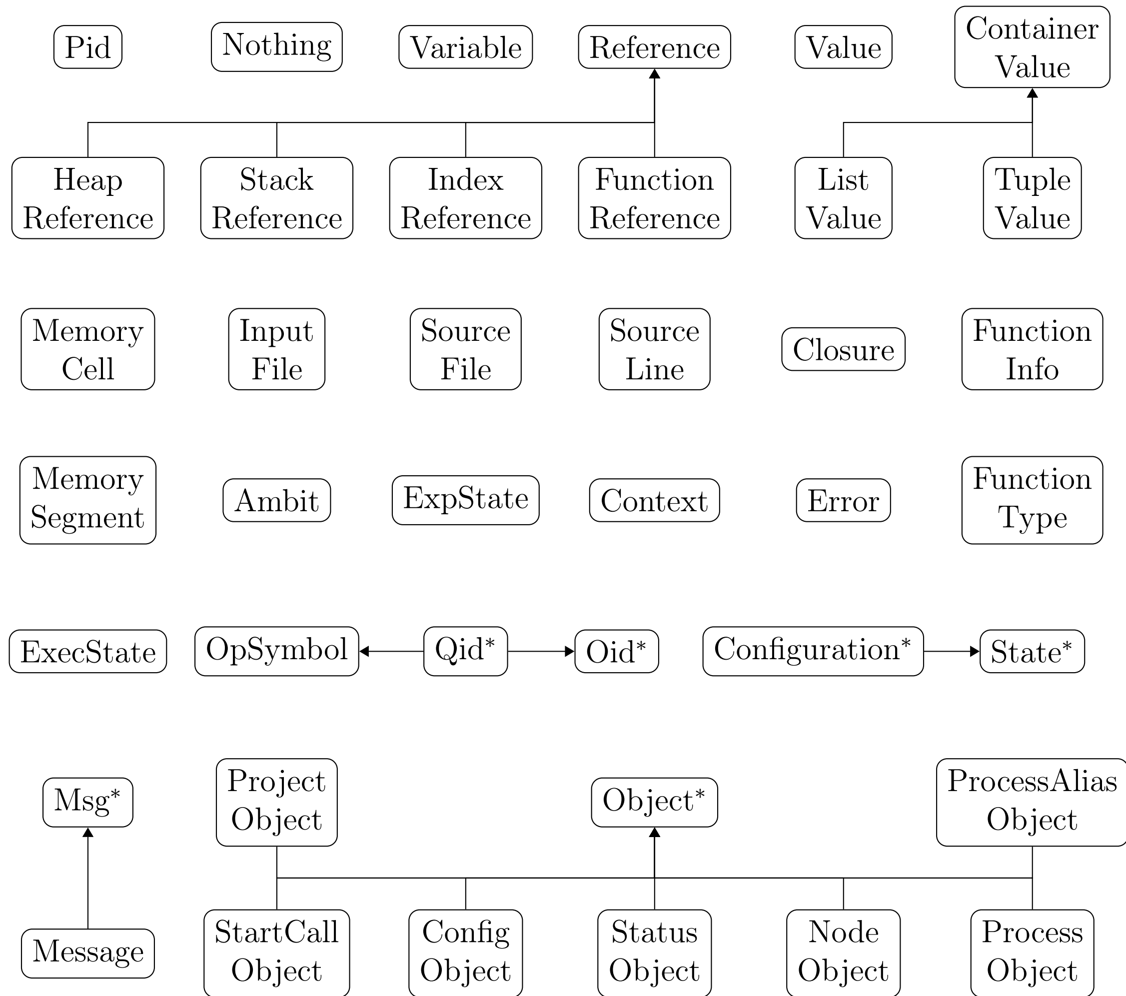

Figura 5.2: *La jerarquía de tipos en Selene, parte 1.*

El utilizar las listas existentes nos evita crear, de forma duplicada, infinidad de veces las mismas operaciones para representar los mismos conceptos más de una vez. Esto es posible gracias a la parametrización de módulos usando las *teorías* y *vistas* de Maude, como puede verse en la figura 5.4.[3] Los módulos del ejemplo están entre paréntesis porque para este proyecto se decidió usar Full Maude.[4]

[3] `---` y `***` son usados en Maude para escribir comentarios de una línea y ambos significan lo mismo. Para comentarios de varias líneas se utiliza `---(Texto...)` y `***(Texto...)`, de nuevo ambos significan lo mismo.

[4] Full Maude es una versión mejorada del propio Maude —usando el *metanivel*— para añadir nuevas características al lenguaje, como por ejemplo la nomenclatura para objetos.

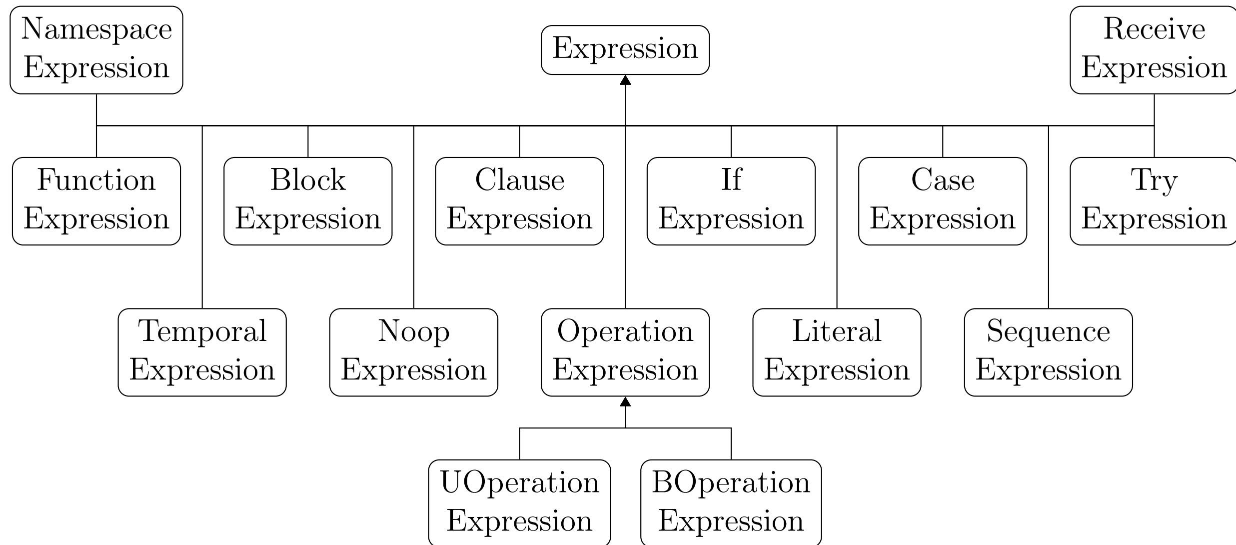

Figura 5.3: *La jerarquía de tipos en Selene, parte 2.*

Los objetos en Maude han de tener un identificador de objeto, representado por el tipo `Oid`. Podemos definir nuestros propios constructores para el tipo `Oid`; sin embargo, si necesitamos una cadena identificadora, la forma más sencilla de lograrlo es usando el tipo `Qid`.[5] Por otro lado, debido a que vamos a estar en un escenario de programación concurrente, tendremos nodos y procesos. Por ello, tenemos para `Oid` el siguiente constructor: `op @id : Nat -> Oid [ctor] .`; que toma como argumento un número natural y construye un identificador del objeto en lugar de usar un identificador alfanumérico al uso.

El tipo `Pid` (Process Identifier), que es utilizado para identificar a un proceso dentro de un nodo en el modelo, tiene el constructor `@pid`. Este constructor contiene dos naturales, el primero representa el nodo y el segundo el proceso al que hace referencia.

Finalmente, Selene incorpora el tipo `Nothing`, que es utilizado para representar valores vacíos (como por ejemplo `void` en C). También se utiliza como elemento auxiliar a la hora de implementar construcciones sin terminar, para rellenar huecos que podrían ser utilizados en un futuro.

[5] El tipo `Qid` en Maude consiste en todas aquellas cadenas de caracteres que comienzan por comilla simple. Por ejemplo: `'123`, `'hello`, etcétera.

```
(fmod LIST2 {X :: TRIV} is
   inc LIST{X} .
   --- More code to extend the list module of Maude...
endfm)

(fmod VIEW-TYPES is
   sort Value .
   --- More code...
endfm)

(view Value from TRIV to VIEW-TYPES is
   sort Elt to Value .
endv)

(fmod VALUE is
   --- Imports code...
   pr LIST2{Value} * (sort List{Value} to Values) .
   *** This last line imports the extended module of the lists
   *** and makes the Values alias to the sort List{Value}.
   --- More code...
endfm)
```

Figura 5.4: *Usando las listas del preludio de Maude.*

5.2. Proyectos, fuentes y líneas

El elemento de entrada principal en nuestro sistema es el proyecto con el código fuente, que puede traer uno o más ficheros fuente. Los ficheros de código fuente son representados con el tipo `InputFile`, que consiste en una tupla de dos cadenas, la primera el nombre del fichero y la segunda el texto del mismo. La operación `makeInputFile` se utiliza a modo de función constructora para crear términos del tipo `InputFile`. Se puede ver un ejemplo de su uso en la figura 5.5, con el resultado de su reducción en la figura 5.6.

El tipo `SourceFile` se utiliza para almacenar ficheros de código fuente. La diferencia entre este e `InputFile`, es que `SourceFile` tiene un campo adicional para almacenar el número de líneas que tiene el fichero. Este valor es calculado una sola vez, ya que el tamaño del código fuente rara vez cambia durante la ejecución del programa. Además, el sistema

```
op prj1ifs : -> InputFiles .
eq prj1ifs = makeInputFile("file 1", "line 1\nline 2\nline 3")
             makeInputFile("file 2", "line 1\nline 2")
             makeInputFile("file 3", "") .
```

Figura 5.5: *Código usando el tipo* ***InputFile****.*

```
Maude> (red prj1ifs .)
reduce in TESTS-PROJECT :
  prj1ifs
result NeList{InputFile} :
  {"file 1","line 1\nline 2\nline 3"}
  {"file 2","line 1\nline 2"}
  {"file 3",""}
```

Figura 5.6: *Salida por consola del código usando el tipo* ***InputFile****.*

utiliza el tamaño en líneas de los ficheros, para usar un valor de índice que nos permita localizar una línea determinada en el proyecto. Se puede ver un ejemplo en la figura 5.7 —que utiliza el valor `prj1ifs` de la figura 5.5—, con el resultado de su reducción en la figura 5.8.

Una vez que tenemos la lista de ficheros fuente con toda la información, podemos crear un objeto proyecto para almacenar esta información y así añadirlo dentro del conjunto de tipo `Configuration`[6] —del que dispone Maude para representar multiconjuntos de objetos y mensajes— para almacenar los objetos de nuestro modelo. Se puede ver un ejemplo en la

[6]`Configuration` representa un multiconjunto de objetos y mensajes.

```
op prj1sfs : -> SourceFiles .
eq prj1sfs = inputToSourceFiles(prj1ifs) .
```

Figura 5.7: *Código usando el tipo* ***SourceFiles****.*

```
Maude> (red prj1sfs .)
reduce in TESTS-PROJECT :
  prj1sfs
result NeList{SourceFile} :
  @sf("file 1","line 1\nline 2\nline 3",3)
  @sf("file 2","line 1\nline 2",2)
  @sf("file 3","",1)
```

Figura 5.8: *Salida por consola del código usando el tipo `SourceFiles`.*

```
op prj1obj : -> Object .
eq prj1obj = newProject('Prj, prj1ifs) .
```

Figura 5.9: *Código usando el tipo `ProjectObject`.*

figura 5.9 —que utiliza el valor `prj1ifs` de la figura 5.5—, con el resultado de su reducción en la figura 5.10.

Nuestro sistema necesita referenciar las líneas dentro de los proyectos que analiza. Una forma simple hubiera sido utilizar una tupla $\langle$*nombre, línea*$\rangle$ para localizar las líneas, como el tipo `SourceLine`. Esto haría que los programas —en el modelo usado por Selene— fueran de un tamaño mucho mayor, con las mismas cadenas repetidas infinidad de ocasiones. Por este motivo usamos un índice global en su lugar, pues se trata de una forma más compacta y la

```
Maude> (red prj1obj .)
reduce in TESTS-PROJECT :
  prj1obj
result ProjectObject :
  < 'Prj : Project | files :(@sf("file 1","line 1\nline 2\nline 3",3)
    @sf("file 2","line 1\nline 2",2)@sf("file 3","",1))>
```

Figura 5.10: *Salida por consola del código usando el tipo `ProjectObject`.*

```
op prj1nts : -> Nats .
eq prj1nts = lineToIndex(prj1obj, makeSourceLine("file 1", 0))
             lineToIndex(prj1obj, makeSourceLine("file 1", 1))
             lineToIndex(prj1obj, makeSourceLine("file 1", 2))
             lineToIndex(prj1obj, makeSourceLine("file 1", 3))
             lineToIndex(prj1obj, makeSourceLine("file 1", 4))
             lineToIndex(prj1obj, makeSourceLine("file 2", 0))
             lineToIndex(prj1obj, makeSourceLine("file 2", 1))
             lineToIndex(prj1obj, makeSourceLine("file 2", 2))
             lineToIndex(prj1obj, makeSourceLine("file 2", 3))
             lineToIndex(prj1obj, makeSourceLine("file 3", 0))
             lineToIndex(prj1obj, makeSourceLine("file 3", 1))
             lineToIndex(prj1obj, makeSourceLine("file 3", 2))
             lineToIndex(prj1obj, makeSourceLine("file 3", 3)) .
```

Figura 5.11: *Código usando los índices de línea.*

```
Maude> (red prj1nts .)
reduce in TESTS-PROJECT :
  prj1nts
result NeList{Nat} :
  0 1 2 3 0 0 4 5 0 0 6 0 0
```

Figura 5.12: *Salida por consola del código usando los índices de línea.*

conversión —entre este índice y el tipo `SourceLine`— es relativamente sencilla. Se puede ver un ejemplo de la conversión de `SourceLine` a índice usando la operación `lineToIndex` en la figura 5.11 —tomando el valor `prj1obj` de la figura 5.9—, con el resultado de su reducción en la figura 5.12; y su contrapartida usando la operación `indexToLine` en la figura 5.13, con el resultado de su reducción en la figura 5.14.

En los ejemplos mostrados, el índice cero y la tupla ⟨*“”, 0*⟩ representan que la línea no ha sido localizada en el interior del proyecto de código fuente. En la sección 5.5 veremos cómo el índice global es utilizado con los tipos de expresiones que conforman los programas.

```
op prj1sls : -> SourceLines .
eq prj1sls = indexToLine(prj1obj, 0) indexToLine(prj1obj, 1)
             indexToLine(prj1obj, 2) indexToLine(prj1obj, 3)
             indexToLine(prj1obj, 4) indexToLine(prj1obj, 5)
             indexToLine(prj1obj, 6) indexToLine(prj1obj, 7) .
```

Figura 5.13: *Código usando el tipo* `SourceLines`*.*

```
Maude> (red prj1sls .)
reduce in TESTS-PROJECT :
  prj1sls
result NeList{SourceLine} :
  @sl("",0)
  @sl("file 1",1)
  @sl("file 1",2)
  @sl("file 1",3)
  @sl("file 2",1)
  @sl("file 2",2)
  @sl("file 3",1)
  @sl("",0)
```

Figura 5.14: *Salida por consola del código usando el tipo* `SourceLines`*.*

5.3. Las referencias

Una variable —en casi cualquier lenguaje de programación— representa el alias de un valor determinado, incluso si este lenguaje permite modificar el valor de la misma tras su declaración. En lenguaje máquina, estos alias representan la dirección de memoria donde el valor está realmente almacenado. Muchas veces esa dirección es relativa a un segmento de memoria declarado. Así que cuando un programa es compilado a lenguaje máquina, los nombres de variables son sustituidos por las direcciones que representan. Conociendo estos detalles, el tipo `Variable` contiene:

- El **nombre** de la variable, almacenado como un `Qid`.
- La **lista de referencias**, que nos permite localizar el valor —representado por la variable— dentro de la memoria.

En muchos lenguajes la memoria se divide entre el montículo y la pila asociada al programa. Depende de la semántica de estos lenguajes ver dónde se almacenan los valores para cada construcción del propio lenguaje. Debido a ello, las referencias en Selene se dividen en cuatro categorías:

- **Referencias del montículo**: Se usan para encontrar un valor en el interior de un segmento de tipo montículo en la memoria. Se compone del **identificador de nodo** donde el segmento se encuentra y de la **dirección de memoria** dentro del segmento.
- **Referencias a la pila**: Se usan para encontrar un valor en el interior de un segmento de tipo pila en la memoria. Se compone del **identificador de nodo** donde el segmento se encuentra, del **identificador de proceso** al que pertenece el segmento y de la **dirección de memoria** dentro del segmento.
- **Referencias con índices**: Se usan para encontrar un valor en el interior de un valor de tipo contenedor, como las listas y las tuplas. Se compone de la **posición** del

valor dentro del contenedor y del **número de elementos** del fragmento al que hace referencia dentro del contenedor (por defecto este valor es `1`).

- **Referencias a funciones**: Se usan para encontrar una función dentro del programa. Se compone de una **lista de Qids** con los nombres a seguir para encontrar la función.

Las referencias podrían verse como punteros, en cierto modo; pero como explicaremos en la sección 5.6, la memoria en nuestro modelo no es un bloque lineal de celdas (como ocurre en las máquinas físicas que ejecutan programas). Por esta clase de motivos, necesitamos algo más que una dirección de memoria para localizar un valor.

5.4. Los valores

Por motivos relativos a la implementación interna de Maude, no podemos juntar los diferentes tipos de valores —que trae el preludio— bajo un único tipo.[7] Esto supone un problema para poder implementar la especificación de un lenguaje, con lo que al final tenemos que recurrir a crear un tipo encapsulador para unificar estos valores bajo un mismo concepto.

En el caso de Selene —en vez de tener una jerarquía de tipos que representen valores— solo tenemos el tipo `Value`, que trae consigo diferentes constructores para encapsular cada uno un tipo de valor distinto. Los tipos que encapsula el tipo `Value` son: átomos (representados con `Qid`), Booleanos, números naturales, números enteros, números reales, cadenas de

[7]El problema en cuestión es que al unificar tipos como `Int` y `Float`, por ejemplo, Maude nos avisa de que existe una colisión de operadores y que no queda claro cuál se debería usar al parsear una expresión (es el caso del operador suma, que ambos tipos lo implementan, pero tienen firmas distintas). Técnicamente esto se puede solucionar reescribiendo el preludio entero, para hacer una versión donde cada tipo tenga operadores con un nombre distinto a los de otros tipos. Esto implica un esfuerzo adicional y rompería con el desarrollo actual de Maude, por lo que no parece realmente una buena idea a tener en cuenta.

Otra alternativa consistiría en usar el renombramiento al importar los módulos, pero como efecto colateral tendríamos que importar todos los módulos de Maude usando el renombramiento de todos los operadores problemáticos cada vez que los importemos. Esto llevaría a tener que obligar a los usuarios a tener que evitar usar los módulos de Maude directamente y tener que reescribir partes de su código para trabajar con nuestro sistema.

caracteres, secuencias de valores (listas y tupas), punteros (usando las referencias explicadas en la sección 5.3), objetos función, identificadores de procesos, errores y el elemento “vacío”.

La secuencia de valores es simplemente una lista de valores, usando las listas de Maude. Los objetos función contienen un valor de tipo `FunctionInfo`, que se compone de una lista de Qids para indicar su nombre, el tipo de la función —estática, lambda, método de objeto o inválida (que se usa para manejar errores durante la ejecución de los programas)— y una lista de clausuras. Una clausura en Selene es una tupla ⟨*nombre, valor*⟩, que será añadida al ámbito actual —del contexto de la función en la que estamos— cuando sea invocada la instancia de la lambda que contiene dicha clausura.

Algunos de los elementos mostrados en esta sección no se necesitaban para implementar las semánticas de Erlang (por ejemplo, las referencias a métodos de objetos), pero se añadieron a la implementación de cara a dar soporte en el futuro a otros lenguajes candidatos para ser utilizados por Selene.

En la idea inicial del proyecto, la idea era hacer un sistema parametrizable usando las semánticas del lenguaje a analizar, pero durante la implementación de la herramienta el objetivo se fue perdiendo. En lugar de parametrizarse, el marco que ofrece Selene incluye funcionalidad para diferentes lenguajes y es en los módulos del motor de ejecución donde se configura la máquina, para adaptar la herramienta a la semántica que se intenta implementar. Este esquema no está exento de detalles cuestionables, como por ejemplo las funciones de conversión para el tipo `Value`, donde tenemos el caso de la operación `toString` cuya implementación seguía las semánticas de Erlang y ciertamente esta puede diferir de la de otros lenguajes. Esa y más operaciones debieron ser parametrizadas de alguna forma en la implementación, pero debido a la falta de experiencia del autor y de la falta de tiempo (para implementar todas las características que se pensaron inicialmente), el proyecto terminó en un estadio intermedio entre partes genéricas y partes que seguían las reglas que tiene Erlang.

5.5. Las expresiones

En el modelo que sigue Selene, los programas están compuestos por expresiones. Como se mencionó anteriormente en la sección 5.1, la idea a la hora de expresar la sintaxis era hacer un sistema de tipos sencillos, con el que poder representar dicha sintaxis de forma rápida. Esto por un lado podría desembocar en recibir un programa mal escrito, que hiciera fallar la ejecución, pero asumiremos que el usuario sólo nos pasará programas que realmente sean correctos en cuanto a su sintaxis. Como resultado de esta decisión, la jerarquía resultante de los tipos que representan las expresiones se pueden ver en la figura 5.3.

A raíz de pretender usar la herramienta con diferentes lenguajes y de la necesidad de tener en las expresiones el número de línea dentro del código fuente, surgió la idea de construir un lenguaje genérico. Este lenguaje está inspirado en la forma que Erlang tiene para representar los árboles sintácticos de sus módulos y fue diseñado con el objetivo de expresar los mismos programas que el lenguaje de origen. Las estructuras implementadas

Símbolo	Tipo	Campos
`@lt`	Literal	Índice, Valor
`@op`	Operación unaria	Índice, Símbolo, Expr. Der.
`@op`	Operación binaria	Índice, Símbolo, Expr. Izq., Expr. Der.
`@sq`	Secuencia	Índice, Lista de Expresiones
`@ns`	Espacio de nombres	Índice, Nombre, Declaraciones
`@fn`	Función	Índice, Nombre, Cláusulas
`@cs`	Cláusula	Índice, Patrón, Guarda, Cuerpo
`@bk`	Bloque	Índice, Cuerpo
`@if`	`if-else`	Índice, Cláusulas, Cuerpo Else
`@cf`	`case/switch`	Índice, Expr. de Prueba, Cláusulas
`@tc`	`try-catch`	Índice, Expr. de Prueba, Cláusulas/Cuerpo, Cláusulas Catch, Cuerpo Finally
`@rc`	`receive-after`	Índice, Cláusulas, Cláusula After

Figura 5.15: *Las estructuras en el lenguaje genérico de árboles sintácticos.*

actualmente en el "Lenguaje Genérico de Árboles Sintácticos" (LGAS)[8] se pueden ver en la figura 5.15.

LGAS, además de estar influido, está centrado en Erlang principalmente. El motivo de esta elección es que no se han probado otros lenguajes candidatos en la versión actual de la herramienta, detalle que hace que resulte cuestionable si realmente esta parte del lenguaje es "genérica". Además, incorporar un lenguaje intermedio requeriría demostrar que las transformaciones —de un lenguaje $\mathcal{L}$ a LGAS— tienen un mapeado isomórfico. Esta demostración debería ser realizada —como trabajo futuro— si Selene continuara en el futuro utilizando LGAS.

Expliquemos un poco más en detalle los constructores vistos en la figura 5.15, así como el significado de los conceptos que representan:

- **Literal**: Símbolos o valores literales en el código.
- **Operaciones**: Operaciones unarias o binarias, cuyo símbolo se almacena como un `Qid`.
- **Secuencia**: Una secuencia de expresiones. Se puede utilizar para representar los parámetros que recibe una función en la operación de llamada a la misma.
- **Espacio de nombres**: Un bloque con nombre de declaraciones (funciones, clases, etcétera).
- **Función**: Una función con nombre, que contiene varias cláusulas posibles.
- **Cláusula**: Una cláusula es una construcción con parámetros, condiciones y un cuerpo a ejecutar. Sería como lo siguiente: *(parámetros) cuando (condiciones) haz (expresiones)*. Por ejemplo, cuando llamamos a una función con una serie de parámetros, la ejecución intentará seleccionar todas aquellas cláusulas con el mismo número de parámetros, entonces comprobará una por una si encajan los valores recibidos con los parámetros

[8] En ingles el nombre es: Generic Syntax Trees Language (GSTL).

declarados por la cláusula, después si la guarda se cumple y por último se ejecutan las expresiones del cuerpo de la cláusula. Un detalle a tener en cuenta es que todos los campos son opcionales a la hora de usarla para construir otras expresiones.

- **Bloque**: Un bloque de expresiones. Estos crean un nuevo ámbito de ejecución, cuando el programa entra en ellos.
- `if-else`: La ya conocida sentencia `if-else`, donde las cláusulas representan cada condición a ser comprobada antes de proseguir por el camino del `else`. En este caso, el campo usado para los parámetros en la cláusula no sería necesario su uso, esto se debe a que las condiciones a comprobar estarían en el campo de la guarda.
- `case/switch`: Esta expresión toma una expresión de prueba, la evalúa y comprueba si el resultado encaja en alguna cláusula definida dentro de la expresión `case` o `switch`.
- `try-catch`: Similar a la expresión `case/switch` anterior, pero —en esta ocasión— si ocurre un error durante la ejecución, comprobará el valor de este sobre las cláusulas que estén definidas en el `catch` para ver si en alguna encaja el error devuelto. Independientemente de si hay error o no, un cuerpo se puede definir en el `finally` para ejecutar código tras el `try-catch`.
- `receive-after`: Esta expresión está tomada de la misma que Erlang tiene, utilizada para parar la ejecución de un proceso para recibir un mensaje de la cola de mensajes del propio proceso. Si el `after` está definido con un número de milisegundos, tras transcurrir dicho tiempo esperando la llegada de un mensaje, la expresión `receive` terminará su ejecución incluso si no llega ningún mensaje.

Conceptos como los bucles, clases y demás, pueden ser añadidos al LGAS en un futuro, extendiendo la funcionalidad y aquello que representa el lenguaje. Pero en el estado actual del proyecto Selene, solo puede representar programas escritos en Erlang.[9] Para ver cómo

[9] Todavía quedan añadir algunos detalles al lenguaje, para poder representar algunos conceptos como

```
% Erlang version:                     --- GSTL version:
-module(helloworld).                  @ns(1, 'test,
                                        @fn(3, 'main,
main() ->                                 @cs(3, nil, nil,
  print("Hello, world!\n").                 @op(4, @call,
                                              @lt(4, 'print),
                                              @sq(4,
                                                @lt(4, "Hello, world!\n")
                                              )))))
```

Figura 5.16: *El ejemplo "hola mundo" en Erlang transformado al LGAS.*

quedarían los programas traducidos, de Erlang al LGAS, tenemos ejemplos en las figuras 5.16 y 5.17.

En los ejemplos de código aquí mostrados, en lugar de usar la función `io:format`, usamos —para la salida por consola— una función llamada `print`. Esto se debe a que para poder ejecutar Erlang, sería necesario implementar parte de su API estándar (o incorporar —al proyecto que se analiza— los ficheros con los módulos utilizados), así como todas las BIFs[10] del lenguaje. Es necesario añadir estos elementos, como parte del trabajo futuro, si se quiere asegurar que la herramienta pueda ser utilizada con proyectos reales y no solo con ejemplos de prueba.

5.6. La memoria

El modelo de la memoria en Selene está diseñado como un espacio dividido en segmentos de dos tipos: de montículo y de pila. El cómo y por qué de dónde se almacenen los valores que maneja el programa depende de la semántica del lenguaje.

Los segmentos están representados por el tipo `MemorySegment`, que se utiliza para al-

expresiones lambda, listas intensionales, etcétera. Además de esta flaqueza existente, las semánticas de muchas de las estructuras que ya existen —como el `if-else`— no están implementadas en la versión actual de la herramienta. Es de esperar ir avanzando en todos estos aspectos en líneas de trabajo futuro.

[10]Built-In Functions, que son funciones que pertenecen al lenguaje pero están implementadas con código en lenguaje nativo, en vez del propio lenguaje Erlang.

```
% Erlang version:
-module(test).

server() ->
   register(server, self()),
   server_loop().

server_loop() ->
   receive V ->
      print(V, "\n"),
      server_loop(V)
   end.

worker() ->
   server ! "EXTERMINATE",
   server ! "ANNIHILATE",
   server ! "DESTROY".

--- GSTL version:
@ns(1, 'test,
   @fn(3, 'server,
      @cs(3, nil, nil,
         @op(4, @call, @lt(4, 'register), @sq(4, @lt(4, 'server)
            @op(4, @call, @lt(4, 'self), @sq(4, nil))))
            @op(5, @call, @lt(5, 'server_loop), @sq(5, nil))))
   @fn(7, 'server_loop,
      @cs(7, nil, nil,
         @rc(8, @cs(8, @lt(8, 'V), nil,
            @op(9, @call, @lt(9, 'print), @sq(9, @lt(9, 'V) @lt(9, "\n")))
            @op(10, @call, @lt(10, 'server_loop), @sq(10, nil))
         ), nil)))
   @fn(13, 'worker,
      @cs(13, nil, nil,
         @op(14, @snd,  @lt(14, 'server), @lt(14, "EXTERMINATE"))
         @op(15, @snd,  @lt(15, 'server), @lt(15, "ANNIHILATE"))
         @op(16, @snd,  @lt(16, 'server), @lt(16, "DESTROY")))))
```

Figura 5.17: *El ejemplo "servidor echo" en Erlang transformado al LGAS.*

macenar una lista de celdas de memoria. Dichas celdas están representadas por el tipo `MemoryCell`, que se trata de una tupla ⟨*dirección, valor*⟩. El valor está representado por el tipo `Value` que hemos visto en la sección 5.4.

El módulo `MEMORY` nos permite usar los segmentos de memoria —de nuestro modelo— como segmentos de pila, con operaciones como: `pushValue`, `popValue`, `topValue`, etcétera. El resto de conceptos que hay en dicho modulo sirve para leer y escribir valores en los tipos definidos.

El ámbito es un concepto que guarda consigo la información asociada a un bloque de ejecución de código. Por ejemplo las variables pertenecen a un ámbito concreto. Su composición es la siguiente:

- **Código**: La lista de expresiones que la máquina va a ejecutar dentro del ámbito.
- **Estado**: El estado en el que se encuentra la expresión actual que va a ser ejecutada. Un ejemplo de transiciones entre estados —a la hora de ejecutar expresiones en un programa— está visible en la figura 5.18.
- **Variables**: La lista de las variables actuales dentro del ámbito. Estas variables contienen los tipos de referencias de los que ya hemos hablado en la sección 5.3.

El motivo para poner el código a ejecutar dentro del ámbito, y no en otro lugar del modelo, se debe a que de este modo es más fácil de manejar su ejecución, así como la limpieza del mismo a la hora de salir del ámbito en cualquier punto del mismo. Pongamos de ejemplo la expresión `while` (aunque no exista en nuestro modelo), al ejecutarlo comprobaríamos su condición para entrar en el cuerpo del mismo o no. Si tenemos que entrar, se crea un ámbito nuevo, con el cuerpo del `while` y lo ejecutamos. Al final de la ejecución del cuerpo, o en un punto de ruptura (por ejemplo un *break*), saldríamos del ámbito. Al salir limpiamos todas las variables nuevas creadas en ese ámbito, volvemos a comprobar la condición del `while` y vuelta a empezar.

El contexto es un concepto que guarda consigo la información asociada a la ejecución de una función. Cada vez que entramos en una función que ha sido invocada, se crea un nuevo contexto de ejecución. La relación entre un contexto y un ámbito, es que los ámbitos se van apilando en el interior del contexto. Los contextos se componen de lo siguiente:

- **Función**: El nombre completo de la función, para ello se usa una lista de Qids.
- **Ámbitos**: La pila de ámbitos del contexto actual.
- **Pila**: El segmento de memoria que se utiliza para representar la sección de la pila asociada al contexto actual.
- **Resultado**: El valor que será devuelto al terminar la ejecución del contexto.

El segmento de pila está particionado entre todos los contextos de la pila de contextos del programa actual en ejecución. Se optó por hacerlo de este modo, en lugar de un segmento de pila único externo a todos los contextos, porque resultaba más sencillo para la limpieza de los valores de la pila; pues una vez se elimina el contexto, se elimina también los datos que contiene con él.

5.7. El mundo

En el modelo de ejecución de Selene, el *mundo* consiste en un conjunto de objetos donde se reúnen todos los elementos relativos a la aplicación: el programa a ser ejecutado, los nodos, los procesos, etcétera. Para implementar el *mundo* se utiliza el tipo `Configuration` de Maude, que se emplea para contener objetos y mensajes.

La nomenclatura para objetos es una característica de Full Maude. Para usarla debemos emplear las siguientes construcciones:

Definición : `class` $nombre_c$ | $nombre_1$: $sort_1$, ... , $nombre_n$: $sort_n$.

Instancia : < $identificador$: $nombre_c$ | $nombre_1$: $valor_1$, ... , $nombre_n$: $valor_n$ >

El *nombre* de la clase es cualquier identificador soportado por Maude. El *identificador* del objeto es un valor de tipo `Oid`. Los mensajes se definen de un modo similar a las operaciones. Sabiendo estos aspectos básicos, sobre los objetos en Maude, vamos a hablar sobre los objetos que se usan dentro de Selene.

5.7.1. El objeto `Project`

Ya mencionamos al objeto `Project` en la sección 5.2 y mostramos su aspecto en la figura 5.10. Este objeto solo tiene un campo que contiene la lista de ficheros fuente del proyecto. En la versión actual de la herramienta no tiene ningún uso real durante la ejecución.

5.7.2. El objeto `Config`

El objeto `Config` se utiliza para configurar el *mundo* en nuestro sistema. Contiene los siguientes campos:

- `project`: Los ficheros fuente del proyecto de entrada.
- `nodes`: El número de nodos del *mundo*.
- `startCalls`: El conjunto de objetos que definen las llamadas de inicio.

El objeto `StartCall` representa la llamada a una función para crear un proceso dentro de un nodo. Contiene los siguientes campos:

- `node`: El número de nodo donde la llamada será ejecutada.
- `function`: El nombre de la función de entrada, almacenada con una lista de Qids.
- `params`: La lista de valores que serán usados como parámetros para la función de entrada.

Así que cuando se va a ejecutar un proyecto, necesitamos los ficheros de código fuente, saber la cantidad de nodos que vamos a necesitar y cuáles son los puntos de entrada para

cada nodo dentro de la biblioteca del programa. Este en líneas generales es el propósito del objeto de configuración.

5.7.3. El objeto `Status`

El objeto `Status` se utiliza como objeto auxiliar para que la máquina pueda guardar y obtener información útil que necesite para la ejecución de un programa. Contiene los siguientes campos:

- `program`: Se trata de la biblioteca del programa, donde están todas las declaraciones usadas por el proyecto. Cada vez que una expresión invoca una función, la máquina busca el objeto de estado para obtener el campo `program` y obtener de la biblioteca la función a llamar.
- `nextIndex`: Se trata de un número auxiliar usado para obtener el siguiente índice en el sistema. Uno de los usos que tiene es dar un número de identificación único a los procesos, en vez de tener que recabar todos los valores asignados y generar uno nuevo no usado.

5.7.4. El objeto `Node`

El objeto `Node` se utiliza para representar los nodos que hay en el interior del *mundo*. Por ejemplo, un nodo puede ser un computador en una red. Los nodos contienen procesos, que son ejecutados de forma concurrente. Los nodos contienen los siguientes campos:

- `heap`: El segmento de memoria del montículo en el nodo.
- `cout`: El buffer de salida del nodo.
- `cin`: El buffer de entrada del nodo.
- `info`: Se trata de un campo auxiliar que contiene otros objetos, los cuáles pueden contener información relevante para la ejecución del programa. Por ejemplo, en el

motor de ejecución de Erlang, este campo contiene objetos de tipo `ProcessAlias` que dan —con un átomo— un alias a los procesos existentes.

5.7.5. El objeto `Process`

El objeto `Process` se utiliza para representar un proceso dentro de un nodo que ejecuta código. Los procesos pueden enviar y recibir mensajes. Contiene los siguientes campos:

- `owner`: El número identificador del nodo donde el proceso está ejecutándose.
- `context`: La pila de contextos del proceso. Con este elemento la máquina controla la ejecución del proceso.
- `messages`: El buzón con la cola de mensajes recibidos por el proceso.
- `newMsgsFlag`: El flag de "nuevo mensaje" que indica a la ejecución si se ha recibido un nuevo mensaje desde la última vez que se revisó el buzón.

5.7.6. Los mensajes

Selene fue diseñado para permitir programación concurrente con paso de mensajes, por ello tenemos mensajes en nuestro *mundo*. Los mensajes contienen los siguientes campos:

- `MsgID`: El número identificador del mensaje. Este valor se utiliza para saber el orden de creación de los mensajes a lo largo del tiempo.
- `OrigPID`: El identificador de proceso origen del mensaje.
- `DestPID`: El identificador de proceso destino del mensaje.
- `Value`: La información contenida por el mensaje.

Para hacer identificadores de mensajes, la máquina de Selene usa el campo `nextIndex` del objeto estado que hay en el *mundo*. De esta forma logramos que se mantenga que un número identificador mayor que otro mensaje, se haya creado después que dicho mensaje con un número menor.

5.8. La máquina

Ahora que sabemos de qué está compuesto el *mundo* de nuestro modelo, vamos a hablar sobre el núcleo de ejecución de la máquina implementada en Selene.

5.8.1. El núcleo de ejecución

El núcleo de Selene busca ofrecer una serie de funcionalidades genéricas en un entorno al que llamamos el `mundo`, donde se encuentra toda la información que maneja el programa.

Entre esas funcionalidades, la más destacable es la invocación de funciones a través de la operación `invokeFunction`. En su interior comprueba el tipo de función que quiere invocar y crea un nuevo contexto para la llamada con la operación `addContext`, que a su vez llama a `makeContext`. Un dato curioso es que a la hora de crear los procesos, con las "llamadas de inicio", se invoca a la operación `makeContext` directamente; esto se debe a que al empezar un nuevo proceso no hay ningún contexto previo. Dentro de `makeContext`, si la operación `checkClause` da por positiva alguna de las cláusulas asociadas a la función, se crea el contexto y su ámbito inicial con `makeAmbitAndContext` pasándole las variables creadas con `makeVariables`.

Comprobar una cláusula con `checkClause` implica primero comprobar los valores de entrada a la función frente a los parámetros con `checkParameters`. Esta operación comprueba primero que el tamaño de los valores de entrada y de los parámetros sea equivalente, luego por cada valor comprueba que encaje en el parámetro que le toca usando la operación `checkParameter`. Esta última operación no está implementada para todos los posibles encajes que Erlang puede ofrecer, sin embargo sí que está preparada para encajar con valores y con variables simples. El flag `CC:Bool` de la operación `checkParameter` tiene un uso especial, cuando se crea un contexto vale falso, pero cuando se crea un ámbito puede valer cierto, esto se debe a que cuando entramos en una expresión —como el `case-of` de Erlang— se crea un ámbito al entrar en el bloque de una de las cláusulas de la expresión. Cuando el flag vale cierto, lo que fuerza es que en el caso de encontrar una variable, se tiene

que buscar en el proceso actual si ya ha sido declarada, para obtener su valor y compararlo con el valor de entrada, mirando que encajen ambos valores. Esto último se realiza con la operación `checkVarMatch`.

Lo segundo que hace la operación `checkClause`, tras comprobar el encaje de los parámetros, es comprobar las condiciones que se encuentren en la guarda de la cláusula. En futuras versiones de la herramienta queda planeado implementar completamente dicha función.

Igual que tenemos una función para añadir un contexto a la ejecución del programa, tenemos su análoga para los ámbitos que son `addAmbit` y `makeAmbit`. El mecanismo es casi por completo igual que al crear un contexto, lo que cambia es el flag `CC:Bool` que la operación recibe, dejando que sea la semántica del ejecutor implementado quien decida si activarlo o no (cosa que no ocurre con el contexto, pues siempre vale falso).

La operación `makeVariables` sería la encargada de, tomando las estructuras de datos descritas en los parámetros, formar una lista de variables que pasaran a añadirse al ámbito que se acabe de crear. Actualmente esta función solo se encuentra implementada para variables simples, que pueden almacenar cualquier valor que se les pase.

Otro tipo de operaciones que ofrece el núcleo de la máquina es buscar alguna declaración en la biblioteca del programa, usando la operación `findExpression`. Se le pasa el nombre completo de lo que estamos buscando con una lista de Qids y el *mundo* de la aplicación, entonces obtiene la biblioteca actual para buscar —desde el exterior al interior— la declaración que estamos buscando.

También están las funcionalidades para crear nodos con `makeNodes` y crear procesos con `makeProcesses`, operaciones que intervienen en el arranque del programa. Para crear un solo proceso tenemos `startProcess`, que es llamada por `makeProcesses`. Otra operación con procesos que nos ofrece el núcleo es encontrar un proceso determinado usando `findProcess`.

La última funcionalidad a destacar es para añadir al *mundo* un nuevo mensaje con la operación `addMessage`, que ayuda a configurar algunos de los campos que tienen los mensajes de forma automática (como es el caso del identificador del mensaje).

5.8.2. El estado de ejecución

Para facilitar la implementación de las operaciones genéricas del núcleo se creó un tipo nuevo llamado `ExecState`, que estaba compuesto por la siguiente información:

- **World**: El *mundo* de nuestra aplicación.
- **Node**: El nodo del proceso que está siendo ejecutado actualmente.
- **Process**: El proceso que está siendo ejecutado actualmente.
- **Function**: El nombre de una función, que se usa para pasarle a `makeContext` cómo se llama la función invocada, de modo que pueda asignar el valor de cuál es la función al contexto que va a ser creado.
- **ExecResult**: El valor resultante de la última evaluación en la ejecución.

De esta forma encapsulamos la información que necesitamos a la hora de ejecutar operaciones en el programa en curso. Porque imaginemos que una operación determinada necesita un nuevo dato: tendríamos que añadir un parámetro a dicha operación y a todas las operaciones que vayan a invocarla, creando un efecto cadena de modificaciones en el código que terminarían por resultar tediosas. De este modo, quizás menos elegante y "oscuro", evitamos sufrir cambios masivos a la hora de incorporar nuevos datos a usar, por parte de operaciones que están en lo más profundo de una cadena de llamadas a operaciones.

Aparte de almacenar toda esta información, ofrece una buena cantidad de operaciones para trabajar con los datos que se encuentran en su interior. Por ejemplo, tenemos funciones para saber si se ha producido un error en ese estado, búsqueda de objetos nodo y proceso, manipulación de las pilas de contextos, acceso y manipulación de valores en la memoria, acceso y manipulación de la expresión que va a ser ejecutada, etcétera.

5.8.3. Las expresiones temporales

Dentro del diagrama de la jerarquía de tipos que hemos mostrado en la figura 5.3, hay un tipo de expresión del que todavía no hemos hablado: `TemporalExpression`. Esta expresión no se puede comprender sin un detalle que en la sección anterior hemos mencionado, que se puede acceder y manipular la expresión que va a ser ejecutada. ¿Por qué es esto importante para este tipo de expresión?

Digamos que tenemos la expresión `3 + 2`, si la evaluamos se queda en `5`. Pero si tenemos la expresión `X = 3 + 2`, en realidad no tenemos una sola expresión, sino que tenemos dos; por un lado está `3 + 2` y por el otro `X = ?`. Se puede ver que hay expresiones "simples" y otras que son "compuestas". Las compuestas tenemos que ir poco a poco reduciéndolas hasta transformarlas en una simple y de ese modo terminar de ejecutar la expresión actual. Entonces, para no utilizar expresiones que sirven para representar el programa, nos creamos una temporal que almacene el resultado de la evaluación de la última subexpresión simple que se haya evaluado. De este modo, tras evaluar `3 + 2`, pasaríamos a tener `X = temp(5)` y ya con eso podríamos ejecutar la asignación.

En la siguiente sección retomaremos el concepto de las expresiones temporales, para hablar sobre la reducción de las expresiones a ejecutar en la implementación del motor de ejecución de Erlang.

5.9. El motor de ejecución de Erlang

Como ya hemos mencionado en más de una ocasión, el lenguaje candidato para probar con nuestro entorno es Erlang. La primera decepción en la implementación —de esta parte de la herramienta— es que en teoría debería incorporar un analizador sintáctico que transforme el código Erlang contenido en una cadena de caracteres al LGAS que hemos definido en el núcleo de Selene. Como tampoco era el objetivo final del trabajo el hacer un analizador sintáctico, se dejó como trabajo futuro.

El caso es que tenemos los módulos `OPERATIONS` y `RUNNER` en nuestra implementación de

la semántica de Erlang. Como ya explicamos sobre los ámbitos, en la sección 5.6, el campo que indica el estado de la ejecución de la expresión actual sigue el esquema indicado en la figura 5.18.

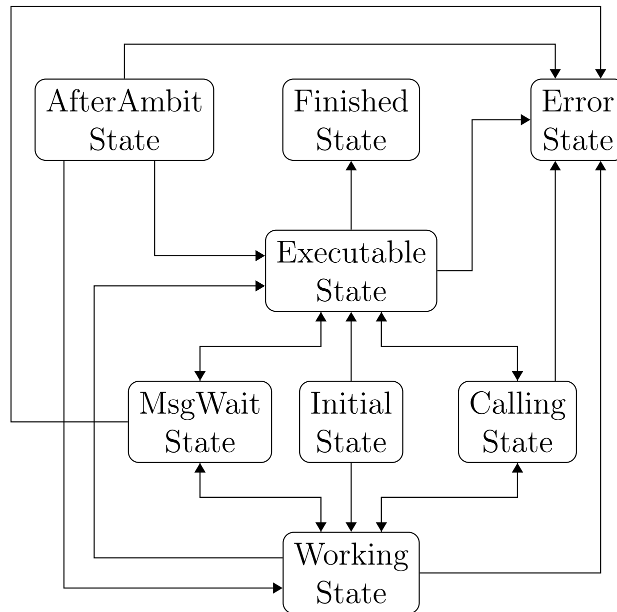

Figura 5.18: *El diagrama de estados de las sentencias.*

Este diagrama, sumado a lo que acabamos de hablar en la sección 5.8.3 sobre las expresiones temporales, nos da una idea sobre el esquema de ejecución de las expresiones en nuestro modelo. Por motivos de separar conceptos, vamos a definir que las sentencias son aquellas expresiones que pueden ser compuestas o no. Así que cada expresión —que conforman la lista de expresiones— del ámbito actual, es una sentencia en nuestro modelo. Cada sentencia puede recorrer una serie de estados, que para el caso de nuestra implementación es el del diagrama de estados que hemos visto antes. Por lo tanto, una sentencia tiene que ir de su estado inicial al estado ejecutable, para poder terminar su evaluación y pasar a la siguiente. Mientras esté en otros estados, como el de "trabajando", se procurará evaluar la primera subexpresión simple que se encuentre en la sentencia, para tomar el resultado de dicha evaluación, meterlo en una expresión temporal y sustituir la expresión que acabamos de evaluar por la temporal, dejando así un poco más reducida la sentencia. El proceso se va

repitiendo, hasta lograr que la sentencia sea una expresión simple y se termine de ejecutar.

`OPERATIONS` contiene operaciones para evaluar los operadores que tiene Erlang, siguiendo las reglas de su semántica. Esta parte claramente no podía ser parte del contenido general, pues no resulta muy complicado encontrar ejemplo de operaciones que funcionan en Erlang de forma distinta en otros lenguajes (como el caso de la suma, que en JavaScript nos permite usarla con cadenas y números, cosa que Erlang no permite).

`RUNNER` contiene las reglas de transición que el motor de reescritura de Maude tomará para ejecutar los programas Erlang que queremos analizar. Un ejemplo de la ejecución, con la versión actual de Selene, es el ejemplo del "servidor echo" del que hablamos en la

```
[Rule: world.init]
[Rule: statement.init]
[Rule: statement.work]
[Rule: statement.exec]
[Rule: statement.init]
[Rule: statement.next]
[Rule: statement.exec]
[Rule: statement.init]
[Rule: process.msg.add]
[Rule: statement.next]
[Rule: statement.exec]
[Rule: statement.init]
[Rule: statement.exec]
[Rule: process.msg.get]
[Rule: statement.work]
[Rule: statement.exec]
[Node: @id(1)] => "EXTERMINATE\n"
[Rule: statement.next]
[Rule: statement.exec]
[Rule: statement.init]
[Rule: process.msg.add]
[Rule: statement.next]
[Rule: statement.exec]
[Rule: statement.init]
[Rule: statement.exec]
[Rule: process.msg.get]
[Rule: statement.work]
[Rule: statement.exec]
[Node: @id(1)] => "ANNIHILATE\n"
[Rule: statement.next]
[Rule: statement.exec]
[Rule: statement.init]
[Rule: process.msg.add]
[Rule: statement.next]
"Final result:" "DESTROY"
[Rule: process.finish]
[Rule: statement.exec]
[Rule: statement.exec]
[Rule: process.msg.get]
[Rule: statement.work]
[Rule: statement.exec]
[Node: @id(1)] => "DESTROY\n"
[Rule: statement.next]
[Rule: statement.init]
[Rule: statement.exec]
[Rule: statement.exec]
[Rule: process.msg.get]
[Rule: statement.exec]
```

Figura 5.19: *Salida por consola de la reescritura del "servidor echo", parte 1.*

```
rewrite in TESTS :
  testworld
result Configuration :
  < 'project : Project | files : @sf("test.erl","-module(test).\n\nserver() ->\
n    register(server, self()),\n    server_loop().\n\nserver_loop() ->\n    rec
eive V ->\n        print(V, \"\\n\"),\n        server_loop(V)\n    end.\n\nwork
er() ->\n    server ! \"EXTERMINATE\",\n    server ! \"ANNIHILATE\",\n    serve
r ! \"DESTROY\".",16)> < 'status : Status | nextIndex : 6,program : @ns(1,
    'test,@fn(3,'server,@cs(3,nil,nil,@op(4,@call,@lt(4,'register),@sq(4,@lt(
    4,'server)@op(4,@call,@lt(4,'self),@sq(4,nil))))@op(5,@call,@lt(5,
    'server_loop),@sq(5,nil))))@fn(7,'server_loop,@cs(7,nil,nil,@rc(8,@cs(8,
    @lt(8,'V),nil,@op(9,@call,@lt(9,'print),@sq(9,@lt(9,'V)@lt(9,"\n")))@op(
    10,@call,@lt(10,'server_loop),@sq(10,nil))),nil)))@fn(13,'worker,@cs(13,
    nil,nil,@op(14,@snd,@lt(14,'server),@lt(14,"EXTERMINATE"))@op(15,@snd,@lt(
    15,'server),@lt(15,"ANNIHILATE"))@op(16,@snd,@lt(16,'server),@lt(16,
    "DESTROY")))))> < @id(1): Node | cin : "",cout :
    "EXTERMINATE\nANNIHILATE\nDESTROY\n",heap : @ms(nil),info : < 'server :
    ProcessAlias | node : 1,process : 1 > > < @id(1): Process | context :(@cx(
    'test 'server,@am(@op(5,@call,@lt(5,'server_loop),@sq(5,nil)),
    @CallingState,nil),@ms(nil),@vl(true))@cx('test 'server_loop,@am(@rc(8,
    @cs(8,@lt(8,'V),nil,@op(9,@call,@lt(9,'print),@sq(9,@lt(9,'V)@lt(9,
    "\n")))@op(10,@call,@lt(10,'server_loop),@sq(10,nil))),nil),@MsgWaitState,
    nil)@am(@op(10,@call,@lt(10,'server_loop),@sq(10,nil)),@CallingState,@vr(
    'V,@sr(1,1,1))),@ms(@mc(1,@vl("EXTERMINATE"))),@vl('ok))@cx('test
    'server_loop,@am(@rc(8,@cs(8,@lt(8,'V),nil,@op(9,@call,@lt(9,'print),@sq(
    9,@lt(9,'V)@lt(9,"\n")))@op(10,@call,@lt(10,'server_loop),@sq(10,nil))),
    nil),@MsgWaitState,nil)@am(@op(10,@call,@lt(10,'server_loop),@sq(10,nil)),
    @CallingState,@vr('V,@sr(2,1,1))),@ms(@mc(2,@vl("ANNIHILATE"))),@vl(
    'ok))@cx('test 'server_loop,@am(@rc(8,@cs(8,@lt(8,'V),nil,@op(9,@call,@lt(
    9,'print),@sq(9,@lt(9,'V)@lt(9,"\n")))@op(10,@call,@lt(10,'server_loop),
    @sq(10,nil))),nil),@MsgWaitState,nil)@am(@op(10,@call,@lt(10,
    'server_loop),@sq(10,nil)),@CallingState,@vr('V,@sr(3,1,1))),@ms(@mc(3,
    @vl("DESTROY"))),@vl('ok))@cx('test 'server_loop,@am(@rc(8,@cs(8,@lt(8,
    'V),nil,@op(9,@call,@lt(9,'print),@sq(9,@lt(9,'V)@lt(9,"\n")))@op(10,
    @call,@lt(10,'server_loop),@sq(10,nil))),nil),@MsgWaitState,nil),@ms(nil),
    @vl(nothing))),messages : nil,newMsgsFlag : false,owner : @id(1)>
```

Figura 5.20: *Salida por consola de la reescritura del "servidor echo", parte 2.*

figura 5.17, cuyo resultado se puede observar en las figuras 5.19 y 5.20.

En la figura 5.19 podemos ver los mensajes que hay en loas atributos `print` en las

reglas del motor de ejecución. Esta salida de texto por consola nos permite ver de forma sencilla cuál ha sido la ruta de cambios de estado que el motor ha realizado para ejecutar un programa. En cuanto a la figura 5.20 vemos el resultado de ejecutar con reescritura el programa de ejemplo. Como se puede observar, la salida final nos muestra el *mundo* que ha quedado resultante tras la ejecución de forma un tanto cruda. Realmente toda la información que aparece es la representación interna de como Selene funciona en Maude, que para un usuario final no sería muy útil el intentar comprenderla.

A continuación vamos a explicar brevemente las reglas de transición que el motor de ejecución de Erlang incorpora en esta versión de la herramienta:

- `world.init`: Esta regla toma el objeto configuración que se encuentra en el *mundo* e inicializa el entorno para poder ejecutar el programa que queremos interpretar con Selene. Al terminar de ejecutarse la regla, el objeto configuración se elimina del *mundo*, para evitar que se pueda volver a repetir la regla de inicialización. Se puede ver un fragmento del código de la regla en la figura 5.21.

- `statement.init`: Esta regla toma un proceso que tenga una sentencia en el estado `@InitialState` para inicializarla, lo que conlleva a que pueda pasar al estado `@WorkingState` si no es una sentencia simple o al estado `@ExecutableState` si es

```
crl [world.init] :
    CO:ConfigObject W:Configuration
    => W4:Configuration
    if W0:Configuration := makeStatus(CO:ConfigObject W:Configuration)
    /\ W1:Configuration := makeProject(W0:Configuration)
    /\ W2:Configuration := makeNodes(W1:Configuration)
    /\ W3:Configuration := makeProcesses(W2:Configuration)
    /\ W4:Configuration := filterConfigObjects(W3:Configuration)
    [print "[Rule: world.init]"] .
```

Figura 5.21: *Código de la regla `world.init`.*

```
crl [statement.init] :
    PO:ProcessObject W:Configuration
    => getWorld(initStatement(ZX:ExecState))
    if validateWorldWithAEIS(W:Configuration, PO:ProcessObject, @InitialState)
    /\ ZX:ExecState := makeExecState(PO:ProcessObject, W:Configuration)
    [print "[Rule: statement.init]"] .
```

Figura 5.22: *Código de la regla `statement.init`.*

```
crl [statement.work] :
    PO:ProcessObject W:Configuration
    => getWorld(workStatement(ZX:ExecState))
    if validateWorldWithAEIS(W:Configuration, PO:ProcessObject, @WorkingState)
    /\ ZX:ExecState := makeExecState(PO:ProcessObject, W:Configuration)
    [print "[Rule: statement.work]"] .
```

Figura 5.23: *Código de la regla `statement.work`.*

simple y se puede ejecutar completamente. Se puede ver un fragmento del código de la regla en la figura 5.22.

- `statement.work`: Esta regla toma un proceso que tenga una sentencia en el estado `@WorkingState` para tomar la primera subexpresión simple, evaluarla y sustituirla por el resultado con una expresión temporal. Entonces se vuelve a comprobar si tiene que seguir en este estado o si por el contrario ya se puede ejecutar del todo. Los únicos motivos para no hacer lo anterior es que la evaluación de un error, que entremos en un `receive` o que acabemos de llamar a otra función (por lo que habrá que inicializar la primera sentencia, por ahorrar una transición al sistema). Se puede ver un fragmento del código de la regla en la figura 5.23.
- `statement.exec`: Esta regla toma un proceso que tenga una sentencia en el estado `@ExecutableState` para terminar su evaluación y ponerla en estado `@FinishedState`.

```
crl [statement.exec] :
    PO:ProcessObject W:Configuration
    => getWorld(execStatement(ZX:ExecState))
    if validateWorldWithAETE(W:Configuration, PO:ProcessObject)
    /\ ZX:ExecState := makeExecState(PO:ProcessObject, W:Configuration)
    [print "[Rule: statement.exec]"] .
```

Figura 5.24: *Código de la regla `statement.exec`.*

```
crl [statement.afcl] :
    PO:ProcessObject W:Configuration
    => getWorld(acscStatement(ZX:ExecState))
    if validateWorldWithAEIS(W:Configuration, PO:ProcessObject, @CallingState)
    /\ ZX:ExecState := makeExecState(PO:ProcessObject, W:Configuration)
    [print "[Rule: statement.afcl]"] .
```

Figura 5.25: *Código de la regla `statement.afcl`.*

Los únicos motivos para no hacer lo anterior es que la evaluación de un error, que entremos en un `receive` o que acabemos de llamar a otra función (por lo que habrá que inicializar la primera sentencia, por ahorrar una transición al sistema). Se puede ver un fragmento del código de la regla en la figura 5.24.

- `statement.afcl`: Esta regla toma un proceso que tenga una sentencia en el estado `@CallingState` para tomar el último resultado generado —como resultado final de la función— para sustituir con ese valor la primera subexpresión simple que en teoría se ha ejecutado. En este caso esa subexpresión se trataría de una llamada a una función. Se puede ver un fragmento del código de la regla en la figura 5.25.

- `statement.afam`: Esta regla toma un proceso que tenga una sentencia en el estado `@AfterAmbitState` para tomar el último resultado generado —como resultado final de la función— para sustituir con ese valor la primera subexpresión simple que en

```
crl [statement.afam] :
    PO:ProcessObject W:Configuration
    => getWorld(acscStatement(ZX:ExecState))
    if validateWorldWithAEIS(W:Configuration, PO:ProcessObject,
                              @AfterAmbitState)
    /\ ZX:ExecState := makeExecState(PO:ProcessObject, W:Configuration)
    [print "[Rule: statement.afam]"] .
```

Figura 5.26: *Código de la regla `statement.afam`.*

```
crl [statement.error] :
    PO:ProcessObject W:Configuration
    => ambitReturn(PO:ProcessObject, true) W:Configuration
    if validateWorldWithHS(W:Configuration, PO:ProcessObject, @ErrorState)
    [print "[Rule: statement.error]"] .
```

Figura 5.27: *Código de la regla `statement.error`.*

teoría se ha ejecutado. En este caso esa subexpresión se trataría de alguna expresión con algún cuerpo que se haya ejecutado (por ejemplo, un `receive`). Se puede ver un fragmento del código de la regla en la figura 5.26.

- `statement.error`: Esta regla toma un proceso que tenga una sentencia en el estado `@ErrorState` para salir del ámbito en el que está actualmente. Al salir del ámbito, se comprobaría el estado en el que estaba el ámbito anterior en la pila, para ver si hay que realizar manejo de errores o no. Se puede ver un fragmento del código de la regla en la figura 5.27.

- `statement.next`: Esta regla toma un proceso que tenga una sentencia en el estado `@FinishedState` para eliminar la sentencia actual, de modo que la ejecución pase a la siguiente en la lista. Se puede ver un fragmento del código de la regla en la figura 5.28.

- `ambit.finish`: Esta regla toma un proceso cuyo ámbito se haya quedado sin sentencias

```
crl [statement.next] :
    PO:ProcessObject W:Configuration
    => nextExpression(PO:ProcessObject) W:Configuration
    if validateWorldWithHEE(W:Configuration, PO:ProcessObject)
    [print "[Rule: statement.next]"] .
```

Figura 5.28: *Código de la regla* ***statement.next****.*

```
crl [ambit.finish] :
    PO:ProcessObject W:Configuration
    => ambitReturn(PO:ProcessObject, true) W:Configuration
    if validateWorldWithHEA(W:Configuration, PO:ProcessObject)
    [print "[Rule: ambit.finish]"] .
```

Figura 5.29: *Código de la regla* ***ambit.finish****.*

que ejecutar para salir del mismo. Al salir del ámbito, se comprobaría el estado en el que estaba el ámbito anterior en la pila, para ver qué curso habría que seguir para continuar con la ejecución. Se puede ver un fragmento del código de la regla en la figura 5.29.

- `call.finish`: Esta regla toma un proceso cuyo contexto se haya quedado sin sentencias que ejecutar, es decir que no queda ningún ámbito en su interior con sentencias a ejecutar, para eliminar el contexto que se acaba de terminar. Además se establecerá como valor final —de la función que había invocado a la función que acaba de terminar— el valor que se acaba de devolver tras terminar la función a la que el contexto acabado estaba asociado. Se puede ver un fragmento del código de la regla en la figura 5.30.

- `process.finish`: Esta regla toma un proceso que ya no le quede ningún contexto que ejecutar —es decir, que el proceso ha terminado— para eliminarlo del *mundo* al

```
crl [call.finish] :
    PO:ProcessObject W:Configuration
    => callReturn(PO:ProcessObject) W:Configuration
    if validateWorldWithHEC(W:Configuration, PO:ProcessObject)
    [print "[Rule: call.finish]"] .
```

Figura 5.30: *Código de la regla `call.finish`.*

```
crl [process.finish] :
    PO:ProcessObject W:Configuration
    => cleanInfo(W:Configuration, PO:ProcessObject)
    if validateWorldWithHEP(W:Configuration, PO:ProcessObject)
    /\ R:Value := getLastResult(PO:ProcessObject)
    /\ B:Bool  := print("Final result:", R:Value)
    [print "[Rule: process.finish]"] .
```

Figura 5.31: *Código de la regla `process.finish`.*

estar ya muerto por inanición. Se puede ver un fragmento del código de la regla en la figura 5.31.

- `process.msg.add`: Esta regla toma un mensaje que esté por el *mundo* para añadirlo a la cola del buzón del proceso destino del mensaje. Se puede ver un fragmento del código de la regla en la figura 5.32.

```
crl [process.msg.add] :
    MSG:Message PO:ProcessObject W:Configuration
    => addMessage(PO:ProcessObject, MSG:Message) W:Configuration
    if validateWorldWithISD(W:Configuration, PO:ProcessObject, MSG:Message)
    [print "[Rule: process.msg.add]"] .
```

Figura 5.32: *Código de la regla `process.msg.add`.*

```
crl [process.msg.get] :
    PO:ProcessObject W:Configuration
    => getWorld(recvStatement(ZX:ExecState))
    if validateWorldWithARIWS(W:Configuration, PO:ProcessObject)
    /\ getNewMsgsFlag(PO:ProcessObject)
    /\ ZX:ExecState := makeExecState(PO:ProcessObject, W:Configuration)
    [print "[Rule: process.msg.get]"] .
```

Figura 5.33: *Código de la regla `process.msg.get`.*

- `process.msg.get`: Esta regla toma un proceso que tenga una sentencia en el estado `@MsgWaitState` y existan nuevos mensajes en la cola del buzón, para que revise si alguno le pueda valer y así continuar la ejecución del programa. Se puede ver un fragmento del código de la regla en la figura 5.33.

Hay que tener en cuenta, que gran parte de la ejecución —de las operaciones que se manejan en las reglas de transición— recae en operaciones que pertenecen a la parte general del núcleo de Selene. Un último detalle sobre esta implementación, es que posiblemente requeriría una refactorización seria, para separar algunas partes susceptibles de ser generales (como es el caso de la regla `world.init`, por ejemplo).

5.10. Comprobación de modelos en Selene

El objetivo, en última instancia, de especificar el lenguaje Erlang —para ser ejecutado bajo Maude— era poder aplicar la comprobación de modelos sobre el mismo. Como hemos ido señalando, la implementación de la especificación no está completa para poder ejecutar cualquier programa existente en Erlang, más bien la herramienta estaría en un estado alpha para dicho aspecto. Pero aun con ello se pudo lograr al menos tener un ejemplo, el "servidor echo" visto en la figura 5.17, que poder ejecutar y sobre el que poder hacer las pruebas iniciales con el comprobador de modelos, para ver la clase de resultados que podíamos obtener.

A pesar de que se pretendía hacer esta parte de forma genérica, al depender del módulo `RUNNER` nos quedamos entre medias de la máquina y del motor de ejecución. Sin duda alguna, como trabajo futuro, podría mejorarse la separación entre la implementación del lenguaje y el núcleo de Selene, trayéndose consigo aquellas partes relacionadas con la comprobación de modelos que fueran independientes de las semánticas específicas del lenguaje.

5.10.1. Por un puñado de propiedades

En la versión actual tenemos dos módulos de propiedades, uno para comprobar temas relacionados con los procesos y el otro para hacer comprobaciones sobre mensajes. Estos dos módulos pueden servir al desarrollador de ejemplo para implementar módulos propios que incorporen nuevas propiedades al sistema.

Las propiedades que tiene el módulo `MCP-PROCESS` son las siguientes:

- `?hasAnyFailed`: Comprueba si en el estado hay algún proceso cuya ejecución haya incurrido en un fallo o error. Por desgracia esta propiedad no contempla que se pudiera luego en el futuro capturar el error, para ello habría que hacer una propiedad que determinara que un error en un proceso ha sido capturado por un `try-catch`. Pero para hacer pruebas iniciales, de cara a ver que la aplicación no falle, es una propiedad útil.

- `?hasFailed`: Como la anterior, pero enfocada a un único proceso. Sin embargo, como para hacerlo tiene que recibir como parámetro el objeto proceso, no resulta muy práctica y habría que darle una vuelta por completo a esta propiedad.

- `?existsProcess`: Comprueba si existe un proceso determinado, ya sea usando un *Qid* o una cadena de caracteres. Esta propiedad solo sirve para poder encontrar procesos que tienen un nombre por alias, algo que sí se puede hacer en Erlang, pero podría no poderse en otros lenguajes.

Las propiedades que tiene el módulo `MCP-MESSAGE` son las siguientes:

- `?existMsgWithOrig`: Comprueba si existe un mensaje con un proceso determinado como origen del mismo.
- `?existMsgWithDest`: Comprueba si existe un mensaje con un proceso determinado como destino del mismo.
- `?existMsgWithBadOrig`: Comprueba si existe un mensaje con un proceso determinado como origen del mismo que no existe en el *mundo*.
- `?existMsgWithBadDest`: Comprueba si existe un mensaje con un proceso determinado como destino del mismo que no existe en el *mundo*.
- `?existMsgWithValue`: Comprueba si existe un mensaje con un valor determinado en el mismo.

Las cuatro primeras operaciones, las que comprueban la existencia o no de un proceso, utilizan el PID para localizar el proceso en cuestión, ya sea con el tipo `Pid` o con dos naturales (uno para el nodo y el siguiente para el proceso). Esta forma de localizar procesos es peliaguda, porque hay que saber de antemano el valor identificador que vaya a tener el proceso, algo que en práctica de lenguajes como Erlang se supone que el PID no es calculable de forma fija para el usuario.

Realmente no son muchas las propiedades que la versión actual de la herramienta tiene. Además en algunos casos se hace complicado poder utilizarlas sin trampear la ejecución de alguna forma. Por ello, como trabajo futuro, sería necesario completar más este área de Selene.

5.10.2. La transformación del contraejemplo

Ya teniendo algunas propiedades, podemos ver en la figura 5.34 un pequeño fragmento del contraejemplo que Maude nos devuelve al aplicar la fórmula `[] (~ ?hasAnyFailed)`, que viene a significa que "siempre no hay ningún error".

```
reduce in TESTS :
  modelCheck(testworld, [] (~ ?hasAnyFailed))
result ModelCheckResult :
  counterexample(...{< 'project : Project | files : @sf("test.erl","-module(tes
t).\n\nserver() ->\n    register(server, self()),\n    server_loop().\n\nserver
_loop() ->\n    receive V ->\n        print(V, \"\\n\"),\n        server_loop(V
)\n    end.\n\nworker() ->\n    server! \"EXTERMINATE\",\n    server ! \"ANNIHI
LATE\",\n    server ! \"DESTROY\".",16)> < 'status : Status | nextIndex : 3,
    program : @ns(1,'test,@fn(3,'server,@cs(3,nil,nil,@op(4,@call,@lt(4,
    'register),@sq(4,@lt(4,'server)@op(4,@call,@lt(4,'self),@sq(4,nil))))@op(5,
    @call,@lt(5,'server_loop),@sq(5,nil))))@fn(7,'server_loop,@cs(7,nil,nil,
    @rc(8,@cs(8,@lt(8,'V),nil,@op(9,@call,@lt(9,'print),@sq(9,@lt(9,'V)@lt(9,
    "\n")))@op(10,@call,@lt(10,'server_loop),@sq(10,nil))),nil)))@fn(13,
    'worker,@cs(13,nil,nil,@op(14,@snd,@lt(14,'server),@lt(14,"EXTERMINATE"))
    @op(15,@snd,@lt(15,'server),@lt(15,"ANNIHILATE"))@op(16,@snd,@lt(16,
    'server),@lt(16,"DESTROY")))))> < @id(1): Node | cin : "",cout : "",heap :
    @ms(nil),info : none > < @id(1): Process | context : @cx('test 'server,
    @am(@op(4,@call,@lt(4,'register),@sq(4,@lt(4,'server)@op(4,@call,@lt(4,
    'self),@sq(4,nil))))@op(5,@call,@lt(5,'server_loop),@sq(5,nil)),
    @InitialState,nil),@ms(nil),@vl(nothing)),messages : nil,newMsgsFlag :
    false,owner : @id(1)> < @id(2): Process | context : @cx('test 'worker,
    @am(@op(14,@snd,@lt(14,'server),@lt(14,"EXTERMINATE"))@op(15,@snd,@lt(15,
    'server),@lt(15,"ANNIHILATE"))@op(16,@snd,@lt(16,'server),@lt(16,"DESTROY")
    ),@InitialState,nil),@ms(nil),@vl(nothing)),messages : nil,newMsgsFlag :
    false,owner : @id(1)>,'statement.init}...,{...,deadlock})
```

Figura 5.34: *Resultado del comprobador de modelos con el "servidor echo".*

El resultado del comprobador de modelos para este caso es demasiado largo como para ponerlo en el documento, así que mostramos solo un fragmento para ver que los contra ejemplos es una tupla, donde la primera componente es una lista de tuplas ⟨*estado, regla*⟩ y la segunda componente es una tupla ⟨*estado final,* ***deadlock***⟩. En el fragmento que mostramos, vemos la segunda transición en la cadena de estados del contraejemplo, ya que la primera transición es la que ejecuta la regla `world.init`.

Como se mostraba en el diagrama de la figura 5.2, el tipo `Configuration` era subtipo de `State`, de esta manera podemos usar la comprobación de modelos sobre nuestro *mundo*. Por ello en el ejemplo se pueden ver los objetos que forman parte del programa que estamos

```
reduce in TESTS :
  modelCheckTransformer(testworld, [] (~ ?hasAnyFailed))
result String :
  "[{\"step\":\"statement.init\",\"node\":1,\"process\":1,\"processes\":[{\"nod
e\":1,\"process\":1,\"index\":4,\"variables\":[],\"messages\":[],\"result\":\"n
ull\"},{\"node\":1,\"process\":2,\"index\":14,\"variables\":[],\"messages\":[],
\"result\":\"null\"}]},{\"step\":\"statement.init\",\"node\":1,\"process\":2,\"
processes\":[{\"node\":1,\"process\":1,\"index\":4,\"variables\":[],\"messages\
":[],\"result\":\"null\"},{\"node\":1,\"process\":2,\"index\":14,\"variables\":
[],\"messages\":[],\"result\":\"null\"}]},{\"step\":\"statement.exec\",\"node\"
:1,\"process\":2,\"processes\":[{\"node\":1,\"process\":1,\"index\":4,\"variabl
es\":[],\"messages\":[],\"result\":\"null\"},{\"node\":1,\"process\":2,\"index\
":14,\"variables\":[],\"messages\":[],\"result\":\"<error>\"}]},{\"step\":\"sta
tement.error\",\"node\":1,\"process\":2,\"processes\":[{\"node\":1,\"process\":
1,\"index\":4,\"variables\":[],\"messages\":[],\"result\":\"null\"},{\"node\":1
,\"process\":2,\"index\":0,\"variables\":[],\"messages\":[],\"result\":\"<error
>\"}]},{\"step\":\"statement.work\",\"node\":1,\"process\":1,\"processes\":[{\"
node\":1,\"process\":1,\"index\":4,\"variables\":[],\"messages\":[],\"result\":
\"null\"}]},{\"step\":\"statement.exec\",\"node\":1,\"process\":1,\"processes\"
:[{\"node\":1,\"process\":1,\"index\":4,\"variables\":[],\"messages\":[],\"resu
lt\":\"true\"}]},{\"step\":\"statement.next\",\"node\":1,\"process\":1,\"proces
ses\":[{\"node\":1,\"process\":1,\"index\":5,\"variables\":[],\"messages\":[],\
"result\":\"true\"}]},{\"step\":\"statement.init\",\"node\":1,\"process\":1,\"p
rocesses\":[{\"node\":1,\"process\":1,\"index\":5,\"variables\":[],\"messages\"
:[],\"result\":\"true\"}]},{\"step\":\"statement.exec\",\"node\":1,\"process\":
1,\"processes\":[{\"node\":1,\"process\":1,\"index\":8,\"variables\":[],\"messa
ges\":[],\"result\":\"null\"}]},{\"step\":\"statement.exec\",\"node\":1,\"proce
ss\":1,\"processes\":[{\"node\":1,\"process\":1,\"index\":8,\"variables\":[],\"
messages\":[],\"result\":\"null\"}]}]"
```

Figura 5.35: *Resultado transformado del comprobador de modelos con el "servidor echo".*

ejecutando, en concreto tenemos el objeto proyecto, seguido del objeto estado, tras ello hay un objeto de tipo nodo, al que le siguen dos objetos de tipo proceso (el primero es el servidor y el segundo el cliente que envía los mensajes).

Con tiempo y dedicación, uno se termina por acostumbrar a los contraejemplos que nos ofrece Maude, pero no se puede negar que resultan un tanto ásperos de cara a manejar estructuras de un tamaño que empiece a ser medio. Por ello en Selene, con la operación

`modelCheckTransformer`, se transforma el contraejemplo en otro tipo de estructura que pueda ser más manejable. Muestra de ello es el resultado —que explicaremos en los siguientes parrafos— de la transformación para nuestro ejemplo en la figura 5.35.

Aparentemente la cosa no ha mejorado con la transformación. En vez de recibir un `ModelCheckResult`, recibimos una `String`. Realmente esta cadena que acabamos de recibir como resultado es un objeto JSON, que si se la pasamos a la función `JSON.parse` de JavaScript se nos queda en lo siguiente:

```
[{"step": "statement.init", "node": 1, "process": 1, "processes": [
    {"node": 1, "process": 1, "index": 4, "variables": [],
     "messages": [], "result": "null"},
    {"node": 1, "process": 2, "index": 14, "variables": [],
     "messages": [], "result": "null"}
]}, {"step": "statement.init", "node": 1, "process": 2, "processes": [
    {"node": 1, "process": 1, "index": 4, "variables": [],
     "messages": [], "result": "null"},
    {"node": 1, "process": 2, "index": 14, "variables": [],
     "messages": [], "result": "null"}
]}, {"step": "statement.exec", "node": 1, "process": 2, "processes": [
    {"node": 1, "process": 1, "index": 4, "variables": [],
     "messages": [], "result": "null"},
    {"node": 1, "process": 2, "index": 14, "variables": [],
     "messages": [], "result": "<error>"}
]}, {"step": "statement.error", "node": 1, "process": 2, "processes": [
    {"node": 1, "process": 1, "index": 4, "variables": [],
     "messages": [], "result": "null"},
    {"node": 1, "process": 2, "index": 0, "variables": [],
     "messages": [], "result": "<error>"}
]}, {"step": "statement.work", "node": 1, "process": 1, "processes": [
    {"node": 1, "process": 1, "index": 4, "variables": [],
     "messages": [], "result": "null"}
]}, {"step": "statement.exec", "node": 1, "process": 1, "processes": [
    {"node": 1, "process": 1, "index": 4, "variables": [],
     "messages": [], "result": "true"}
]}, {"step": "statement.next", "node": 1, "process": 1, "processes": [
    {"node": 1, "process": 1, "index": 5, "variables": [],
     "messages": [], "result": "true"}
]}, {"step": "statement.init", "node": 1, "process": 1, "processes": [
    {"node": 1, "process": 1, "index": 5, "variables": [],
     "messages": [], "result": "true"}
]}, {"step": "statement.exec", "node": 1, "process": 1, "processes": [
    {"node": 1, "process": 1, "index": 8, "variables": [],
```

```
     "messages": [], "result": "null"}
]}, {"step": "statement.exec", "node": 1, "process": 1, "processes": [
    {"node": 1, "process": 1, "index": 8, "variables": [],
     "messages": [], "result": "null"}
]}]
```

Lo que tenemos aquí es una lista de objetos de JavaScript dentro de un array, que representan cada transición de la ejecución. Estos objetos están compuestos por los siguientes campos:

- `step`: La regla de transición ejecutada.
- `node`: El nodo donde está el proceso que fue ejecutado.
- `process`: El proceso que fue ejecutado.
- `processes`: La lista de procesos que había en el *mundo*.

Dentro de la lista de procesos tenemos un array con más objetos que representan los procesos que todavía están vivos en la ejecución. Estos objetos están compuestos por los siguientes campos:

- `node`: El nodo donde está el proceso.
- `process`: El proceso que representa.
- `index`: La línea dentro del código fuente donde se encuentra actualmente la ejecución del proceso.
- `variables`: Las variables del contexto actual de ejecución. Estas son representadas en forma de tupla $\langle$*nombre, valor*$\rangle$.
- `messages`: La cola de mensajes actual del proceso, que contiene una lista de valores recibidos.

- `result`: El resultado final a devolver por la función que se está ejecutando en ese momento.

Con esta información podemos reconstruir la ejecución que el contraejemplo representa, estando pendiente solo de la información relevante de dicha ejecución. Puede ser objeto de discusión si la información recibida por la transformación es relevante o no, pero como mínimo creemos que ofrece lo básico como para poder simular una sesión de depuración.

En cuanto al proceso seguido para construir el objeto JSON, por parte de la operación `modelCheckTransformer`, es el siguiente. Teniendo el resultado de la comprobación de modelos, obtenemos la lista de transiciones del contraejemplo. Por cada elemento de dicha lista, se compara con el siguiente para observar cual ha sido el proceso que ha cambiado durante la transición, de ese modo obtenemos el proceso que acaba de ser ejecutado en la transición. Teniendo el objeto proceso que ha sido modificado, podemos sacar del mismo la información que consideremos pertinente. Durante el proceso de transformación, iremos pasando el estado por diferentes operaciones para poder sacar información que pudiera estar fuera del proceso que hemos recibido (el ejemplo obvio es obtener todos los procesos que hay vivos, como se hace para el campo `processes`). En la figura 5.36 se puede apreciar en pseudo-código el núcleo del algoritmo que sigue la transformación.

```
for i = 0 .. (N-1):
    c_state = counterexample.states_list[i]
    n_state = counterexample.states_list[i+1]
    process = get_changed_process(c_state, n_state)
    json_step = make_step(counterexample.rule[i], process, c_state)
    json_array.append(json_step)
```

Figura 5.36: *Algoritmo que sigue la operación `modelCheckTransformer`.*

No es un proceso realmente complicado. La pega es que hubiera sido interesante añadir parámetros que permitieran modificar la composición final de la transformación, pudiendo decidir qué información recibir o si queremos más detalle o menos. El único intento en esa

dirección es un parámetro que `modelCheckTransformer` tiene para poder indicar una lista de Qids, de modo que solo se seleccionen las reglas de transición que al usuario le interese. Sin embargo, incluso esa parametrización del algoritmo debería ser mejorada y extendida para ofrecer mejor funcionalidad.

5.10.3. La representación visual del contraejemplo

Aunque no se llegó a implementar para esta versión de la herramienta, la idea de utilizar un JSON —como resultado final de la transformación— era buscar una forma estándar de representar nuestro resultado. De este modo podemos coger el objeto JSON y utilizarlo para hacer una representación en HTML o cualquier lenguaje que sea capaz de manejar el formato JSON, que dada su popularidad son innumerables las APIs y frameworks que podemos encontrar para usarlo en aplicaciones.

Volviendo a la representación en HTML, la idea sería coger el código fuente del proyecto y el contraejemplo transformado para recorrer el objeto JSON, paso por paso. En cada paso mostraríamos de forma paralela, a modo de columna, la sección del código donde se encuentra el proceso. Debajo del código se pondría la información que tenemos del contexto de ejecución, que en este caso es el identificador del proceso, la regla de transición que se ha ejecutado (solo para el proceso actual del paso), la lista de variables que hay en la función, los mensajes recibidos y el resultado final que devolverá la función en la que estamos. Mediante los botones de los que dispusiera la interfaz de la representación, se le debería poder dejar al usuario ir hacia delante o hacia atrás en la cadena de ejecución.

Indiscutiblemente, este tema debería ser objeto de trabajo futuro en la herramienta, aunque podría verse afectado por el desarrollo de otras líneas, como es el caso de la línea propuesta para mejorar la parametrización de la transformación.

Capítulo 6

Trabajo futuro

Como ya hemos dejado ver en más de una ocasión, la versión actual de la herramienta dista mucho de estar terminada o de ser usable con proyectos de software reales. Muchos aspectos dentro de Selene requieren de líneas de trabajo futuro, para avanzar su estado a buen puerto. Las líneas imprescindibles para mejorar la herramienta son las siguientes:

1. Sin un analizador sintáctico/transformador, del código del lenguaje origen al lenguaje especificado por la sintaxis de la implementación, la herramienta no puede hacer gran cosa (salvo que el usuario reescriba el código fuente de su aplicación, adaptándolo a la manera especificada dentro de Maude). Bien sea realizando una herramienta externa que realice este paso o usando el metanivel de Maude para parsear el código introducido, es un trabajo necesario para completar la herramienta.

2. Revisar si el uso del LGAS es la buena dirección a seguir o si por el contrario terminará dando más problemas que beneficios. En caso de que se siguiera con su uso, habría que demostrar matemáticamente que la transformación que realizamos se ajusta a un mapeado isomórfico.

3. Parametrizar el núcleo de Selene en base a la semántica definida. Actualmente la separación, entre el núcleo y la semántica del lenguaje Erlang, no es todo lo limpia que cabría de esperar. Por ello se requiere revisar el núcleo y la semántica implementada

—de esta versión— con el fin de separar mejor las partes genéricas de las específicas; y encontrar el modo de usar las específicas como parámetros del núcleo de ejecución.

4. Implementar la semántica de las expresiones del lenguaje que faltan, como son el caso del `if`, el `case-of`, etcétera.

5. Implementar más APIs y BIFs de Erlang como parte del motor de ejecución. Actualmente hay algunas BIFs incluidas, alguna como `print` es inventada y debiera ser sustituida por la función `io:format`; pero queda muchísimas funciones en el aire actualmente, como para poder ejecutar programas de Erlang estándar.

6. Seguir implementando más propiedades y corregir los problemas de las actuales, para poder hacer el mayor número posible de análisis con el comprobador de modelos.

7. Parametrizar el algoritmo de transformación, de modo que permita configurar por parte del usuario cómo quiere que se genere el resultado, ya sea con mayor o menor información.

8. Implementar la visualización del contraejemplo transformado, ya sea en HTML o con una herramienta de escritorio.

Capítulo 7

Conclusiones

Las conclusiones que el autor saca del proyecto son las siguientes. Si bien el proyecto no ha sido completado en base a los objetivos inicialmente marcados, la versión actual del proyecto está en un estadio de prototipo a la espera de ser extendido y revisado para su continuación.

Los motivos que llevaron a no terminar lo que se proponía, son principalmente el haber sido demasiado ambicioso con lo que se quería desarrollar, así como la falta de experiencia como investigador del autor. Hay que tener en cuenta que algunos de los trabajos de investigación sobre comprobación de modelos aplicados a lenguajes han llevado tiempos considerables de investigación y desarrollo. Se trata de un área de trabajo muy específica, en la que todavía queda mucho por investigar al respecto. Sin embargo, la experiencia de haber desarrollado este trabajo final, ha enriquecido al autor enormemente y es de esperar que con la lección aprendida —de los errores que se han cometido— no se vuelva a caer en los mismos fallos que han evitado un mejor resultado.

Tomando en cuenta el primer objetivo, que consistía en desarrollar la herramienta que aplicara la comprobación de modelos sobre un proyecto de código fuente y devolviera el contraejemplo transformado, se han alcanzado con éxito aspectos muy importantes del mismo. A pesar de no haber logrado completar la totalidad de lo que se había propuesto inicialmente, disponemos de un marco de trabajo sobre el que implementar semánticas de lenguajes para su interpretación con el motor de reescritura de Maude, que a su vez nos permite

poder aplicar la comprobación de modelos a los lenguajes que se definan sobre el marco. También se ha desarrollado con éxito un transformador de contraejemplos de Maude a un nuevo formato más compacto y fácil de manejar por otras aplicaciones.

Estos avances, asociado al primer objetivo, quedan relacionados del siguiente modo dentro del plan de trabajo:

- "Implementar la representación de la sintaxis del lenguaje", ha sido completado en gran medida usando el lenguaje genérico de árboles sintácticos. Si bien es cierto que esto requería de un analizador sintáctico/transformador, esperamos que en futuras versiones del proyecto se llegue a desarrollar.

- "Implementar el funcionamiento de la semántica del lenguaje", ha sido completado parcialmente. Aunque no están implementadas todas las construcciones sintácticas de Erlang relacionadas con su semántica, sí que tenemos las más importantes de cara a los aspectos relacionados con la programación concurrente, así como con el paso de mensajes. Las carencias en este aspecto no han impedido que se logre ejecutar algunos ejemplos de prueba, que podrían haber sido programas reales escritos en Erlang (con la salvedad de la función inventada `print`).

- "Implementar las propiedades que serán usadas en la LTL", ha sido completado para poder realizar algunos tests —de comprobación de modelos— sobre los ejemplos escritos en Erlang que habíamos logrado ejecutar en la herramienta.

- "Implementar un algoritmo de transformación para el contraejemplo generado por el comprobador de modelos", ha sido completado con éxito. No por ello cabe conformarse, pues —como ya se ha mencionado con anterioridad— habría sido bastante positivo e interesante que la transformación fuera configurable mediante parámetros. Actualmente existe un nivel de parametrización en el algoritmo que permite filtrar las reglas a tener en cuenta.

En cuanto al segundo objetivo, que consistía en desarrollar la herramienta que visualizara el contraejemplo transformado, no ha sido completado. El motivo para no haber logrado avances —en la representación visual de los contraejemplos— es debido a que existe una fuerte dependencia entre completar el primer objetivo y terminar el segundo, en tanto y cuanto que si por motivos justificados se cambia la estructura del resultado final del contraejemplo —por extensión— se requiere reimplementar la herramienta visualizadora. Por ello, mientras no se alcance una versión madura del primer objetivo, no parece prudente dedicar esfuerzos al segundo objetivo.

Como hemos señalado, el resultado final representa pasos significativos para alcanzar los objetivos marcados. Por ello —en opinión del autor— con mayor tiempo y replanteando algunos aspectos, se podría continuar el desarrollo de la herramienta hasta alcanzar una versión beta que analizara código real de Erlang y no solo códigos de prueba como los que se han estado usando a lo largo del desarrollo.

Capítulo 8

Conclusions

The final conclusions of the author about the project are as follows. The project was not fully complete to achieve the initial goals, but the current version of the project is at a prototype stage waiting to be extended in the future.

The reasons to not finish the initial goals were: being too ambitious with the wanted tool to be developed and the lack of experience of the author as researcher. Taking into account that some of the research works—in this specific area of model checking over programming languages—has been made spending lots of time. Nonetheless, the experience obtained after the development of this Master's Thesis, has improve the knowledge of the author. So with this learned lesson, it is our hope to avoid the same mistakes and get a better result in the future.

Focusing on the first objective, which was to develop the tool to apply model checking on source code projects and return the transformed counterexample, some significant steps have been achieved successfully in its development. While is true that the fully functionality proposed initially has not achieved, it is also true that we have achieved some remarkable features, like having a framework where programming language semantics can be implemented to be interpreted by the rewriting engine of Maude. Having this framework, we can use model checking over programming languages. We also have achieved to develop a counter-example transformation algorithm, to convert Maude counter-examples into something handier.

These steps, related to the first objective, are related with the work plan as follows:

- "Implementing the representation of the language syntax", this point has been mostly completed using the generic syntax trees language. But it requires a parser/transformer tool, which we hope to develop in the future.
- "Implementing the language semantics", this point has been partially completed. Many of the Erlang syntactical constructions have remained unimplemented; but the most important related to concurrent programming and message passing are implemented. Even with these absences it has successfully run some test examples, which could have been written in Erlang real programs (except the invented function `print`).
- "Implementing properties that will be used in LTL", this point has been completed in order to make some tests with model checking, those tests were made over the written examples in Erlang that we had successfully run in the tool.
- "Implementing a transformation algorithm for the counter-example generated by the model checker", this point has been completed. Nonetheless, we can improve how to configure the algorithm with additional parameterization. In the current version we have a first level of parameterization that allows us to select the rules we want to query in the counter-example.

The second objective, which was to develop the tool to visualize the transformed counterexample, it has not been completed. The reason to have not achieved this was the strong dependency between the first objective and the second, because if the structure of the counter-example is changed in a newer version of the first objective, the second objective will need to be reimplemented. So, until we reach a mature version of the first objective, is not wise to use developing efforts in the second objective.

The end result of the project represents significant steps in its progress. In opinion of the author, with more time and rethinking some aspects, the development of the tool could continue to achieve a beta version, in which would be possible to analyze real code of Erlang and not only the test codes used in the development.

Bibliografía

[1] Joe Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, 2010.

[2] Andrei Arusoaie, Dorel Lucanu, Vlad Rusu, Traian-Florin Serbanuta, Andrei Stefanescu, and Grigore Rosu. Language Definitions as Rewrite Theories. In *Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*, pages 97–112, 2014.

[3] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[4] Jan A. Bergstra and Jan Willem Klop. Algebra of Communicating Processes with Abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.

[5] Dirk Beyer. CPAchecker: The Configurable Software-Verification Platform, 2016. Available at `http://cpachecker.sosy-lab.org/`.

[6] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.

[7] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 504–518, 2007.

[8] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Program Analysis with Dynamic Precision Adjustment. In *23rd IEEE/ACM International Conference on*

Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy, pages 29–38, 2008.

[9] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 184–190, 2011.

[10] Philippa J. Broadfoot and A. W. Roscoe. Tutorial on FDR and Its Applications. In *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*, page 322, 2000.

[11] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. Core Erlang 1.0 language specification, november 2000. Available at http://www.it.uu.se/research/publications/reports/2000-030/.

[12] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. Core Erlang 1.0. 3 language specification, 2004. Available at https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf.

[13] NASA Ames Research Center. JPF: Java Pathfinder, 2016. Available at http://babelfish.arc.nasa.gov/trac/jpf.

[14] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[15] Noam Chomsky and Marcel P. Schützenberger. The algebraic theory of context-free languages. *Studies in Logic and the Foundations of Mathematics*, 35:118–161, 1963.

[16] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001.

[17] Edmund M. Clarke and Bernd-Holger Schlingloff. Model Checking. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 1635–1790. Elsevier and MIT Press, 2001.

[18] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. Maude Manual (Version 2.7.1), july 2016. Available at `http://maude.lcc.uma.es/manual271/maude-manual.html`.

[19] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[20] Dominique Clément, Joëlle Despeyroux, Th. Despeyroux, and Gilles Kahn. A Simple Applicative Language: Mini-ML. In *LISP and Functional Programming*, pages 13–27, 1986.

[21] Niels H. M. Aan de Brugh, Viet Yen Nguyen, and Theo C. Ruys. MoonWalker: Verification of .NET Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 170–173, 2009.

[22] Peli de Halleux, Madan Musuvathi, Shaz Qadeer, Sebastian Burckhardt, and Tom Ball. CHESS: Find and Reproduce Heisenbugs in Concurrent Programs, 2016. Available at `https://www.microsoft.com/en-us/research/project/chess-find-and-reproduce-heisenbugs-in-concurrent-programs/`.

[23] ECMA. *Standard ECMA-262 (7th edition / June 2016) — ECMAScript® 2016 Language Specification*. ECMA International, Geneva, Switzerland, june 2016.

[24] Peter Van Eijk and Michel Diaz. *Formal Description Technique LOTOS: Results of the Esprit Sedos Project*. Elsevier Science Inc., New York, NY, USA, 1989.

[25] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL Model Checker and Its Implementation. In *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, pages 230–234, 2003.

[26] Ericsson. wxErlang Reference Manual (Version 1.7), 2016. Available at http://erlang.org/doc/man/wx.html.

[27] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Rosu. Formal Analysis of Java Programs in JavaFAN. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, pages 501–505, 2004.

[28] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Rosu. JavaRL: The Rewriting Logic Semantics of Java, 2006. Available at http://fsl.cs.illinois.edu/index.php/Rewriting_Logic_Semantics_of_Java.

[29] Lars-Åke Fredlund, Clara Benac Earle, and Hans Svensson. McErlang: A model checker for Erlang, 2011. Available at http://babel.ls.fi.upm.es/~fred/McErlang/.

[30] Lars-Åke Fredlund, Clara Benac Earle, and Hans Svensson. McErlang (GitHub), 2011. Available at https://github.com/fredlund/McErlang.

[31] Lars-Åke Fredlund and Hans Svensson. McErlang: a model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 125–136, 2007.

[32] Joseph A. Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, José Meseguer, and Timothy C. Winkler. An Introduction to OBJ 3. In *Conditional Term Rewriting Systems, 1st International Workshop, Orsay, France, July 8-10, 1987, Proceedings*, pages 258–263, 1987.

[33] Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. *Introducing OBJ*, pages 3–167. Springer US, Boston, MA, 2000.

[34] Ganesh Gopalakrishnan and Robert M. Kirby. Practical Formal Verification of MPI and Thread Programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7-10, 2009. Proceedings*, page 8, 2009.

[35] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding Counterexamples with explain. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, pages 453–456, 2004.

[36] Klaus Havelund and Thomas Pressburger. Model Checking Java Programs using Java PathFinder. *STTT*, 2(4):366–381, 2000.

[37] Klaus Havelund and Grigore Rosu. Java PathExplorer: A Runtime Verification Tool, 2001. Available at https://ti.arc.nasa.gov/m/pub-archive/archive/0262.pdf.

[38] Klaus Havelund and Grigore Rosu. Monitoring Java Programs with Java PathExplorer. *Electr. Notes Theor. Comput. Sci.*, 55(2):200–217, 2001.

[39] Fred Hébert. *Learn You Some Erlang for Great Good! (A Beginner's Guide)*. No Starch Press, 2013.

[40] Matthew Hennessy. *Semantics of programming languages - an elementary introduction using structural operational semantics*. Wiley, 1990.

[41] Matthew Hennessy. Semantics of programming languages (CS3017) Course Notes 2012-2013, 2013. Available at https://www.scss.tcd.ie/Matthew.Hennessy/splexternal2015/LectureNotes/Notes14%20copy.pdf.

[42] Thomas A. Henzinger, Rupak Majumdar, Dirk Beyer, and Ranjit Jhala. BLAST:

Berkeley Lazy Abstraction Software Verification Tool, 2008. Available at http://mtc.epfl.ch/software-tools/blast/.

[43] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969.

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

[45] Wilfrid Hodges. *A Shorter Model Theory*. Cambridge University Press, 1997.

[46] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[47] Gerard J. Holzmann. Software model checking with SPIN. *Advances in Computers*, 65:78–109, 2005.

[48] Gerard J. Holzmann. Spin, 2016. Available at http://spinroot.com/.

[49] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation, Second Edition*. Addison-Wesley, 2000.

[50] SRI International. The Maude System, 2016. Available at http://maude.cs.illinois.edu/.

[51] ISO. *ISO 8807:1989 Information technology – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*. International Organization for Standardization, Geneva, Switzerland, february 1989.

[52] ISO/IEC. *ISO/IEC 10514-1:1996 Information technology – Programming languages – Part 1: Modula-2, Base Language*. International Organization for Standardization, Geneva, Switzerland, may 1996.

[53] ISO/IEC. *ISO/IEC 14977:1996 Information technology – Syntactic metalanguage – Extended BNF*. International Organization for Standardization, Geneva, Switzerland, december 1996.

[54] ISO/IEC. *ISO/IEC 10514-2:1998 Information technology – Programming languages – Part 2: Generics Modula-2*. International Organization for Standardization, Geneva, Switzerland, december 1998.

[55] ISO/IEC. *ISO/IEC 10514-3:1998 Information technology – Programming languages – Part 3: Object Oriented Modula-2*. International Organization for Standardization, Geneva, Switzerland, december 1998.

[56] ISO/IEC. *ISO/IEC 14882:2011 Information technology – Programming languages – C++*. International Organization for Standardization, Geneva, Switzerland, september 2011.

[57] ISO/IEC. *ISO/IEC 9899:2011 Information technology – Programming languages – C*. International Organization for Standardization, Geneva, Switzerland, december 2011.

[58] ISO/IEC. *ISO/IEC 14882:2014 Information technology – Programming languages – C++*. International Organization for Standardization, Geneva, Switzerland, december 2014.

[59] Derek Jones. Forms of language specification: Examples from commonly used computer languages, 2007. Available at http://www.knosof.co.uk/vulnerabilities/langconform.pdf.

[60] Robert M. Keller. Formal Verification of Parallel Programs. *Commun. ACM*, 19(7):371–384, 1976.

[61] Richard Kelsey, William Clinger, and Jonathan Rees. Revised 5 Report on the Algorithmic Language Scheme, 7.2 Formal semantics, february

1998. Available at http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-10.html#%_sec_7.2.

[62] Donald E. Knuth. Backus Normal Form vs. Backus Naur Form. *Commun. ACM*, 7(12):735–736, 1964.

[63] Peter Linz. *An introduction to formal languages and automata (4. ed.)*. Jones and Bartlett Publishers, 2006.

[64] José Meseguer and Grigore Rosu. The rewriting logic semantics project. *Theor. Comput. Sci.*, 373(3):213–237, 2007.

[65] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[66] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[67] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, USA, 1997.

[68] Madan Musuvathi and Shaz Qadeer. CHESS: Systematic Stress Testing of Concurrent Software. In *Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers*, pages 15–16, 2006.

[69] Madan Musuvathi, Shaz Qadeer, and Tom Ball. CHESS: A systematic testing tool for concurrent software. Technical report, Technical Report MSR-TR-2007-149, Microsoft Research, November 2007.

[70] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent

Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 267–280, 2008.

[71] Martin R. Neuhäußer and Thomas Noll. Abstraction and Model Checking of Core Erlang Programs in Maude. *Electr. Notes Theor. Comput. Sci.*, 176(4):147–163, 2007.

[72] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications - a formal introduction*. Wiley professional computing. Wiley, 1992.

[73] University of Utah. ISP (In-situ Partial Order): A dynamic verifier for MPI Programs, 2016. Available at http://formalverification.cs.utah.edu/ISP-release/.

[74] C. E. Pitts. Parallel processing support: so what is a "Heisenbug" anyway? In *Proceedings of the 17th Annual ACM SIGUCCS Conference on User Services, Bethesda, Maryland, USA, 1989*, pages 237–242, 1989.

[75] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[76] Amir Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977.

[77] William Pugh. The Java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6):445–455, 2000.

[78] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[79] Adrián Riesco. Using Big-Step and Small-Step Semantics in Maude to Perform Declarative Debugging. In *Functional and Logic Programming - 12th International Sym-*

posium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings, pages 52–68, 2014.

[80] Adrián Riesco, Irina Mariuca Asavoae, and Mihail Asavoae. A Generic Program Slicing Technique Based on Language Definitions. In *Recent Trends in Algebraic Development Techniques, 21st International Workshop, WADT 2012, Salamanca, Spain, June 7-10, 2012, Revised Selected Papers*, pages 248–264, 2012.

[81] Adrián Riesco, Irina Mariuca Asavoae, and Mihail Asavoae. Memory Policy Analysis for Semantics Specifications in Maude. In *Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected Papers*, pages 293–310, 2015.

[82] Adrián Riesco, Alberto Verdejo, Rafael Caballero, and Narciso Martí-Oliet. Declarative Debugging of Rewriting Logic Specifications. In *Recent Trends in Algebraic Development Techniques, 19th International Workshop, WADT 2008, Pisa, Italy, June 13-16, 2008, Revised Selected Papers*, pages 308–325, 2008.

[83] Adrián Riesco, Alberto Verdejo, and Narciso Martí-Oliet. A Complete Declarative Debugger for Maude. In *Algebraic Methodology and Software Technology - 13th International Conference, AMAST 2010, Lac-Beauport, QC, Canada, June 23-25, 2010. Revised Selected Papers*, pages 216–225, 2010.

[84] Adrián Riesco, Alberto Verdejo, Narciso Martí-Oliet, and Rafael Caballero. A Declarative Debugger for Maude. In *Algebraic Methodology and Software Technology, 12th International Conference, AMAST 2008, Urbana, IL, USA, July 28-31, 2008, Proceedings*, pages 116–121, 2008.

[85] Adrián Riesco, Alberto Verdejo, Narciso Martí-Oliet, and Rafael Caballero. Declarative debugging of rewriting logic specifications. *J. Log. Algebr. Program.*, 81(7-8):851–897, 2012.

[86] Grigore Rosu. Programming Language Design (CS422, Fall 2006), 2006. Available at http://fsl.cs.illinois.edu/index.php/CS422_-_Programming_Language_Design_(Fall_2006).

[87] Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebr. Program.*, 79(6):397–434, 2010.

[88] Theo Ruys. MoonWalker: model checking .NET applications, 2016. Available at http://fmt.cs.utwente.nl/tools/moonwalker/.

[89] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[90] Traian-Florin Serbanuta and Grigore Rosu. K-Maude: A Rewriting Based Tool for Semantics of Programming Languages. In *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*, pages 104–122, 2010.

[91] Traian-Florin Serbanuta, Grigore Rosu, and José Meseguer. A rewriting logic approach to operational semantics. *Inf. Comput.*, 207(2):305–340, 2009.

[92] Kenneth Slonneger and Barry L. Kurtz. *Formal syntax and semantics of programming languages - a laboratory based approach.* Addison-Wesley, 1995.

[93] Sarvani S. Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic Verification of MPI Programs with Reductions in Presence of Split Operations and Relaxed Orderings. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 66–79, 2008.

[94] Alberto Verdejo and Narciso Martí-Oliet. Executable structural operational semantics in Maude. *J. Log. Algebr. Program.*, 67(1-2):226–293, 2006.

[95] Farn Wang. REDLIB for the Formal Verification of Embedded Systems. In *Leveraging Applications of Formal Methods, Second International Symposium, ISoLA 2006, Paphos, Cyprus, 15-19 November 2006*, pages 341–346, 2006.

[96] Farn Wang. REDLIB (SourceForge), 2016. Available at https://sourceforge.net/projects/redlib/.

[97] Mark Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981.*, pages 439–449, 1981.

[98] Mark Weiser. Program Slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[99] Glynn Winskel. Lecture Notes on Denotational Semantics for Part II of the Computer Science Tripos, 2005. Available at http://www.cl.cam.ac.uk/~gw104/dens.pdf.

[100] wxWidgets. wxWidgets, 2016. Available at https://www.wxwidgets.org/.