UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño y desarrollo de una técnica y sistema de especialización incremental en Maude

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Daniel Galán Pascual
Tutor: María Alpuente Frasnedo
Director experimental: Julia Sapiña Sanchis

Curso 2020-2021

Agradecimientos

Este proyecto ha sido posible gracias a la atenta guía de las tutoras María Alpuente y Julia Sapiña, cuyo compromiso encomiable con el aprendizaje y el desarrollo personal ha permitido al autor la exploración de la amplia y estimulante área de los métodos formales. La dirección del trabajo experimental por parte de Julia Sapiña, autora de la implementación original de Presto, ha sido un apoyo invalorable para la realización del proyecto. Por otro lado, la dirección de María Alpuente ha posibilitado la correcta comprensión de las bases teóricas y formales subyacentes. Un agradecimiento adicional para el profesor Santiago Escobar por numerosos consejos y discusiones sobre Maude y sobre la herramienta Presto.

Resum

L'estat de l'art en el disseny d'eines d'especialització automàtica de codi està enfocat a la construcció d'eines monolítiques, on es prevalen l'automatització completa i l'eficiència enfront de la usabilitat i la facilitat de depuració de les optimitzacions generades.

L'objectiu d'aquest treball és doble. En primer lloc es proposa un nou esquema d'especialització automàtica de programes multi-paradigma escrits en el llenguatge Maude introduint una modalitat d'especialització incremental, permetent així encapsular els components clau de l'especialització al nivell de granularitat desitjat i refinar progressivament les especialitzacions realitzades. En segon lloc, l'esquema d'especialització incremental resultant s'implementa en una eina pràctica que suporta l'especialització versàtil i eficient de codi Maude.

La component de incrementalitat s'introdueix de manera ortogonal als mòduls funcionals o de sistema, donant suport a l'especialització gradual de programes complets i fent possible corregir qualsevol optimització defectuosa que es derive d'una violació dels requisits d'especialització o de decisions d'especialització inadequades.

Finalment, l'eina desenvolupada es complementa amb un mòdul de generació de casos de prova que facilita l'avaluació experimental d'optimitzacions progressives de programari com ara verificadors algorítmics de programes per a les lògiques CTL i LTL, permetent especialitzar-los a instàncies de fórmules que eren intractables originalment.

Paraules clau: mètodes formals, optimització automàtica de programes, avaluació parcial, comprovació de models, Maude

Resumen

El estado del arte en el diseño de herramientas de especialización automática de código está enfocado a la construcción de herramientas monolíticas, donde se priman la automatización completa y la eficiencia frente a la usabilidad y la facilidad de depuración de las optimizaciones generadas.

El objetivo de este trabajo es doble. En primer lugar se propone un nuevo esquema de especialización automática de programas multi-paradigma escritos en el lenguaje Maude que introduce una modalidad de especialización incremental, permitiendo así encapsular los componentes clave de la especialización al nivel de granularidad deseado y refinar progresivamente las especializaciones realizadas. En segundo lugar, el esquema de especialización incremental resultante se implementa en una herramienta práctica que soporta la especialización versátil y eficiente de código Maude.

La componente de incrementalidad se introduce de forma ortogonal a los módulos funcionales o de sistema, dando soporte a la especialización gradual de programas completos y haciendo posible corregir cualquier optimización defectuosa que se derive de una violación de los requisitos de especialización o de decisiones de especialización inadecuadas.

Finalmente, la herramienta desarrollada se complementa con un módulo de generación de casos de prueba que facilita la evaluación experimental de optimizaciones progresivas de herramientas software tales como verificadores algorítmicos de programas para las lógicas CTL y LTL, permitiendo especializarlos a instancias de fórmulas que eran intratables originalmente.

Palabras clave: métodos formales, optimización automática de programas, evaluación parcial, comprobación de modelos, Maude

Abstract

The state of the art in the design of automated program specialization tools is populated with monolithic tools, where usability and debugging of the optimizations is traded for efficiency and full automation.

The goal of this work is twofold. First, the core specialization scheme for multi-paradigm Maude programs is endowed with a new incremental modality that allows the key specialization components to be encapsulated at the desired granularity level to facilitate progressive refinements of the specialization. Second, the proposed incremental specialization scheme is implemented in a practical optimization tool that provides versatile and efficient Maude code specialization.

The new incremental modality is orthogonal to the use of functional or system modules so that gradual specialization of whole programs is supported, making it easier to correct any faulty optimization that may result from a violation of the specialization requirements or from fixing inadequate specialization criteria.

Finally, the tool will be endowed with a novel capability for test case generation that facilitates the experimental evaluation of progressive optimizations of non-trivial software tools such as the algorithmic program verifiers for CTL and LTL, allowing them to be specialized to formulae that cannot be handled by the original algorithms.

Key words: formal methods, automated program optimization, partial evaluation, model-checking, Maude

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Índice de fragmentos de código

CAPÍTULO 1
Introducción

El desarrollo de *software* es una actividad transversal que usualmente requiere del análisis de dominios complejos y de la formalización y estructuración de información y entidades del mundo real que, a veces, pueden resultar aparentemente caóticas o plantear situaciones difíciles de predecir. Ya desde los inicios de la ingeniería del software, la necesidad de producir programas de calidad que sean capaces de funcionar de forma correcta, eficiente, segura, atractiva, entendible e integrable ante situaciones potencialmente complejas ha dado pie al origen y desarrollo de técnicas formales y semiformales que buscan otorgar control y predictibilidad al fino equilibro entre coste, tiempo de desarrollo y calidad.

La ingeniería del *software* busca la aplicación de prácticas bien entendidas y reproducibles para la solución de problemas mediante soluciones *software* de calidad durante el análisis, diseño, desarrollo y posterior mantenimiento a lo largo de su ciclo de vida [1]. Una de las muchas áreas de esta amplia disciplina se centra en los métodos formales, que utilizan las formalizaciones otorgadas por la lógica formal y las matemáticas en unión con herramientas *software* para realizar un abanico de posibilidades extremadamente amplio, como verificar el comportamiento de cierto programa respecto a una especificación atendiendo a la semántica formal del lenguaje de programación utilizado, generar código fuente de forma automática respecto a conocimiento estructurado como pueden ser modelos o datos de entrenamiento, o transformar programas de forma automática garantizando la conservación o el intercambio controlado de ciertas propiedades [2].

El lenguaje de programación Maude es un lenguaje multiparadigma que integra gran cantidad de características que permiten escribir los programas como especificaciones verificables y otorga un marco metalógico en el que poder ejecutar y desarrollar una gran variedad de métodos formales de alta complejidad y alto rendimiento [3]. El lenguaje integra características de la programación lógica, funcional, concurrente y orientada a objetos, y su ecosistema ha visto crecer una gran cantidad de herramientas relacionadas con los métodos formales. Entre otras, incluye funcionalidades para la comprobación de modelos o *model checking*, la verificación de protocolos criptográficos o la optimización automática de programas [4].

Una de estas herramientas es un *especializador* automático llamado Presto [5]. Esta herramienta consiste en una aplicación web en la que se puede introducir un programa escrito en Maude para restringir su comportamiento de forma controlada y así facilita poder incrementar su rendimiento temporal, logrando optimizaciones automáticas de forma versátil y eficiente que pueden resultar considerables con los parámetros adecuados [6, 7]. El desarrollo de este trabajo orbita principalmente alrededor de esta potente herramienta, cuyo diseño se revisa en profundidad para añadir un nuevo modo de fun-

cionamiento incremental. Además, se crea una herramienta auxiliar que facilita la evaluación experimental de optimizaciones progresivas para Maude, permitiendo automatizar la prueba de las especializaciones obtenidas con Presto (u otras herramientas) para programas reales.

1.1 Motivación

El estado del arte de las herramientas de especialización automática de código como Presto [5], ECCE [8] o Logen [9] está mayoritariamente enfocado a crear herramientas capaces de producir código altamente eficiente donde prima la automatización completa, la eficiencia de desempeño o la corrección funcional. Sin embargo, resulta deseable facilitar la inclusión de características adicionales y mejoras que aumenten la facilidad de uso de estas herramientas, así como la posibilidad de soportar procesos de depuración o prueba de la especialización obtenida, o la capacidad de obtener un mayor control sobre la granularidad del proceso, entre otras.

Este trabajo comenzó durante el transcurso de una beca de colaboración del autor en el instituto *Valencian Research Institute for Artificial Intelligence* (VRAIN) de la *Universitat Politècnica de València* con el grupo que ha desarrollado el especializador automático Presto. El trabajo se centró inicialmente en la especialización de programas Maude no triviales, incluyendo un comprobador de modelos o *model checker* basado en lógica CTL desarrollado por el autor de este trabajo. Dichas exploraciones permitieron descubrir resultados no tratables por otras herramientas actuales relacionadas con la comprobación de modelos, evaluando simbólicamente fórmulas CTL con variables que actúan como "agujeros" lógicos a valores lógicos de verdadero y falso para ciertas entradas específicas, permitiendo discernir en una ejecución cuáles serían los valores de verdad resultantes para un conjunto de fórmulas parcialmente instanciadas en función de las entradas dadas que se ajusten al patrón dado.

Sin embargo, ciertas sesiones de especialización con Presto se vieron frustradas o incompletas debido a que decisiones de especialización inadecuadas o limitaciones algorítmicas causaron que el proceso terminara de forma no satisfactoria, normalmente porque se alcanzaba el límite de tiempo establecido, para evitar colapsar de trabajo al servidor y mitigar posibles ataques de denegación de servicio al tratarse Presto de una aplicación pública. En dichos casos, la herramienta mostraba un mensaje de error que indicaba la imposibilidad de completar el proceso, derivando su aproximación monolítica en un proceso de caja negra, y sin otorgar información de los resultados parciales que se pueden haber obtenido antes de alcanzar el límite de tiempo máximo.

La experiencia descrita condujo a plantear la conveniencia de rediseñar la herramienta para permitir las especializaciones *incrementales*, pudiendo ajustar la granularidad deseada y permitiendo el análisis de los pasos intermedios realizados por el algoritmo, teniendo un efecto positivo en la usabilidad de la herramienta al permitir al usuario obtener refinamientos progresivos y analizar así con mayor prontitud las decisiones de especialización tomadas y sus efectos.

De forma adicional, la comprobación de las decisiones de especialización o de la eficiencia del proceso requiere en ciertas situaciones la prueba experimental de los resultados obtenidos mediante la ejecución de llamadas, expresadas como términos Maude, que pueden tener requisitos de tamaño o complejidad no despreciables. Por ejemplo, realizar una prueba de rendimiento o *benchmark* de un programa de gran tamaño o considerable complejidad (como el comprobador de modelos CTL anteriormente mencionado), o comprobar si ciertos operadores son soportados para unos términos de especificación complejos, requieren términos de entrada de tamaño significativo o que tengan un buen

cubrimiento de las funciones ofrecidas por el programa. En el caso específico del comprobador de modelos CTL, generar de forma manual y reiterativa fórmulas lógicas CTL o *estructuras de Kripke* (grafos de estados) de tamaño significativo supone en un coste de tiempo y humano no desdeñable ni deseable.

Por lo tanto, para completar este trabajo se plantea una utilidad que permita la generación automática de términos Maude que puedan servir para realizar de forma automática pruebas experimentales en programas no triviales, incluyendo (pero no limitado a) los programas especializados por Presto.

1.2 Objetivos

El objetivo de este trabajo es doble. En primer lugar, se plantea analizar y comprender el algoritmo de especialización utilizado por Presto y la literatura relacionada con el objetivo de poder diseñar y desarrollar un nuevo esquema de especialización *incremental* que permita aplicar pasos parciales de una granularidad ajustable y den soporte a refinamientos progresivos relacionados con las entradas y las decisiones del proceso de especialización. Además, se implementa dicho esquema incremental de forma práctica en el especializador automático Presto, realizando la correspondiente ingeniería inversa y extensión de la implementación existente, aumentado sus capacidades con las facilidades mencionadas de forma retrocompatible y armónica con el diseño existente.

En segundo lugar, se plantea desarrollar un módulo adicional capaz de generar en masa términos Maude que respeten los tipos y condiciones requeridos por el programa de interés y actúen como entradas de pruebas experimentales, donde prima que el tamaño general del término y la complejidad en anchura y profundidad del árbol formado por los operadores que lo representan sean adecuados y significativos.

1.3 Estructura de la memoria

En primer lugar, la Parte I incluye los capítulos que introducen y desarrollan brevemente la literatura previa de los conceptos tratados y métodos formales implicados, explicando también la motivación y el funcionamiento de la herramienta Presto e indagando en el algoritmo y la arquitectura de implementación que lo hacen funcionar.

En segundo lugar, la Parte II describe el diseño y el desarrollo realizados para lograr los objetivos descritos, relatando primero la construcción del nuevo esquema de especialización incremental y siguiendo entonces con la revisión y la extensión de la implementación de Presto. A continuación, se presenta el metagenerador de casos de prueba desarrollado desde su concepción hasta su implementación final. Finalmente, siguen las comentarios finales en una sección de conclusiones y la bibliografía utilizada.

Una sección de apéndices acompaña el texto con fragmentos clave del código fuente e información técnica asociada.

1.4 Relación con los estudios cursados

Este trabajo profundiza en el área de los métodos formales de la ingeniería del software, introducida en dos asignaturas del Grado de Ingeniería Informática en la *Universitat Politècnica de València*: *Métodos Formales Industriales*, de tercer curso, y *Análisis, Validación y Depuración*, de cuarto curso.

Como lenguajes de implementación se utilizan principalmente los lenguajes Maude y Java, ambos estudiados en el Grado e introducidos en las asignaturas de *Métodos Formales Industriales* e *Introducción a la Programación*, respectivamente. Al haberse desarrollado una aplicación web con arquitectura cliente-servidor, ha sido fundamental el conocimiento impartido en asignaturas como *Tecnología de Sistemas de Información en la Red* e *Integración e Interoperabilidad*. Los patrones y buenas prácticas de programación explorados en *Diseño de Software*, junto con la semántica formal de los lenguajes de programación lógicos y funcionales estudiada en *Lenguajes, Tecnologías y Paradigmas de la Programación* han otorgado la base teórica que ha permitido la manipulación automática de construcciones del lenguaje Maude y la correcta utilización de sus propiedades metalingüísticas y reflexivas.

1.5 Colaboraciones

Este trabajo se ha desarrollado en el contexto del programa de *becas de colaboración* dentro del grupo *Extensions of Logic Programming* (ELP) de la *Universitat Politècnica de València*, quien lo ha hecho posible cediendo el código fuente de la herramienta Presto para su análisis y ampliación.

Parte I

Estudio del contexto y trabajo previo

CAPÍTULO 2
Contexto tecnológico

En este capítulo se introducen con más detalle los conceptos tecnológicos fundamentales de la especialización de programas y la comprobación de modelos, así como las herramientas y utilidades prácticas que constituyen un prerrequisito tanto para el análisis del problema que se pretende abordar como para la implementación de la solución. El objetivo es aportar una visión de alto nivel del estado del arte del área de estudio de la verificación y transformación automática de *software* en el ámbito de los métodos formales.

En primer lugar, se introduce brevemente la técnica de comprobación de modelos, un método formal que permite probar la corrección de un programa informático. Seguidamente, se introduce el lenguaje de programación Maude, mostrando una visión general de sus características y relacionándolas con las técnicas y prácticas del campo de la verificación formal y, específicamente, con la comprobación de modelos. A continuación, se presenta la especialización de programas como método de optimización automática de programas y, aunque este concepto trasciende al lenguaje utilizado, se relaciona con Maude con el objetivo de poder ilustrarlo con ejemplos específicos. Finalmente, se introduce brevemente la técnica de ejecución simbólica mediante estrechamiento (*narrowing*), utilizada por el algoritmo EqNPE que se describe en los capítulos siguientes.

2.1 Comprobación de modelos

La comprobación o verificación de modelos, también conocida por su nombre en inglés *model checking*, es una técnica de verificación formal que comprueba si cierto sistema modelado como una máquina de estados satisface una serie de propiedades descritas en una especificación. Esta técnica se suele utilizar para comprobar que los sistemas *software* y *hardware* cumplen ciertos requisitos de seguridad (*safety*) o vivacidad (*liveness*), como la incapacidad de alcanzar estados indeseables o la ausencia de bloqueos [10]. Por ejemplo, puede utilizarse para comprobar que el *software* de un ascensor no le permita moverse con sus puertas abiertas, lo que sería peligroso.

La comprobación de modelos es realizada por un programa *comprobador de modelos* o *model checker*, el cual toma dos entradas: un modelo, que representa el comportamiento real del sistema a evaluar; y una especificación, que describe formalmente el comportamiento esperado describiendo las propiedades deseables o indeseables. Tras comparar el modelo introducido con las propiedades especificadas, el programa emitirá como salida una respuesta binaria: "sí", si el modelo satisface la especificación, o "no" en caso contrario. En caso de obtenerse una salida negativa, se entrega también un *contraejemplo*, que es una traza mostrando por qué el modelo ha violado la especificación y respecto a qué propiedades específicas, permitiendo al programador evaluar y desarrollar las acciones

correctivas a tomar. La comprobación de modelos puede verse, por lo tanto, como un problema de decisión.

El modelo a comprobar se representa con una estructura de Kripke, que es un grafo etiquetado que puede verse como una máquina de estados finita. Con el objetivo de aumentar la ergonomía, ciertos comprobadores de modelos permiten describir el modelo ya sea con un lenguaje específico que más tarde es traducido a una estructura de Kripke, o bien tomando como entrada directamente un lenguaje de programación de propósito general, como C o Java. La estructura final contiene los estados posibles que puede tomar el sistema, etiquetados con propiedades que se cumplen en dichos estados y relacionados por transiciones.

Por otro lado, la especificación consiste en una serie de propiedades expresadas formalmente como fórmulas de alguna lógica o lenguaje formal. Son muy conocidas y utilizadas algunas lógicas temporales como la *linear temporal logic* (LTL) y la *computation tree logic* (CTL), que particularizan la lógica modal para expresar proposiciones y razonar en relación con el tiempo, extendiendo la expresividad mucho más allá de lo que permite la lógica de primer orden tradicional. Ejemplos de esta capacidad serían "ocurrirá A hasta que ocurra B", "siempre ocurre A", o bien "si ocurre A, acabará ocurriendo B".

2.2 El lenguaje Maude

Maude es un lenguaje de programación declarativo, multiparadigma y de alto rendimiento basado en la lógica de reescritura (RWL). Este lenguaje presenta rasgos típicos de la programación funcional y lógica, utilizando ecuaciones y reglas de reescritura para describir programas y especificaciones de propósito general.

La lógica de reescritura es una lógica tipada (*many-sorted*) que permite expresar la evolución de los sistemas concurrentes de forma simple, expresiva y muy efectiva, en forma de términos que definen estados, ecuaciones que simplifican dichos estados y reglas que definen cómo se pueden transformar estos estados de forma aislada e independiente. Dadas las propiedades de esta lógica, Maude resulta especialmente útil como marco semántico general y metalógico para definir la semántica ejecutable de un gran rango de lenguajes y sistemas formales, incluyendo la representación y ejecución de otras lógicas y destacando en las computaciones concurrentes orientadas a objetos. Además, Maude es reflexivo de una forma natural y ha visto crecer a su alrededor un gran número de herramientas que utilizan sus potentes propiedades metalingüísticas y de metaprogramación, sirviendo de entorno para herramientas tan variadas como demostradores de teoremas [11], depuradores y optimizadores de código [12] o analizadores de protocolos criptográficos [13], entre muchas otras.

A continuación se presenta una breve introducción al lenguaje Maude en su versión 3.1 [3], que es la más reciente en el momento de escribirse esta memoria. El sitio web de Maude se encuentra disponible en `http://maude.cs.illinois.edu`.

2.2.1. Teorías ecuacionales

Maude permite dos construcciones en el nivel global, llamadas *módulos funcionales* y *módulos de sistema*.[1] Los *módulos funcionales* son agrupaciones lógicas de definiciones de operadores y ecuaciones, describiendo un programa de forma convencional y comparable a otros lenguajes. Es sin embargo en los *módulos de sistema* donde vemos las primeras

[1] Maude permite también módulos orientados a objetos que se traducen internamente a módulos funcionales o de sistema, y que por simplicidad no son considerados explícitamente en este trabajo.

diferencias, ya que estos permiten también la definición de *reglas de reescritura* que introducen el no determinismo y la concurrencia que caracterizan a la lógica de reescritura. Un "programa" escrito en Maude es equivalente a una *teoría ecuacional* cuando no hace uso de reglas de reescritura, y *teoría de reescritura* en caso contrario.

Los módulos pueden organizarse jerárquicamente a través de inclusiones con las palabras reservadas `include`, `extending` y `protecting`.

Una teoría ecuacional $\mathscr{E}$ está formada por uno o más módulos funcionales. Cada uno de ellos puede definir una serie de tipos de datos o "géneros" (*sorts*), *operadores* y *ecuaciones*.

El siguiente fragmento de código Maude muestra un módulo funcional que describe los números naturales y su operación de adición, utilizando los axiomas de Peano:

```
fmod NAT is
    sort Nat .

    op 0 : -> Nat [ctor] .
    op s : Nat -> Nat [ctor] .
    op _+_ : Nat Nat -> Nat .

    eq 0 + N:Nat = N:Nat .
    eq s(N:Nat) + M:Nat = s (N:Nat + M:Nat) .
endfm
```

Fragmento 2.1: Suma de números naturales

Tipos: *sorts* y *kinds*

Los *sorts* son análogos a los tipos de otros lenguajes de programación: permiten clasificar a los términos en dominios diferenciados sobre los cuales se aplican diferentes operadores, en lugar de tratar todas estas construcciones como una colección homogénea en un mismo universo. Se definen con la palabra reservada `sort`. Si, por ejemplo, decimos que un término es del *sort* `Nat`, entonces decimos que este representa un número natural. De igual forma, el *sort* `NeNatList` representa una lista no vacía de números naturales, `Bool` un valor booleano, `String` una cadena de caracteres, y así sucesivamente.

Los `sorts` pueden tener relaciones de subtipado, definiendo un grafo acíclico de órdenes parciales que se puede imaginar como un diagrama de Hasse, donde algunos `sorts` puede tener `subsorts` y así recursivamente. Esto constituye una forma de polimorfismo de forma similar a como se vería en otros lenguajes, donde un parámetro de tipo `NatList` puede aceptar un argumento del *sort* más específico `NeNatList` (*non-empty Nat List*).

En Maude la palabra "tipo" tiene una connotación especial, más general que la de *sort*, al incluir también los llamados *kinds*. Brevemente, un *kind* se representa como uno o más *sorts* envueltos con los símbolos de corchete (por ejemplo, `[Nat]` en lugar de `Nat`), e incluye en su clasificación a dicho *sort* y a sus "comparables" respecto a las relaciones de subtipado[2]. Esto incluye a las expresiones parcialmente evaluadas, por lo que esta característica se utiliza normalmente para la gestión de errores: una función que devuelve `[Nat]` no funciona o no está definida para todas la entradas; o devuelve un `Nat` o "falla", quedándose parcialmente sin evaluar.

[2]Más formalmente, representa al maximal de los *sorts* indicados, según el grafo de las relaciones de *subsorts*.

Operadores

Un operador, introducido con la palabra reservada `op`, es análogo al bien conocido concepto de función: se trata de un símbolo o nombre que, aplicado a una serie de términos de cierto *sort*, produce un nuevo término de un *sort* que puede ser diferente. Si nos limitamos a hablar de teorías ecuacionales, un operador de aridad cero corresponde, como es esperable en los paradigmas de la programación relacionados, a una constante. En el ejemplo anterior se puede observar que se ha definido el símbolo `0` como una constante de los números naturales, el operador sucesor como el símbolo `s`, y la suma como el símbolo binario infijo +.

Los operadores son *constructores* (de datos) cuando su uso no resulta en una transformación por parte de una ecuación (excepto para los axiomas entre constructores) o una regla de reescritura: se limitan a organizar información tomando otros términos simples y componiéndolos de forma ordenada en uno más complejo [3, 14].

Ecuaciones

Finalmente, un módulo puede contener una serie de ecuaciones, introducidas con la palabra reservada `eq`, y que definen cómo se reducen los diferentes términos en relación a los operadores que utilizan. Las ecuaciones pueden utilizar variables, y un sistema de ajuste de patrones (*pattern matching*) busca posibles aplicaciones de las ecuaciones para un término dado a reducir y vincula partes de este término con las correspondientes variables. Las variables deben ser tipadas, ya sea con el símbolo `:` en el propio lugar de utilización (como se muestra en el ejemplo) o en una definición separada introducida con la palabra reservada `var`. También existen las ecuaciones condicionales, introducidas por `ceq`.

Podemos pedir a Maude que reduzca el siguiente término utilizando el comando `reduce` durante una sesión interactiva:

```
Maude> reduce s(0) + s(0) .
reduce in NAT : s(0) + s(0) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s(s(0))
```

Se puede observar que el resultado es, tal y como se esperaba, `s(s(0))`. La ecuación utilizada para llegar a tal resultado ha sido la definida en la línea 9 del fragmento 2.1, con la sustitución $\{N \mapsto \texttt{0}, M \mapsto \texttt{s(0)}\}$, e inmediatamente después la de la línea 8, que elimina el cero sobrante, con la sustitución $\{N \mapsto \texttt{s(0)}\}$.

2.2.2. Axiomas ecuacionales

Maude permite al programador indicar ciertas propiedades útiles para los operadores mediante axiomas ecuacionales. Para ilustrar esta característica, vamos a intentar definir la multiplicación para los números naturales en notación de Peano. Una posible implementación podría ser esta:

```
1 fmod NAT is
2     sort Nat .
3
4     op 0 : -> Nat [ctor] .
5     op s : Nat -> Nat [ctor] .
6     op _+_ : Nat Nat -> Nat .
```

```
    op _*_ : Nat Nat -> Nat .

    eq 0 + N:Nat = N:Nat .
    eq s(N:Nat) + M:Nat = s (N:Nat + M:Nat) .

    eq 0 * N:Nat = 0 .
    eq N:Nat * s(M:Nat) = N:Nat + (N:Nat * M:Nat) .
endfm
```

Sin embargo, si ahora decidimos reducir el siguiente término, observaremos que Maude nos devuelve el mismo término intacto y no obtenemos ninguna reducción:

```
Maude> reduce s(0) * 0 .
reduce in NAT : s(0) * 0 .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s(0) * 0
```

Esto ocurre porque el ajuste de patrones no tiene éxito para ninguna de las ecuaciones definidas. El operador de multiplicación, al igual que el resto de operadores, respeta el orden de sus argumentos (en este caso dos, al ser binario); es decir, no es conmutativo. Esto se explica porque en Maude, como es común en muchos otros lenguajes de programación, el orden de los parámetros formales de una función se utiliza para desambiguar el proceso de enlazado (*binding*) de dichos parámetros con los argumentos de cada llamada. Por consiguiente, en la segunda ecuación se espera un término encabezado por un operador `s` en el lado derecho de `*`, pero se encuentra un `0`.

Maude ofrece, entre sus muchas otras características, un complejo sistema de ajuste de patrones *módulo* axiomas que potencia la expresividad de las definiciones. El lenguaje soporta una variedad de *axiomas ecuacionales* que podemos utilizar como etiquetas sobre un operador con objeto de cambiar el comportamiento del algoritmo de ajuste de patrones al intentar aplicar ecuaciones y reglas que le conciernen.

En este caso concreto, podemos indicar que las operaciones de suma y producto sobre los números naturales son ambas conmutativas utilizando el axioma `comm` sobre los respectivos operadores, de la siguiente forma:

```
fmod NAT is
    sort Nat .

    op 0 : -> Nat [ctor] .
    op s : Nat -> Nat [ctor] .
    op _+_ : Nat Nat -> Nat [comm] .
    op _*_ : Nat Nat -> Nat [comm] .

    eq 0 + N:Nat = N:Nat .
    eq s(N:Nat) + M:Nat = s (N:Nat + M:Nat) .

    eq 0 * N:Nat = 0 .
    eq N:Nat * s(M:Nat) = N:Nat + (N:Nat * M:Nat) .
endfm
```

Fragmento 2.2: Producto de números naturales

Nótese que añadir una segunda ecuación que intercambia los lados izquierdo y derecho no sería equivalente, ya que esto podría causar un bucle que impide la terminación

del programa, al poderse aplicar la ecuación infinitas veces sobre su propio resultado. El orden en el que Maude aplica las ecuaciones disponibles es dependiente de la implementación. Es responsabilidad del usuario asegurar que una teoría Maude es terminante.

```
eq N:Nat * M:Nat = M:Nat * N:Nat .  --- NO TERMINANTE!
```

Los axiomas ecuacionales, por otra parte, solucionan este problema trabajando a nivel de ajuste de patrones en lugar de afectar a la propia ejecución del programa, normalizando el término de acuerdo con un orden prefijado y aplicando más tarde la ecuación correspondiente de forma transparente para el usuario.

También podemos utilizar otros axiomas como la asociatividad (`assoc`) y el elemento neutro (`id`) para definir de forma expresiva una lista de enteros, en un único operador:

```
1 sort NatList .
2 subsort Nat < NatList . --- definimos la jerarquía de "subtipos"
3
4 op nil : -> NatList .   --- constante que simboliza una lista vacía
5 op _._ : NatList NatList -> NatList [ctor assoc id: nil] .  --- concatenación
6 --- ejemplo de uso: obtener el tamaño de una lista
7 op size : NatList -> Nat .
8 eq size(E:Nat . L:NatList) = s(size(L:NatList)) .
9 eq size(nil) = 0 .
```

Fragmento 2.3: Lista de números enteros

Dadas estas definiciones, un ejemplo de lista concreta sería: `s(0) . s(s(s(0))) . 0`.

Gracias al uso de axiomas ecuacionales, en Maude es extremadamente sencillo trabajar con estructuras de datos tales como listas, conjuntos o multiconjuntos, entre otras.

2.2.3. Teorías de reescritura

Una teoría de reescritura $\mathscr{R}$ está formada por uno o más módulos de sistema y opcionalmente módulos funcionales. Cada uno de ellos puede definir los mismos elementos que un módulo funcional y además un nuevo elemento, en apariencia similar a las ecuaciones, conocido como *regla de reescritura*. Las ecuaciones y operadores constituyen la teoría ecuacional $\mathscr{E}$ subyacente de $\mathscr{R}$.

Por ejemplo, considérese la especificación (parcial) de los números enteros dada por las siguientes ecuaciones, donde `s` y `p` son los operadores para el sucesor y el predecesor de un número natural, respectivamente:

```
1 eq X:Int + 0 = X:Int .
2 eq X:Int + s(Y:Int) = s(X:Int + Y:Int) .
3 eq p(s(X:Int)) = X:Int .
4 eq s(p(X:Int)) = X:Int .
```

En cierto sistema, se desea representar un estado binario con el constructor `[_,_]` : `Int` $\times$ `Int` $\rightarrow$ `State` que modela una serie de procesos en una red que consumen una serie de objetos desde un origen (el primer componente del estado) y los mueven hacia un destino (el segundo componente), manteniendo el número global de objetos invariante. Un estado `[s(0),s(0) + p(0)]` puede ser reescrito a `[0,s(0)]` utilizando la siguiente regla de reescritura que especifica la dinámica del sistema:

```
1 rl [A:Int, B:Int] => [p(A:Int), s(B:Int)] .
```

Durante la ejecución de una teoría de reescritura en Maude usando el comando `rewrite`, se toma un término de entrada y se simplifican en él todas las expresiones reducibles con ecuaciones módulo axiomas hasta llegar a un término normalizado e irreducible. A continuación, Maude selecciona una regla de reescritura de acuerdo con su estrategia predefinida (denominada "*outermost fair*") y de nuevo normaliza con ecuaciones, y así sucesivamente hasta que no queden reglas que aplicar sobre el resultado irreducible o eventualmente se alcance una condición de finalización (longitud de la secuencia). Esto causa la exploración de una de las ramas de ejecución posibles desde el término inicial.

En una búsqueda, sin embargo, se aplican todas las reglas posibles sobre los términos irreducibles respecto a las ecuaciones módulo axiomas. Cada aplicación de regla despliega un nuevo camino de ejecución, describiendo una estructura de árbol y permitiendo, por lo tanto, la exploración independiente de cada una de las posibilidades descritas por las reglas aplicables. Después, este proceso se repite de forma recursiva en cada una de las ramas desplegadas, reduciendo con ecuaciones módulo axiomas y desplegando en cada rama las correspondientes subramas en función de las reglas disponibles. El resultado final es un árbol que describe todas las combinaciones de aplicación de todas las posibilidades descritas por las reglas de reescritura, modelando todos los estados alcanzables por el sistema. Las hojas de tal árbol son términos irreducibles para los cuales ya no pueden aplicarse ecuaciones ni reglas.

Las reglas de reescritura se introducen con la palabra reservada `rl` (o `crl` si es condicional). Cuando se utilizan reglas de reescritura, una ejecución puede contener más de una solución. Una serie de comandos y parámetros permiten controlar qué estados de dicho árbol serán tomados como solución: por ejemplo, pueden solicitarse todas sus hojas.

Consideremos el siguiente ejemplo, que modela una tienda en un sistema monetario en el que las unidades son el dólar (`$`) y los cuartos de dólar (`c`), y existe una relación de conversión $1\$ = 4\mathtt{c}$. Un cliente de esta tienda puede tener tanto dólares como cuartos en cualquier combinación, y puede decidir comprar los objetos `x` e `y`, de precio $1\$$ y $3\mathtt{c}$ respectivamente, de forma impredecible y a placer, como se indica en el Fragmento 2.4.

Mediante la utilización de comandos podemos guiar a Maude indicando qué reglas pueden ser aplicadas, cuántas veces o qué estados del árbol son relevantes para los cómputos. Opcionalmente, el llamado *lenguaje de estrategias* [15, 16] permite expresar con detalle exhaustivo cómo se utilizan y aplican las reglas, separando dichas decisiones de la teoría a ejecutar.

2.2.4. Relación con los métodos formales

Las características de Maude permiten ver a los programas como especificaciones formales ejecutables, en forma de prototipos ejecutables precisos, rigurosos y útiles para la aplicación de técnicas formales variadas y versátiles que permiten probar la corrección del sistema, simular su comportamiento o verificar ciertas propiedades de este.

La capacidad de Maude para explorar ramas de ejecución a través de su sistema de reglas con el comando `search` permite comprobar ciertas propiedades relacionadas con la alcanzabilidad, entrando en el territorio de la comprobación de modelos. Es posible verificar propiedades en sistemas de infinitos estados a través de diversas abstracciones mediante la combinación de Maude con ciertas herramientas, incluyendo, pero no limitado a:

```
mod STORE is
 sorts Coin Item Marking .
 subsorts Coin Item < Marking .

 op __ : Marking Marking -> Marking [assoc comm id: null] . --- el operador es el propio carácter del espacio
 op null : -> Marking .
 op $ : -> Coin .
 op c : -> Coin .
 op x : -> Item .
 op y : -> Item .

 --- comprar
 rl [buy-x] : M:Marking $     => M:Marking x .
 rl [buy-y] : M:Marking c c c => M:Marking y .

 --- conversiones
 rl $ => c c c c .
 rl c c c c => $ .
endm
```

Fragmento 2.4: Programa Maude que modela una tienda con dólares y cuartos

- Church-Rosser Checker (CRC), que comprueba si un módulo es confluente para sus términos básicos (*ground terms*) y decreciente respecto a sus *sorts*;
- Maude Termination Tool (MTT), que comprueba que un módulo es terminante; o
- Sufficient Completeness Checker (SCC), que comprueba si las operaciones de un módulo son suficientemente completas, es decir, si están definidas para todo valor posible de sus argumentos.

Ciertas características de ejecución simbólica en Maude, a través del mecanismo de estrechamiento o *narrowing*, permiten la demostración automática de ciertas propiedades, la evaluación parcial, la verificación de protocolos criptográficos, la verificación simbólica de modelos o la unificación ecuacional de forma nativa. Los comandos de búsqueda con *narrowing* se encuentran disponibles en *Core Maude*, ofreciendo un entorno de verificación versátil y potente para una clase amplia de teorías [3].

2.3 Especialización de programas

La especialización o evaluación parcial de programas es una técnica de optimización que permite aumentar las prestaciones de un programa a partir del conocimiento de ciertos datos estáticos que utiliza como entrada [7].

Un programa P puede verse como una caja negra que toma ciertos datos de entrada i y, tras realizar una serie de cálculos o trabajo, devuelve unos datos de salida o. Los datos de entrada pueden dividirse en dos partes según el momento en el que se obtienen:

- Los datos de entrada *estáticos* (i_{static}) son aquellos que se conocen en el momento de compilar o procesar el programa, antes de su distribución o despliegue.
- Los datos de entrada *dinámicos* ($i_{dynamic}$) son aquellos que se conocen inmediatamente antes de o durante cada ejecución concreta.

Denotamos que o es la salida de la ejecución de un programa P que toma como datos de entrada $i_{static}, i_{dynamic}$ de la siguiente forma:

$$o = [\![P]\!][i_{static}, i_{dynamic}] \tag{2.1}$$

Durante un proceso de especialización parcial, utilizamos estos datos de entrada *estáticos* i_{static} para ejecutar aquellos fragmentos del programa que dependen únicamente de ellos, dejando un programa "residual" P^* cuyas operaciones dependen ahora únicamente de los datos dinámicos en lugar de depender de ambos, tal que ejecutar P^* sobre los datos dinámicos es equivalente a ejecutar P sobre todos los datos de entrada: [7]

$$[\![P^*]\!][i_{dynamic}] = [\![P]\!][i_{static}, i_{dynamic}] \tag{2.2}$$

También es posible realizar este proceso sobre una parte de los datos de entrada estáticos en lugar de su totalidad.

El programa parcialmente evaluado P^* resultante puede comportarse de igual forma que el original, o bien puede haberse restringido a unos casos concretos, fijando ciertos datos de entrada según unas asunciones concretas que son más restrictivas de lo que soportaba el programa original. Por lo tanto, la evaluación parcial de programas es una especialización segura solamente para las llamadas cubiertas por la especialización, ya que el programa resultado puede fallar para otras entradas no consideradas que sí eran soportadas en el programa original [17].

Por ejemplo, considérese el programa Maude en el Fragmento 2.5, que describe la potencia a^x para $a > 0$ y $x \geq 0$.[3]

```
1 fmod POWER is
2     protecting NAT .     --- importamos un módulo
3
4     op _**_ : Nat Nat -> Nat .
5
6     var A : Nat . var X : NzNat .       --- NzNat es un natural que no es cero
7     eq A ** X = A * (A ** sd(X,1)) .    --- sd(A:Nat,B:Nat) es la diferencia simétrica
8     eq A ** 0 = 1 .
9 endfm
```

Fragmento 2.5: Potencia de números naturales

Ahora imagínese un escenario en el que se necesita, para resolver cierto problema, obtener los cuadrados de ciertos números enteros. El código en cuestión permite calcular la potencia para cualquier exponente pero esto es ahora demasiado general y, en el contexto actual, es posible fijar el exponente `X` para que siempre tenga el valor 2. Lo que anteriormente era un dato dinámico ahora es un dato estático, permitiendo aplicar una especialización en este programa, empezando por la ecuación que utiliza la variable fijada `X`. Lo siguiente es una representación (que no se corresponde con código Maude válido) que ilustra la idea visualmente:

```
eq A ** 2 = A * (A ** 1) =
          = A * A * (A ** 0) =
          = A * A * 1 .
```

[3] La exponenciación para números naturales ya está implementada en el preludio de Maude bajo el operador `_^_`. Se reimplementa la operación de forma simplista con un operador *ad-hoc* solamente con propósito de construir este ejemplo.

En el resultado final no se invoca a la ecuación general de la línea 7 del Fragmento 2.5 con la variable `X`, ni a la ecuación específica de la línea 8 que eleva a cero. Podemos intuitivamente eliminar ambas ecuaciones del programa. Finalmente, como indica el axioma ecuacional `id` sobre el operador `_*_` del módulo `NAT`, el 1 es el elemento neutro del producto y, por consiguiente, el último producto de la ecuación desplegada no altera el resultado. El resultado final es el siguiente:

```
1 fmod SQUARE-POWER is
2     protecting NAT .
3
4     op _**_ : Nat Nat -> Nat .
5
6     var A : Nat .
7     eq A ** 2 = A * A .
8 endfm
```

Fragmento 2.6: Potencia al cuadrado de números naturales

Nótese que las capacidades del programa, antes capaz de realizar cualquier potencia arbitraria, se han visto reducidas. Sin embargo, el nuevo programa evita aplicar ciertas ecuaciones, reduciendo la potencia al cuadrado a una única operación. Entre otras ventajas, en programas suficientemente complejos, la especialización puede introducir mejoras notables de varios órdenes de magnitud sobre el comportamiento temporal de la ejecución [7].

Algunas área relevantes de aplicación de la evaluación parcial consisten en el desarrollo conducido por modelos, la ingeniería de lenguajes de dominio específico, la programación genérica, y la generación de casos de prueba, por mencionar algunas de las más recientes.

2.4 Ejecución simbólica en Maude

El estrechamiento o *narrowing* es una técnica de ejecución simbólica para lenguajes funcionales inicialmente introducida para la prueba de teoremas y hoy en día encontrada en algunos lenguajes de programación de alto nivel generalmente lógico-funcionales, como Curry, TOY o el propio Maude. En la actualidad, el *narrowing* se aplica en una gran variedad de áreas como en las pruebas de programas (*testing*) y la especialización de programas antes nombrada [14].

Esta técnica permite reducir términos cuando se desconoce parte de la información presente en ellos, por ejemplo por la presencia de *variables*. De hecho, se puede considerar la reescritura como un caso específico de *narrowing* en el que se dispone de toda la información necesaria, es decir, en el que no hay variables lógicas. En resumen, esta técnica permite conocer, dado un término con variables, cuáles son las instancias más generales de ese término que pueden ser reescritas con las reglas existentes [3, 14].

En capítulos posteriores se ilustra la utilidad de esta técnica al ser esta una parte fundamental del algoritmo de especialización de la herramienta Presto.

CAPÍTULO 3
El especializador automático Presto

Presto (estilizado como PRESTO) es un especializador o evaluador parcial simbólico y automático para Maude que busca mejorar el análisis y la verificación de sistemas mediante la optimización automática tanto de teorías ecuacionales como de teorías de reescritura [5].

En este capítulo se realiza una presentación y estudio de esta herramienta, que tras aplicar técnicas de ingeniería inversa es mejorada con el esquema de especialización incremental cuyo diseño y desarrollo es objetivo de este trabajo.

3.1 Presentación de la herramienta

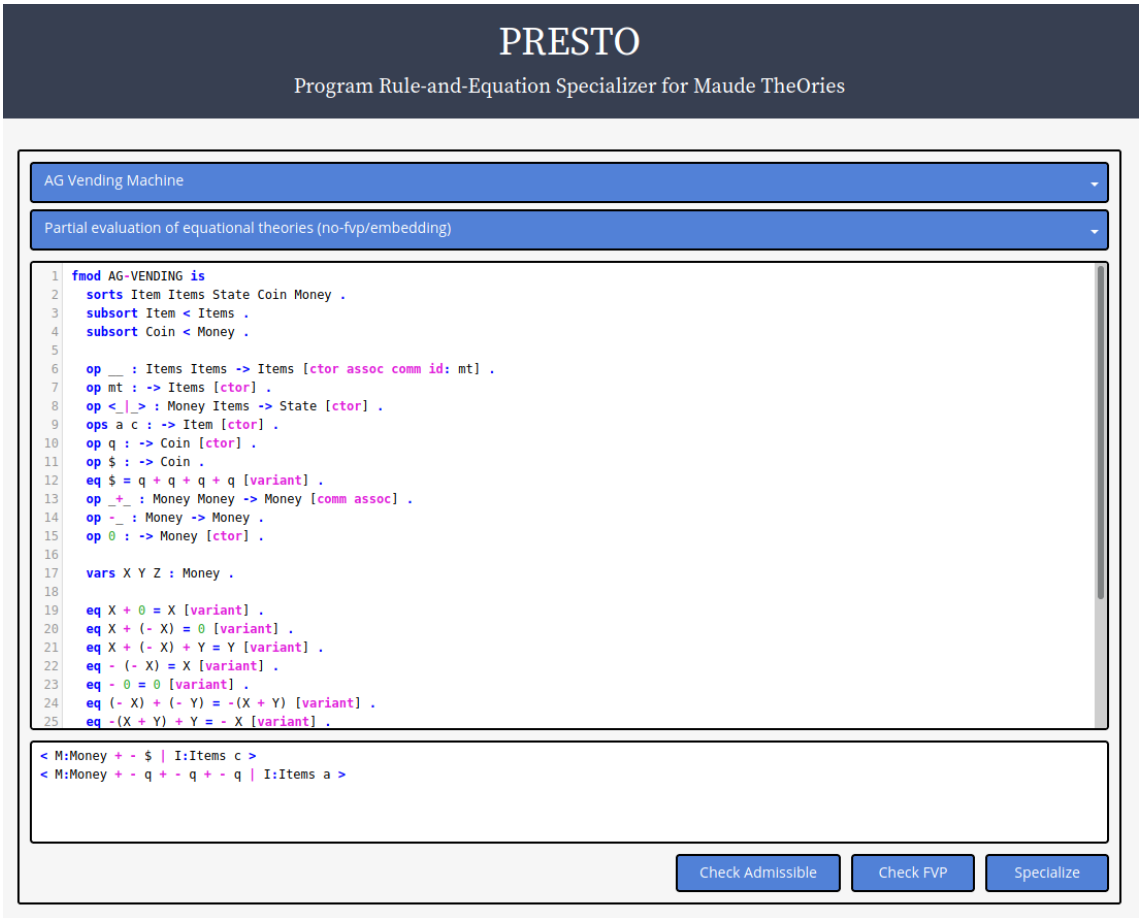
Presto es una implementación de la técnica descrita anteriormente de *especialización de programas* para el lenguaje Maude: se introduce un programa P escrito en Maude y (opcionalmente) algunos términos, para generar de forma automática un programa residual P^* que es más concreto y eficiente, si bien potencialmente más restringido [7]. Presto tiene como característica diferenciadora el poder especializar tanto teorías ecuacionales como de reescritura, dando finalmente soporte a la lógica de reescritura que subyace en Maude [6].

La herramienta Presto ha sido desarrollada por el grupo *Extensions of Logic Programming* (ELP) dentro del instituto *Valencian Research Institute for Artificial Intelligence* (VRAIN) de la *Universitat Politècnica de València* y se encuentra disponible en línea en `http://safe-tools.dsic.upv.es/presto`. Puede obtenerse información sobre su funcionamiento en su guía de inicio rápido [6], de la cual se resume información relevante en este apartado. Esta herramienta implementa el algoritmo NPER [18] descrito en secciones posteriores, que es una extensión reciente de la evaluación parcial dirigida por *narrowing* para programas lógico-funcionales (NPE) [4] al caso de las teorías de reescritura.

Presto ha demostrado ser útil en una nueva área de aplicación de la especialización de programas: en el análisis de protocolos, donde las teorías ecuacionales subyacentes pueden ser lo suficientemente complicadas como para incurrir en comportamientos temporales indeseables, al incluir intrincados operadores que pueden resultar en una gran cantidad de combinaciones de axiomas de asociatividad, conmutatividad o elemento neutro. En general, se busca aumentar las prestaciones en aquellos casos en los que se tiene una teoría ecuacional, ya sea por ella misma o como subyacente a una teoría de reescritura, que resulta demasiado general para una serie de casos específicos [18].

Presto permite a cualquier usuario la especialización, obteniendo mejoras de rendimiento de forma sencilla y automática, y proporcionando a mano toda la artillería necesaria para poder llevar a cabo este proceso.

Figura 3.1: La interfaz de usuario de Presto



3.1.1. Funcionamiento general

Presto tiene dos modos principales de funcionamiento. Primero, puede realizar la especialización de teorías ecuacionales, tal y como se ha presentado intuitivamente en el capítulo anterior. Como ya se ha visto a alto nivel, durante la especialización de un programa Maude dado por una teoría ecuacional se introduce código fuente formado por módulos funcionales en la primera caja de texto de entrada (Figura 3.1), y se introducen una serie de términos de especialización en la segunda caja de texto, más pequeña, que aparece en la parte inferior. Estos términos de especialización son los datos estáticos tal y como se veían en la formulación anterior: son nuestro compromiso de las posibles llamadas o ejecuciones genéricas que podrá aceptar el programa una vez finalizada la especialización. Intuitivamente, el programa resultante solamente podrá reducir términos que encajen con, al menos, uno los términos de especialización introducidos. Puede observarse la interfaz de Presto en la Figura 3.1.

Segundo, Presto también puede especializar programas complejos que contienen teorías de reescritura. Cuando se selecciona este modo, Presto especializa la teoría ecuacional subyacente respecto a las llamadas que se pueden observar en las reglas. Este modo está, por lo tanto, dirigido a aquellos casos en los que se tiene una teoría ecuacional subyacente que es demasiado compleja y general en relación a lo que en realidad necesitan

las reglas de reescritura del programa. Por consiguiente, en este modo no se introducen términos de especialización ni ningún dato de entrada estático: estos se infieren automáticamente a partir del propio código Maude correspondiente a las reglas.

Presto también dispone de una funcionalidad de compresión, que renombra los operadores utilizados en la teoría especializada con nuevos símbolos de función, mejorando de forma adicional el rendimiento en casos con llamadas anidadas. También hay disponible una función de inspección que permite observar los valores intermedios de las iteraciones del algoritmo interno.

3.1.2. Consideraciones de uso: propiedad FVP y atributo `variant`

Además de lo visto en el apartado anterior, cada uno de estos dos modos se despliega en más modos de funcionamiento respecto al tratamiento que se realiza de una propiedad crítica denominada propiedad de variantes finitos [19] (*finite variant property*, FVP) de los operadores del programa. Esta propiedad se relaciona con el hecho de si reescribir términos con ciertas sustituciones genera un número finito de trazas de ejecución a considerar [4].

La importancia de la FVP reside en que, si ésta se cumple, los árboles que genera internamente el algoritmo serán siempre finitos y se puede asegurar la terminación del algoritmo. Sin embargo, si esta propiedad no se satisface, la teoría en cuestión puede generar árboles infinitos y es necesario realizar operaciones adicionales de generalización para asegurar la terminación del proceso de especialización [4, 6]. Es decir, que si se elige un proceso genérico capaz de aceptar aquellas teorías que no cumplen la FVP es necesario realizar comprobaciones y generalizaciones que pueden incurrir en una penalización de rendimiento del proceso de especialización, y si se decide aceptar únicamente teorías que cumplen la FVP se reducen las capacidades de Presto al cubrir menos teorías.

El algoritmo utilizado por Presto, introducido en más detalle más adelante, permite la selección de un operador de desplegado (*unfolding*), solucionando el problema de si es necesario esperar la propiedad FVP o no, ya que puede ser elegido por el usuario.

Presto incluye tres operadores de desplegado [6]:

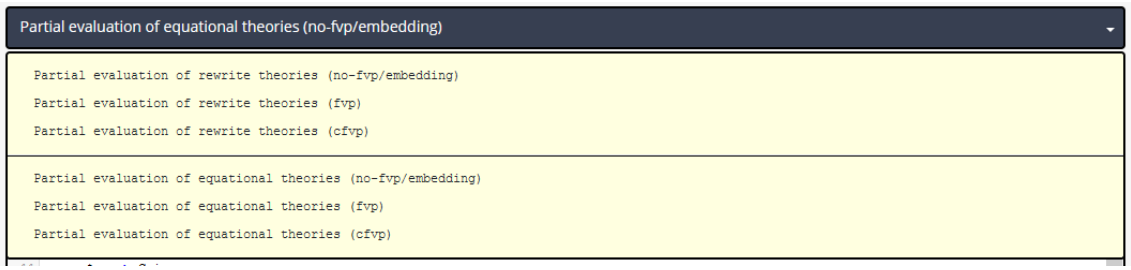
- El operador $\mathcal{U}_{\overline{fvp}}$ es el más genérico, cubriendo todas las teorías de reescritura ejecutables por Maude. Para ello, utiliza pasos adicionales relacionados con la comprobación de subsunción (*embedding*) ecuacional homeomórfica con el objetivo de asegurar la terminación del proceso. Puede verse este paso adicional como una comprobación de incrustación de estados abstractos, en el que se podan aquellas ramas del árbol interno que se consideran "equivalentes" a otras ya exploradas anteriormente.

- El operador $\mathcal{U}_{fvp}$ solamente funciona en aquellas teorías que satisfacen la FVP pero no realiza los pasos adicionales nombrados anteriormente. Esto permite una mayor celeridad en el proceso de especialización.

- El operador $\mathcal{U}_{cfvp}$ es para aquellas teorías que cumplen la propiedad de variantes constructoras finitos (*constructor finite variant property*, CFVP), que es una versión más estricta de la FVP. Este operador permite reducir el algoritmo interno de especialización a su versión más sencilla y veloz.

Las combinaciones de los dos modos de funcionamiento explicados anteriormente, junto con los 3 operadores explorados en este apartado, producen los 6 modos de funcionamiento que pueden verse en el desplegable de la interfaz de Presto:

Figura 3.2: Modos de funcionamiento de Presto



Con el objetivo de asistir al usuario a elegir el mejor operador para su caso de uso específico, Presto incluye un analizador de FVP disponible al presionar el botón "*Check FVP*". Como esta propiedad es indecidible, el resultado será o bien "sí", si la propiedad se satisface, o bien "incierto" si se termina el tiempo límite o el usuario aborta la operación. Únicamente si el resultado es positivo se recomienda al usuario elegir el operador de FVP.

Como ya se ha visto, la forma y propiedades de las ecuaciones en una teoría pueden ocasionar importantes decisiones sobre el proceso de especialización. Como no es posible asegurar que las teorías no tendrán ecuaciones que entren en conflicto con alguna de las invariantes esperadas para el proceso de especialización con Presto, el lenguaje Maude requiere el marcado de todas las ecuaciones utilizadas para la generación de variantes por algoritmos con el atributo ecuacional `variant`, mientras que las no marcadas se utilizan para simplificación [3]. Por consiguiente, solamente las ecuaciones marcadas con dicho atributo serán tomadas en cuenta por esta herramienta.

3.1.3. Ejemplo de uso

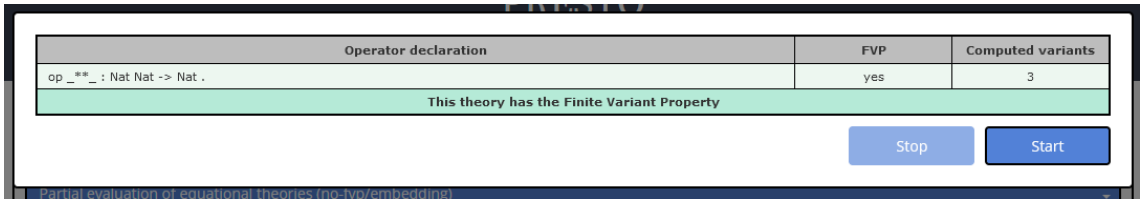
A continuación se expone un ejemplo sencillo que muestra el flujo de trabajo con la herramienta Presto. Pueden encontrarse más ejemplos en la propia herramienta, a través del desplegable superior.

Recuérdese del capítulo anterior la teoría ecuacional que calcula la potencia de un número natural, descrita en el Fragmento 2.5, pero añadiendo el atributo ecuacional `variant` tal y como requiere Presto:

```
1 fmod POWER is
2     protecting NAT .
3
4     op _**_ : Nat Nat -> Nat .
5
6     var A : Nat . var X : NzNat .
7     eq A ** X = A * (A ** sd(X,1)) [variant] .
8     eq A ** 0 = 1 [variant] .
9 endfm
```

Primero, es necesario comprobar la FVP para los operadores de esta teoría. Si se presiona el botón "*Check FVP*", tras unos instantes aparece el resultado que confirma que este programa sí cumple con dicha propiedad:

Figura 3.3: Resultado positivo de la comprobación de FVP en Presto



Teniendo en cuenta que la entrada es una teoría ecuacional (como se puede ver al ser un único módulo funcional) y el resultado obtenido en esta comprobación, es recomendable seleccionar el modo de funcionamiento "*Partial evaluation of equational theories (fvp)*" y mantener todos los atributos `variant`.

Al igual que se ha descrito en el ejemplo del capítulo anterior, el objetivo de especialización son las potencias cuya base es 2. Los términos de especialización, requeridos debido al modo de especialización para teorías ecuacionales, son los siguientes:

```
2 ** N:Nat
```

Tras pulsar en "*Specialize*", aparece el siguiente resultado:

```
1  fmod POWER is
2    inc BOOL .
3    pr NAT .
4    op _**_ : Nat Nat -> Nat .
5    --- eq A:Nat ** 0 = 1 [ variant ] .
6    --- eq A:Nat ** X:NzNat = A:Nat * (A:Nat ** sd(X:NzNat, 1)) [ variant ] .
7  endfm
8
9  fmod SPECIALIZED-FVP-POWER is
10   pr POWER .
11   eq A:Nat ** 2 = A:Nat * A:Nat * 1 [ variant ] .
12 endfm
```

El resultado obtenido es extremadamente similar al que se había derivado de forma intuitiva en la sección anterior, con énfasis en la línea 11, y de comportamiento completamente equivalente al del Fragmento 2.6.

La principal razón por la que Presto resulta tan efectivo en el área de análisis de protocolos es que no sólo logra una gran optimización para clases muy relevantes de teorías de reescritura sino que también puede reducir un espacio de *narrowing* de infinitas variantes a un espacio de búsqueda finito para la teoría ecuacional subyacente. Esto permite realizar análisis simbólico para $\mathscr{R}$ módulo $\mathscr{E}'$ de forma efectiva cuando era inabordable previamente [6, 18].

3.2 El algoritmo NPER

Como ya se ha mencionado, Presto utiliza un algoritmo llamado *narrowing-driven partial evaluation algorithm for rewrite theories* (NPER) para realizar la especialización o evaluación parcial de una teoría de reescritura dada $\mathscr{R}$ respecto a su teoría ecuacional subyacente $\mathscr{E}$, constituyendo este la parte nuclear de la herramienta [6]. Este procedimiento se basa en una extensión del algoritmo previo EQNPE [18] que permite la especialización dirigida por *narrowing* (NPE) de teorías ecuacionales [4]. Siguiendo las dependencias en

orden, primero se introduce el algoritmo EQNPE, que permite la especialización de teorías únicamente ecuacionales, y más tarde se extiende para introducir finalmente NPER, dando soporte a las teorías de reescritura características del lenguaje Maude.

Recordamos que para especializar un programa es necesario disponer de una serie de datos estáticos introducidos por el usuario. En el contexto de la especialización de teorías ecuacionales para el lenguaje Maude que ofrece EQNPE, el programa de entrada es la teoría ecuacional $\mathscr{E}$, y los datos estáticos proporcionados por el usuario son los términos Q. Por el contrario, el esquema NPER no requiere la introducción de términos Q, ya que dichos términos se infieren según las llamadas que se requieren de la teoría ecuacional subyacente $\mathscr{E}$, como ya se ha mencionado anteriormente.

El algoritmo NPER divide su flujo de control en local y global. El control local está gestionado por un operador de desplegado (*unfolding*) $\mathscr{U}$ y es el responsable de evitar las evaluaciones infinitas y de la construcción de las reglas residuales para cada llamada especializada. Por otro lado, el control global evita que se realicen un número infinito de iteraciones y decide qué funciones especializadas aparecerán en la teoría final. La existencia de esta separación permite la parametrización del algoritmo respecto a $\mathscr{U}$ [18], es decir, que pueda elegirse dinámicamente el operador de desplegado, solucionando problemas relacionados con el necesario equilibrio entre prestaciones y situaciones específicas que puedan presentarse en ciertas entradas según sus propiedades [6]. Por ejemplo, en la Sección 3.1.1 puede observarse cómo la selección de operadores permite una ganancia de prestaciones condicional para aquellos programas que cumplen la propiedad crítica conocida como FVP.

Algoritmo 1: Especialización de teorías ecuacionales (simplificado)

Entrada: Una teoría de reescritura $\mathscr{R}$, un conjunto de términos Q a ser especializados en $\mathscr{R}$ y un operador de desplegado $\mathscr{U}$.
Salida: Un conjunto de términos Q' que cubren completamente a (son capaces de evaluar lo mismo que) Q respecto a $\mathscr{R}$.

```
1  función EqNPE(R, Q, U):
2      Q ⟵ normalizar(Q)
3      T ⟵ ∅
4      repetir
5          Q' ⟵ Q
6          T ⟵ T ∪ desplegar(Q', R, U)
7          L ⟵ hojasNormalizadas(T)
8          Q ⟵ abstraer(Q', L, R)
9      hasta que Q' = Q
10     devolver (Q', T)
```

El algoritmo EQNPE toma por lo tanto como entradas una teoría que puede ser de reescritura, $\mathscr{R}$, de la que solamente se utilizarán las partes correspondientes a la teoría ecuacional, los términos de especialización Q y el operador de desplegado $\mathscr{U}$. Primero, el algoritmo normaliza los términos Q para prepararlos para su posterior procesamiento programático, y después inicia un bucle iterativo en el que busca encontrar un conjunto de términos Q' que *cubran* al Q original [4]. Informalmente, este concepto de *cobertura*, derivado del concepto formal de *closedness* [4], se relaciona con la capacidad de los términos derivados Q' de ejecutar con la misma capacidad operativa lo que describían mediante variables los términos de especialización del usuario Q. Si Q' cubre completamente a Q, se ha logrado una especialización completa tal y como se ha descrito anteriormente.

El segundo paso consiste en aplicar una operación de despliegue que utiliza la técnica de *narrowing* para evaluar parcialmente los términos Q, que (potencialmente) contienen

variables y por lo tanto (potencialmente) no pueden ser sometidos a un proceso convencional de reescritura o reducción. Esta técnica implica construir un árbol de *narrowing* finito $\mathcal{T}$ pero potencialmente parcial, y finalmente de dicho árbol se extrae el conjunto de hojas normalizadas $\mathcal{L}$ [4]. Estas hojas son términos, y algunos de ellos podrían estar aún *sin cubrir*, por lo que el árbol podría crecer a cada iteración. Para mantener finito dicho conjunto, se ejecuta una operación de abstracción que combina los términos aún no cubiertos con otros términos ya especializados, creando un nuevo conjunto mezclado de términos cubiertos y sin cubrir que podría requerir más especialización. El algoritmo entonces itera, empezando de nuevo el despliegue y así sucesivamente. Este paso de abstracción es necesario para asegurar que el conjunto de términos Q no crezca de forma infinita [4].

Finalmente, cuando se alcanza el punto fijo en el que se ha producido un conjunto de términos que ya no necesita más especialización (es decir, cuando no se han introducido nuevos términos sin cubrir) el algoritmo se detiene satisfactoriamente y retorna dicho conjunto Q'. Con él puede construirse inequívocamente el nuevo programa especializado [4].

Algoritmo 2: Especialización de teorías de reescritura (simplificado)

Entrada: Una teoría de reescritura $\mathscr{R} = (E, R)$ donde E son las ecuaciones y R las reglas, y un operador de desplegado $\mathscr{U}$.
Salida: Una nueva teoría de reescritura $\mathscr{R}''$ especializada respecto a su teoría ecuacional subyacente.

1 **función** `NPER`$(\mathscr{R}, \mathscr{U})$**:**
2 $R' \longleftarrow$ `normalizar`(R) `Fase 1: evaluación parcial`
3 $Q \longleftarrow$ `llamadasMaximales`(R')
4 $Q' \longleftarrow$ `EqNPE`$(\mathscr{R}, Q, \mathscr{U})$
5 $\mathscr{E}' \longleftarrow$ `generarTeoría`$(\mathscr{R}, Q')$
6 $\mathscr{R}' \longleftarrow$ `comprimir`$((E, R'), \mathscr{E}', Q')$ `Fase 2: compresión`
7 **devolver** $\mathscr{R}''$

Por otro lado, el algoritmo NPER toma como entradas una teoría de reescritura $\mathscr{R}$ y el operador de desplegado $\mathscr{U}$. De forma similar, empieza normalizando las reglas de reescritura para su posterior procesamiento y, a partir de ellas extrae todas las llamadas maximales a los operadores definidos por ecuaciones, determinando cuál es la teoría ecuacional subyacente que realmente es "necesaria", al ser requerida de forma directa por las reglas.

El algoritmo continúa llamando a EQNPE para la misma teoría y para los términos extraídos correspondientes a las llamadas maximales. También se transfiere conocimiento de cuál es el operador de desplegado a utilizar, ya que se delega en esta subrutina el flujo del despliegue del árbol de *narrowing*.

Finalmente, con los términos devueltos por EQNPE se construye la teoría ecuacional especializada $\mathscr{E}'$ y se realiza un proceso de compresión para crear una teoría de reescritura especializada compacta $\mathscr{R}''$ sin repetición de variables o términos innecesarios [18].

3.3 Ingeniería inversa del trabajo existente

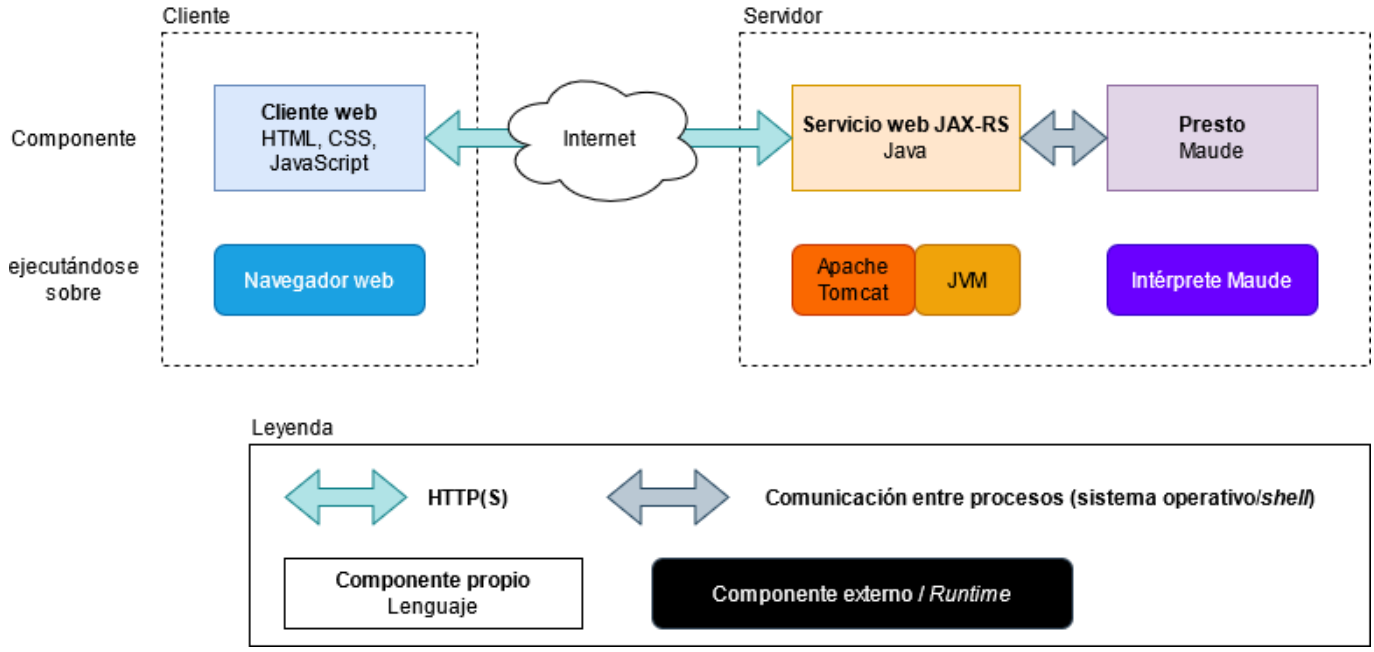
El especializador Presto contiene como parte nuclear la implementación del algoritmo NPER, y podemos ver la herramienta, a grandes rasgos, como una envoltura o carcasa encargada de recoger datos para lanzar y dirigir la ejecución de dicho algoritmo, recoger los resultados y mostrarlos al usuario. Si bien los fundamentos formales de Presto se desarrollan en varias publicaciones de revista [4, 18] no existe documentación descriptiva del código de la herramienta ni tampoco una descripción detallada de su arquitectura o diseño de la misma.

3.3.1. Resumen de la arquitectura

Presto es una aplicación web, ejecutada por un servidor web controlado por los desarrolladores y expuesta para que el público interesado pueda utilizarla con su navegador. Lo único que los usuarios necesitan para acceder a Presto es conocer su URL, `http://safe-tools.dsic.upv.es/presto`, y apuntar su navegador de preferencia hacia ella.

Siguiendo una anatomía típica de una aplicación web, el servidor HTTP encargado de la herramienta Presto actúa de *back end* distribuyendo el *front end*, un cliente web desarrollado en HTML, CSS y JavaScript para que se ejecute en el navegador cliente del usuario. A partir de ese momento, la aplicación cliente descargada al vuelo coopera con el servidor enviando solicitudes a los puntos de entrada de comunicación (*endpoints*) de la API REST expuesta por el servidor, esperando las respuestas que este le otorga y más tarde mutando el DOM de forma conveniente para mostrar al usuario la información que resulta de los procesos ejecutados.

Figura 3.4: Esquema de alto nivel de la arquitectura de componentes de Presto



En la máquina servidor, el servicio web lanza un intérprete de Maude encargado de ejecutar la implementación del algoritmo NPER, descrita completamente en Maude, y mediante primitivas de comunicación entre procesos provistas por el propio sistema operativo se entregan los resultados de vuelta al servicio web. Esta parte de la imple-

mentación es llamada en posteriores secciones la *lógica de dominio*, al realizar el trabajo útil propiamente dicho de especializar programas de forma automática.

3.3.2. Servicio web

El *servicio web* de Presto se refiere en estas líneas al código fuente que describe la API y la lógica invocada, que determina cómo se responde en cada URI, cuándo y por qué. Esto es diferente del *servidor web*, que se encuentra en una capa lógica inferior, recibiendo y manejando las solicitudes HTTP desde la red y transfiriendo el control de forma oportuna al *servicio web* con los datos ya decodificados para su consumo.

La implementación específica del *servicio web* utiliza Jakarta RESTful Web Services (JAX-RS), una tecnología desarrollada originalmente por Sun Microsystems y ahora mantenida por Eclipse Foundation. Es una especificación de API para el lenguaje Java que permite definir rápida y cómodamente servicios web según el patrón arquitectónico Representational State Transfer (REST) [20]. La implementación específica de *servidor web* HTTP compatible con JAX-RS utilizada por Presto es Apache Tomcat, aunque no existe ningún acoplamiento específico con esta plataforma en el código fuente y podría ser sustituida.

JAX-RS permite definir una API REST en forma de funciones con anotaciones. Un *servidor web* HTTP compatible con esta tecnología recibe las solicitudes HTTP, las procesa y obtiene los argumentos, y luego busca funciones anotadas tal que su nombre y el tipo de sus parámetros formales sean compatibles con la solicitud entrante. En caso afirmativo, lanza a ejecución dicha función con los datos, delegando el flujo de control a la lógica escrita por el usuario en Java, con ayuda de un entorno de tiempo de ejecución de Java (JRE) y su correspondiente máquina virtual Java (JVM). Los argumentos recibidos son enlazados con los parámetros formales de las funciones de forma transparente para el desarrollador del servicio [21].

Al solicitar la especialización de un programa, el cliente lo envía a la ruta[1] `POST /parse-program` para comprobar si existen errores sintácticos detectados por Maude y poder así fallar de forma temprana para entradas que no corresponden a programas Maude bien formados. Si este paso termina de forma satisfactoria, el cliente realiza una segunda solicitud a `POST eval-program`, que inicia el proceso de especialización como tal. Los parámetros formales de la ruta en cuestión se muestran en la Tabla 3.1.

Tabla 3.1: Parámetros formales de la ruta `POST eval-program` de la API del servicio web Presto

Nombre	¿Obligatorio?	Tipo	Descripción
`modname`	Sí	Texto	Nombre del módulo padre a especializar.
`program`	Sí	Texto	El programa Maude entero a especializar.
`terms`	Situacional	Texto	Los términos iniciales de especialización. Únicamente se requieren cuando se ejecuta una especialización de teoría ecuacional (modos 1, 2 y 3).
`mode`	Sí	Entero	Un indicador del modo seleccionado, indicando si se realiza una especialización de teoría ecuacional o de reescritura, o si se asume FVP. (Tabla 3.2)
`fullmaude`	Sí	Booleano	Verdadero si el programa requiere Full Maude para ejecutarse, o falso en caso contrario.

[1] Las rutas se indican como el verbo HTTP seguido de la URL relativa sobre la raíz implícita del servicio, expuesta en algún dominio web.

El servidor utiliza durante el procesamiento de las solicitudes las utilidades de Java, independientes del sistema operativo subyacente, para lanzar un *shell* que lance un intérprete de Maude con los parámetros correspondientes. Más tarde, ya en código Java, se recoge de los flujos estándar (`stdout` y `stderr`, o sus análogos en el sistema operativo correspondiente) la salida, y finalmente se realiza un breve análisis sintáctico (*parsing*) desarrollado específicamente para esta tarea, con el objetivo de interpretar los resultados y devolver las partes más relevantes con la representación adecuada.

Una vez finalizada la ejecución de la función Java, se realiza según sea conveniente una serie de procesos de serialización o *marshalling* para "empaquetar" los datos de retorno, construyendo una respuesta HTTP válida para el solicitante original y enviándola por la red.

Los datos devueltos finalmente por el servicio web al cliente en caso de una ejecución exitosa consisten en JSON codificado en una codificación propia que sustituye ciertos caracteres especiales y de control por unas secuencias únicas de caracteres ASCII, que se aplica en el momento en el que se ejecuta el código en Maude de la lógica de dominio. La decodificación de dicha respuesta, proceso realizado por el cliente en JavaScript, resulta en JSON válido con el esquema ilustrado por el siguiente ejemplo resumido (Figura 3.1). Las señalizaciones de recuadros con puntos suspensivos al final de los literales de cadena denotan que ésta es más larga, pero parte ha sido omitida visualmente.

```
{
    "ori": "fmod MK-EVEN is\n  inc BOOL .\n  sort Nat .\n  op 0 : -> Nat [ ctor ] ."...,
    "eval": "fmod MK-EVEN is\n  inc BOOL .\n  sort Nat .\n  op 0 : -> Nat [ ctor ] "...,
    "evalren": "fmod MK-EVEN is\n  inc BOOL .\n  sort Nat .\n  op 0 : -> Nat [ ctor"...,
    "total": "",
    "ren": "Org: mkEven(X:Nat, Y:Nat)\nRen: f0(X:Nat, Y:Nat)",
    "debug": "Iter: 0 \nQSet: mkEven(X:Nat, Y:Nat)\n\nIter: 1 \nQSet: mkEven(X:Nat, "...
}
```

Fragmento 3.1: Resumen de respuesta decodificada de la ruta `POST eval-program` de la API del servicio web Presto

3.3.3. Cliente web

El cliente web ha sido desarrollado utilizando HTML, CSS y JavaScript utilizando *Bootstrap* [22], que incluye una serie de definiciones y utilidades en estos lenguajes que permiten la construcción acelerada de sitios web. El sitio web es estático al servirse siempre de la misma forma desde el servidor tal y como este lo tiene almacenado, modificándose el sitio únicamente a él mismo como respuesta a eventos de interacción del usuario en el navegador o en respuesta a solicitudes AJAX iniciadas por el propio cliente.

El cliente web utiliza también la librería de JavaScript *jQuery* [23]. De ella se utilizan más concretamente funciones de manipulación del DOM para crear animaciones y transiciones de navegación. Además, se utiliza su función `ajax` [24] para realizar las solicitudes HTTP asíncronas a la API del servidor. Los datos enviados, incluyendo los programas en Maude, se codifican como componente de URI (`encodeURIComponent`) [25] antes de enviarse. La información recibida es utilizada por el cliente web para dibujar la interfaz de usuario de la pantalla de resultado con la información correspondiente.

Para el dibujo de cajas de texto editables con coloración de sintaxis se ha utilizado *CodeMirror* [26], una solución de lado de cliente que añade editores de texto extensibles para sitios web en JavaScript.

3.3.4. Lógica de dominio

Como ya se ha expuesto anteriormente, en cierto punto del funcionamiento de Presto se lanza un intérprete Maude para realizar la especialización automática de programas con los datos correspondientes. A este intérprete se le instruye para que cargue el archivo `presto.maude`, que implementa el algoritmo NPER y todas las transformaciones y utilidades auxiliares sobre programas Maude, utilizando las potentes propiedades de metalenguaje del propio Maude.

El funcionamiento interno codifica los modos de funcionamiento de forma numérica, como se observa en la Tabla 3.2. Recuérdese que un modo de funcionamiento es seleccionado por el usuario utilizando uno de los desplegables de la interfaz, tal y como se desarrolla en la Sección 3.1.1.

Tabla 3.2: Codificación interna de los modos de funcionamiento de Presto

Modo	Tipo teoría	FVP	Nombre en la interfaz
0	Ecuacional	No	Partial evaluation of equational theories (no-fvp/embedding)
1	Ecuacional	Sí	Partial evaluation of equational theories (fvp)
2	Ecuacional	Sí, constructores	Partial evaluation of equational theories (cfvp)
3	Reescritura	No	Partial evaluation of rewrite theories (no-fvp/embedding)
4	Reescritura	Sí	Partial evaluation of rewrite theories (fvp)
5	Reescritura	Sí, constructores	Partial evaluation of rewrite theories (cfvp)

Para lanzar a ejecución la lógica de dominio escrita en Maude, es necesario guiar al intérprete Maude con los comandos apropiados. Para ello, el servicio web en Java genera una serie de módulos y comandos de forma condicional al modo actual y a si el programa enviado utiliza las características de orientación a objetos ofrecidas por *Full Maude*, entre otras. Puede observarse un ejemplo en el Fragmento 3.2, donde el servicio web ha generado los comandos que instruyen a la lógica de dominio en Maude a especializar el programa `mkEven`, disponible entre los ejemplos de Presto, utilizando el modo cero.

En estos comandos se llama al operador punto de entrada `eval` utilizando un módulo auxiliar `ELP-TOOL`, que importa el módulo `PRESTO`, que incluye transitivamente toda la lógica de dominio y sus utilidades auxiliares. El propio programa a especializar enviado por el usuario, en este caso el módulo `MK-EVEN`, y los términos de especialización (si el modo implica la especialización de teorías ecuacionales), se incluyen en la información pasada el intérprete de Maude, como se puede observar en el ejemplo. El servicio web causa que el intérprete de Maude cargue el archivo `presto.maude` de la ruta correspondiente del sistema y acepte toda esta información por su entrada estándar, realizando la invocación de un comando semánticamente similar a `cat comandos | maude presto.maude`.

El lenguaje Maude permite la sobrecarga (*overloading*) de operadores según la aridad y el tipo de sus parámetros formales [3], como lo permiten otros lenguajes de programación, por ejemplo Java o C# con sus métodos [27, 28]. De esta forma, existen varios operadores llamados `eval` que se diferencian ya sea por aridad o por los tipos de sus parámetros, y muchos de estos operadores se llaman los unos a los otros. A continuación se puede ver un resumen de las funciones implicadas en el inicio de una traza de especialización de un programa, las cuales mayoritariamente deciden el flujo de las funciones subyacentes según el *modo* seleccionado.

```
mod ELP-TOOL is pr PRESTO .
  op elpTerms : -> String .
  eq elpTerms = "mkEven(X:Nat,Y:Nat)" .
endm

fmod MK-EVEN is
  sort Nat .

  ops 0 1 : -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat [ctor assoc comm id: 0] .

  vars X Y : Nat .

  op mkEven : Nat Nat -> Nat .
  eq mkEven(X + X + 1,1 + Y) = mkEven(X + X + 1 + 1,Y) [variant] .
  eq mkEven(X:Nat + X:Nat,0) = X + X [variant] .
endfm

set print format off .
set print conceal on .
print conceal fmod_is_sorts_.____endfm .
print conceal mod_is_sorts_._____endm .
rew in ELP-TOOL : eval(upModule('MK-EVEN,true),upModule('MK-EVEN,false),(elpTerms),0) .

quit .
```

Fragmento 3.2: Ejemplo de comandos enviados por el servicio web al intérprete Maude para la especialización de una teoría ecuacional en modo 0

```
--- caso ecuacional (modos 0, 1 y 2)
op eval : Module Module String Nat -> String . --- punto de entrada
op eval : Module Module TermSet Nat -> String .

--- caso reescritura (modos 3, 4 y 5)
op eval : Module Module Nat -> String . --- punto de entrada

--- funciones principales
op embPE  : Module TermSet -> RootLeafSet .
op fvpPE  : Module TermSet -> RootLeafSet .
op cfvpPE : Module TermSet -> RootLeafSet .
```

Las funciones marcadas como *punto de entrada* son aquellas llamadas por el servicio web en el texto con comandos pasado por la entrada estándar, iniciando la traza. Si los modos coinciden con especialización de una teoría ecuacional (modos 0, 1 y 2) se llama a una función `eval` que recibe un parámetro de tipo `String` adicional, para los términos de especialización del usuario. En caso contrario, como no se proporcionan términos de especialización, se llama una sobrecarga de `eval` que requiere tal `String`.

La traza tras los operadores de punto de entrada está representada de forma esquemática en la Figura 3.5. Los cuadrados coloreados representan operadores sobrecargados desambiguados por su nombre y sus parámetros formales. Las líneas entre parámetros formales indican los datos que los operadores "arrastran" entre llamadas; es decir, los datos que ha recibido y pasará a dicha llamada siguiente en la traza. Las flechas continuas siguen el flujo de datos de los términos de especialización, de especial importancia en el algoritmo. Las líneas y flechas que aparecen por el lateral de una de las funciones representan su valor de retorno.

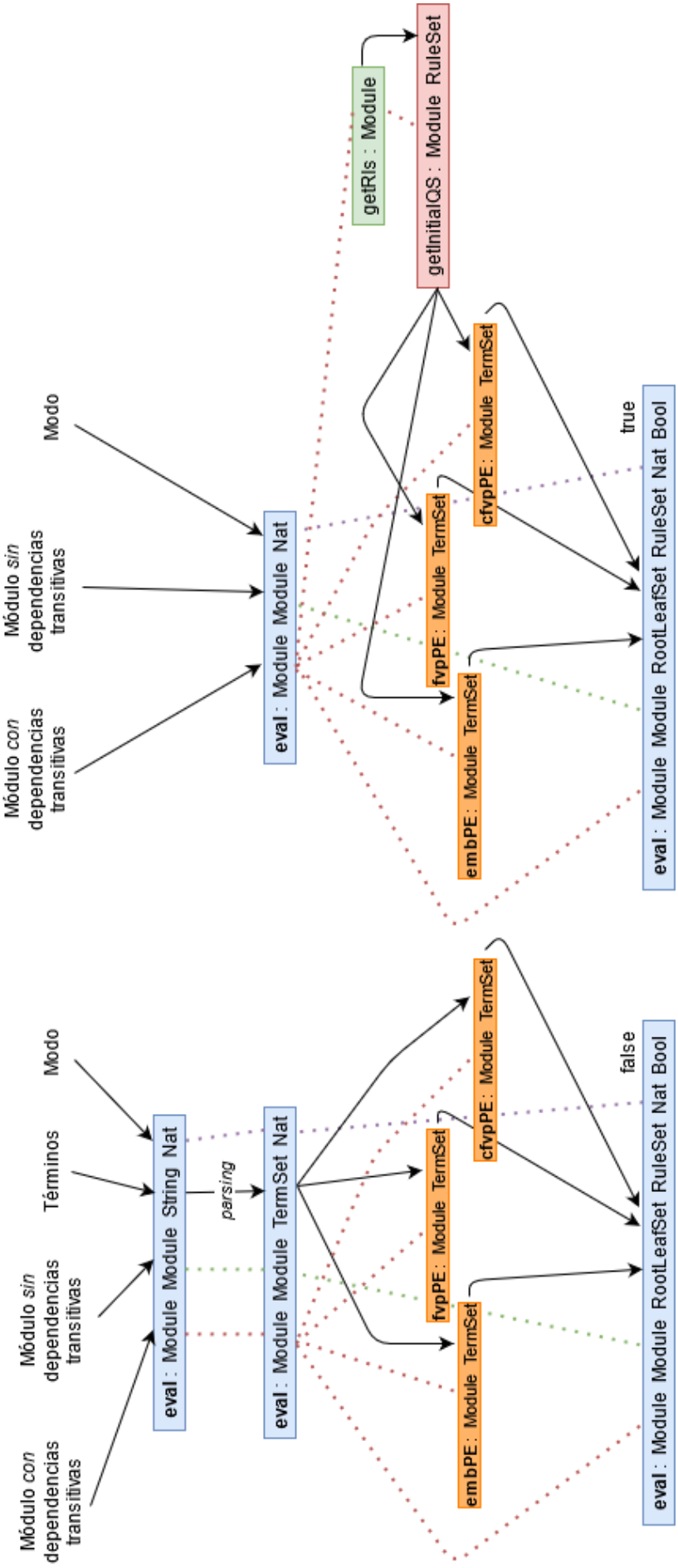


Figura 3.5: Visión esquemática del flujo de datos entre llamadas a operadores para la especialización en Presto

Respecto a los tipos, `TermSet` es consistente con el concepto de conjunto de términos tal y como se presenta en los algoritmos con la letra Q. El tipo `RootLeafSet` corresponde con el llamado árbol de *narrowing* $\mathcal{T}$, aunque es en realidad una representación compacta con las raíces y las hojas, información suficiente para la codificar la traza del algoritmo. Este tipo `RootLeafSet` guarda información importante para la inspección, anotando los nodos de raíces y hojas con las iteraciones correspondientes y sus sustituciones. Como indica el algoritmo y aunque no es estrictamente necesario para obtener el programa especializado como tal, se devuelve de forma adicional este árbol junto con el resultado Q', aportando información de depuración.

La función `eval` para teorías ecuacionales realiza un análisis sintáctico o *parsing* de los términos en forma de `String` utilizando las capacidades de metalenguaje de Maude y luego llama a la función `eval` que recibe los mismos datos pero intercambiando `String` por `TermSet`, que es el resultado de dicho *parsing*. Un error en este proceso resulta en la interrupción de la traza y la devolución de un código de error al servicio web, que será devuelto al cliente.

La sobrecarga de `eval` con `TermSet`, o su análoga para los casos en los que no se aceptan términos, realiza entonces una selección de la rama de ejecución atendiendo al modo: si el modo corresponde a uno en el que se espera la propiedad FVP (modos 1 y 4) entonces se llama a la función `fvpPE`, que inicia el algoritmo correspondiente con el operador de despliegue para FVP. De forma similar, para los modos sin FVP (0 y 3) se selecciona la función `embPE` y para la CFVP (2 y 5) se selecciona la función `cfvpPE`. El resultado de esta función es pasado entonces a una última sobrecarga de `eval` que genera finalmente el resultado, tomando el `RootLeafSet` devuelto por una de las funciones de desplegado presentadas anteriormente. Además, se pasa un parámetro de tipo `Bool` indicando si la especialización es para teorías de reescritura, es decir, `false` para los modos 0, 1 y 2; y `true` para el resto.

En los casos de teoría de reescritura no se dan términos de especialización, sino que estos se infieren. Para ello, se toma el módulo de entrada (concretamente el módulo cargado en el metanivel con sus dependencias) y se pasa como parámetro a la función `getInitialQS`, que obtiene las llamadas que se realizan en las reglas a las ecuaciones del mismo programa. Esta función implica una llamada a `getRls`, que obtiene las reglas de un módulo. Nótese que aunque los términos se obtienen de forma distinta, tanto en el caso ecuacional como en el caso de reescritura la traza termina en una de las funciones `embPE`, `fvpPE` o `cfvpPE`. Los términos en un caso son otorgados por el usuario y en otro inferidos automáticamente, pero el punto de entrada al algoritmo es el mismo y siempre requiere términos de especialización.

Enlazando este análisis con el algoritmo presentado anteriormente, las funciones `eval` implementan el flujo general del algoritmo NPER y las funciones terminadas en `PE` son llamadas al algoritmo EQNPE con diferentes operadores de despliegue $\mathcal{U}$. El operador `getInitialQS` se corresponde con la función `llamadasMaximales` tal y como se presenta en el Algoritmo 2. La generación de teoría y la compresión se realizan en la sobrecarga final de `eval`, que es la que recibe un `RootLeafSet`.

3.4 Caso de estudio: comprobador de modelos CTL

Se puede ilustrar la necesidad de especialización con un ejemplo de programa real: un intérprete de lógica de árboles de computación (CTL), un componente vital de los comprobadores de modelos que consiste en un verificador algorítmico de programas que comprueba si cierto modelo satisface o no cierta fórmula dada en dicha lógica. Las en-

tradas de dicho componente son por lo tanto dos, un modelo y una fórmula CTL, ambos válidos [10].

Este programa ha sido implementado por el autor de este trabajo en Maude siguiendo el algoritmo originalmente escrito para Prolog propuesto por M. Leuschel y S. Gruner [29]. Aunque el código fuente completo de este programa puede verse en la sección anexa A.1, para construir este ejemplo solamente es necesario observar a un alto nivel la forma de los términos de entrada que aceptan. En la literatura sobre comprobación de modelos y CTL, es habitual utilizar la siguiente notación para indicar el hecho de que un modelo $\mathcal{M}$ satisface la fórmula ϕ a partir de cierto estado inicial s:

$$(\mathcal{M}, s) \models \phi$$

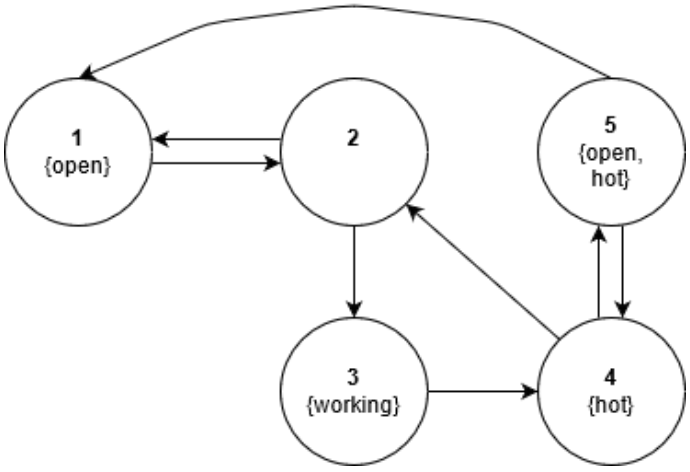
Utilizando la expresividad del lenguaje Maude, el programa define un operador infijo imitando el símbolo de consecuencia lógica ($\models$) para aceptar términos iniciales de forma y apariencia muy similares:

$$(\boxed{\textit{modelo}}, \boxed{\textit{estado inicial}}) \ \texttt{|=} \ \boxed{\textit{fórmula}}$$

Un término de esta forma es reducido al átomo `True` si el modelo en dicho estado inicial satisface la fórmula considerada, o a `False` en caso contrario.

Como ha sido introducido en capítulos anteriores, un $\boxed{\textit{modelo}}$ es una *estructura de Kripke*, es decir, un sistema de transiciones con etiquetas que recuerda a un autómata finito determinista. Por ejemplo, la siguiente figura ilustra una estructura de Kripke válida que modela un horno industrial que se puede abrir y cerrar y puede estar caliente, donde los números en negrita identifican cada estado y los conjuntos indican las etiquetas de cada uno[2]:

Figura 3.6: Una estructura de Kripke para un horno, sin estados de error



La estructura de Kripke nos indica con sus arcos que si el horno se encuentra abierto (estado uno), este puede pasar a un estado en el que ya no lo está (dos), y desde ahí puede volver a abrirse o puede activarse para empezar a calentar (tres). Sin embargo, una vez alcanzado ese estado de trabajo, ya no puede abrirse, porque no existe transición

[2]Se aplica la llamada *asunción del mundo cerrado* (CWA), esto es, que si un estado no tiene una etiqueta entonces representa implícitamente su negada. Por ejemplo, un estado que no tiene `hot` es necesariamente `¬hot`.

desde el estado tres al uno, ni al cinco, los únicos en los que se encuentra abierto. De forma similar, el resto de estados y transiciones describen el comportamiento contextual del horno industrial respecto a los impulsos que puede recibir del exterior y percibir desde sus propios sensores. En este caso de estudio se utilizará este modelo de horno industrial cuando se requiera de una entrada para el comprobador de modelos CTL, que es el principal programa de interés.

La representación de este modelo en el programa Maude que nos concierne consiste en una lista de transiciones de la forma *estado origen* -> *estado destino*, seguido de una lista de etiquetas expresadas como [*estado* : *proposición*] que representan las proposiciones que son ciertas en cada estado (por ejemplo, "abierto" (`open`) en el estado uno). La estructura de Kripke mostrada arriba se representa < *transiciones* ; *etiquetas* > como se muestra a continuación:

```
< 1 -> 2, 2 -> 1, 2 -> 3, 3 -> 4, 4 -> 2, 4 -> 5, 5 -> 4, 5 -> 1 ; [1 : open], [3 : working], [4 : hot], [5 : open], [5 : hot] >
```

Utilizando el comprobador de modelos CTL es posible demostrar automáticamente, por ejemplo, la siguiente propiedad de seguridad sobre este modelo: *"el horno no puede empezar a funcionar directamente mientras está abierto"*. Un usuario interesado en la verificación formal puede obtener de este enunciado una fórmula CTL y entonces traducirla a la representación esperada por este programa Maude, esto es, el campo *fórmula*:

"El horno no puede empezar a funcionar directamente mientras está abierto."

$$\downarrow$$

$$\mathrm{AG}\ \neg(open \wedge (open \implies \mathrm{EX}\ working))$$

$$\downarrow$$

```
(AG (Not (open And (open Implies (EX working)))))
```

Los operadores AG y EX son parte de la lógica CTL y se han implementado en el programa Maude en su forma literal. La implicación lógica se corresponde con el operador `Implies` y, finalmente, `open` y `working` son constantes relacionadas con las proposiciones que aparecen en el modelo del horno industrial.

A continuación, se extiende el programa del comprobador de modelos CTL con el siguiente módulo que contiene las definiciones sobre el horno industrial a modelar, que son requisito indispensable para poder introducir la estructura de Kripke y la fórmula. De otra forma, Maude no tendría constancia de las correspondientes proposiciones atómicas como, por ejemplo, `open`.

oven.maude

```
load ctl.maude  --- cargar el model checker CTL

fmod OVEN is
    protecting CTL-SATISFACTION .

    op open : -> Proposition .              --- definir las proposiciones...
    op working : -> Proposition .
    op hot : -> Proposition .

    eq open == working  = False [variant] . --- ...y sus relaciones (todas diferentes)
    eq open == hot      = False [variant] .
    eq working == hot   = False [variant] .
endfm
```

Ya es posible ejecutar el comprobador de modelos CTL. Existe a disposición un modelo $\mathcal{M}$ y una fórmula ϕ, por lo que solamente es necesario reducir $(\mathcal{M}, s) \models \phi$ tras cargar el módulo `OVEN` con Maude:

```
Maude> load oven.maude
Maude> reduce (< 1 -> 2, 2 -> 1, 2 -> 3, 3 -> 4, 4 -> 2, 4 -> 5, 5 -> 4, 5 -> 1 ; [1 :
open], [3 : working], [4 : hot], [5 : open], [5 : hot] >, 1) |= (AG (Not (open And (open
Implies (EX working))))) .
    --- salida parcialmente omitida
rewrites: 2359 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool': True
```

El resultado es el átomo `True`, que indica que la propiedad de seguridad es cierta: si el horno está abierto, nunca en el siguiente estado el horno se encontrará funcionando. De forma similar a lo aquí realizado, con las mismas definiciones puede comprobarse cualquier fórmula CTL sobre el modelo del horno mostrado. Cambiando las definiciones de `OVEN` por otras, puede realizarse la misma operación sobre otros modelos. La única parte "constante" que se importa sin modificar para todos los modelos es el módulo `CTL-SATISFACTION`.

Ahora bien, cierto usuario podría estar interesado solamente en una parte de todas las infinitas fórmulas CTL que pueden ser correctamente procesadas y resueltas por el comprobador de modelos con el modelo planteado. Imagínese, por ejemplo, una empresa que fabrica hornos industriales que está interesada en la verificación de varias fórmulas CTL para este modelo, pero únicamente las de seguridad. Con el objetivo de ganar rendimiento en los procesos de especialización, es posible especializar el comprobador de modelos para que solamente verifique fórmulas que sigan cierto patrón, en un programa más compacto, sencillo y rápido.

Para especializar este programa, que solamente utiliza ecuaciones, debe seleccionarse el modo "*Partial evaluation of equational theories (no-fvp/embedding)*". El programa de entrada se introduce en la caja superior y es el nombrado como `ctl`, disponible en la sección anexa A.1, concatenado con el módulo `OVEN` justo después (sin el comando `load`). Como término de especialización, por ejemplo, considérese el siguiente:

```
(<... ; ...>, S:State) |= (AG (Not (open And (open Implies (EX P:Proposition)))))
```

Las partes subrayadas son variables (con su tipo), indicando un "agujero" lógico de la fórmula que puede ser rellenado más tarde de forma dinámica. El programa especializado podrá procesar fórmulas que cumplan con este patrón, es decir, que sean iguales a la propuesta excepto por estas partes resaltadas. La parte antecedente a `|=` se encuentra parcialmente omitida y se corresponde con el mismo modelo del horno industrial, igual que en las invocaciones anteriores.

Al finalizar la especialización, si se accede a la pestaña *Specialized with Compression* se podrá observar el programa especializado y comprimido. Este programa contiene las definiciones necesarias para poder expresar el modelo y las fórmulas de entrada en una copia de los módulos originales sin ecuaciones y reglas. Asimismo contiene un nuevo módulo con las nuevas ecuaciones y reglas generadas (en este caso solamente ecuaciones, al tratarse de una teoría ecuacional) que permiten calcular los resultados que eran cubiertos por el programa original y que coinciden con el patrón dado en los términos de especialización. Son, en cierto modo, un "resumen" de todas las ecuaciones y reglas eliminadas dirigidas a cubrir solamente el comportamiento especificado. Dicho módulo es el siguiente:

```
1 fmod SPECIALIZED-noFVP*-OVEN is
2   pr OVEN .
```

```
3    op f0 : State Proposition -> [Formula] .
4    eq < 1 -> 2,2 -> 1,2 -> 3,3 -> 4,4 -> 2,4 -> 5,5 -> 1,5 -> 4 ; [1 : open],[3 :
working],[4 : hot],[5 : hot],[5 : open] >,S:State |= (AG Not (open And (open Implies (EX
P:Proposition)))) = f0(S:State, P:Proposition) [ variant ] .
5    eq f0(0, $1:Proposition) = True [ variant ] .
6    eq f0(s(0), hot) = False [ variant ] .
7    eq f0(s(0), open) = False [ variant ] .
8    eq f0(s(0), working) = True [ variant ] .
9    eq f0(s(s(0)), hot) = False [ variant ] .
10   eq f0(s(s(0)), open) = False [ variant ] .
11   eq f0(s(s(0)), working) = True [ variant ] .
12   eq f0(s(s(s(0))), hot) = False [ variant ] .
13   eq f0(s(s(s(0))), open) = False [ variant ] .
14   eq f0(s(s(s(0))), working) = True [ variant ] .
15   eq f0(s(s(s(s(0)))), hot) = False [ variant ] .
16   eq f0(s(s(s(s(0)))), open) = False [ variant ] .
17   eq f0(s(s(s(s(0)))), working) = True [ variant ] .
18   eq f0(s(s(s(s(s(0))))), hot) = False [ variant ] .
19   eq f0(s(s(s(s(s(0))))), open) = False [ variant ] .
20   eq f0(s(s(s(s(s(0))))), working) = True [ variant ] .
21   eq f0(s(s(s(s(s(s($1:Nat')))))), $2:Proposition) = True [ variant ] .
22 endfm
```

Este módulo define directamente mediante constantes el resultado para cada combinación de estado y proposición en las variables `S` y `P` originales, respectivamente. Si se observa por ejemplo la línea 12, se puede observar que la función auxiliar llamada con los valores sin modificar de `S` y `P` devuelve `False` para las entradas `s(s(s(0)))` (representación interna del número tres) y `hot`. Esto es, la reducción final es tal que para la fórmula de entrada "*El horno no puede estar caliente directamente tras estar abierto.*" sobre el estado tres el nuevo programa es capaz de responder falso[3] en una única llamada, sin ejecutar ni una sola línea del algoritmo original del comprobador de modelos, que ocupa más de 200 líneas y utiliza estructuras de repetición mediante recursión. Toda la lógica ha quedado reducida de forma similar en las otras líneas, consumiendo gran parte del coste computacional de forma estática y resultando en un programa notablemente más simple que el anterior.

El programa especializado completo, con unas pocas ecuaciones originales añadidas para la representación más cómoda de los números naturales (por ejemplo, para poder utilizar el literal `3` como entrada en lugar de `s(s(s(0)))`) se muestra en la sección anexa A.2.

[3] La respuesta correcta para esta fórmula con el modelo dado es efectivamente "falso", porque el horno puede estar caliente mientras está abierto (estado 5) y al cerrarse puede seguir caliente (estado 6).

Parte II

Descripción del trabajo realizado

CAPÍTULO 4
Mejorando Presto: especialización incremental

Se ha introducido anteriormente la herramienta Presto, un especializador parcial automático para teorías de reescritura en Maude capaz de especializar tanto teorías ecuacionales respecto a un conjunto de términos iniciales como teorías de reescritura respecto a su teoría ecuacional subyacente. Esta herramienta, ya operativa y desplegada en producción de forma previa al inicio de este trabajo, ha sido rediseñada con el fin de soportar un modo incremental que permite la especialización iterativa, bajo control del usuario, lo que amplía los escenarios posibles de utilización de la herramienta, su experiencia de uso y en general sus prestaciones en materia de aplicabilidad.

Con el objetivo de evaluar también las mejoras, que requieren la ejecución y manipulación directa del código fuente, ha sido necesario previamente realizar un proceso de ingeniería inversa que incluye la comprensión del código fuente y del algoritmo de especialización utilizado internamente, cuya implementación ha debido ser modificada para permitir la ejecución incremental controlable desde el exterior sin alterar su comportamiento y propiedades formales como la corrección y completitud. A continuación, y atendiendo a la arquitectura específica de esta aplicación, se han efectuado los cambios pertinentes en el código fuente desde el propio núcleo de Presto hacia la API del servidor web, extendiéndose finalmente a modificaciones de la interfaz de usuario final destinadas a exponer las nuevas posibilidades de control al usuario.

4.1 Análisis del problema

El estado del arte de herramientas de especialización automática de código como Presto está mayoritariamente enfocado a producir código altamente eficiente, donde prima la automatización completa, la eficiencia de desempeño, la corrección funcional y atributos de calidad centrados en la parte formal, técnica y funcional, frente a características relacionadas con la usabilidad como el soporte a la depuración o prueba de la especialización obtenida, o la capacidad de obtener un mayor control sobre la granularidad del proceso, entre otras.

Siguiendo en esta línea, la versión de Presto existente de forma previa a este trabajo, tal y como se expone en el capítulo anterior, es monolítica respecto al proceso: una vez ha empezado el proceso de especialización, de forma que este o bien finaliza completamente con resultados correctos o falla sin otorgar ningún resultado parcial de forma que sólo un usuario superexperto podrá ser capaz de revisar sus decisiones, y sin apoyo de la herramienta. Todas las diferentes partes que forman el algoritmo de especializa-

ción utilizado se presentan de forma compacta y homogénea como un único proceso de "especialización", un todo o nada que oculta los detalles subyacentes.

Esto tiene sentido desde un punto de vista donde priman la completitud y corrección funcionales: si no es posible obtener los mejores resultados posibles que puede producir la herramienta hasta la completitud debido a problemas como el tiempo excesivo que puede requerir su compleción, se muestra un mensaje de error y se descartan todos los datos intermedios sin posibilidad de extraer ninguna información del proceso abortado. En la especialización, es imperativo preservar la semántica original [7], y no existe riesgo de exponer al usuario un programa que no funcione o lo haga de forma distinta a la esperada pero tampoco se le da la posibilidad de inspeccionar o revisar las entradas y elecciones activamente introducidas.

Esta decisión tiene claros inconvenientes respecto a la usabilidad, porque fuerza una granularidad determinada sobre el usuario y dificulta el refinamiento progresivo de la especialización. Si el usuario introduce una serie de entradas en la herramienta que, ya sea por violación o inadecuación de los requisitos de especialización o de decisiones de especialización inadecuadas, resultan en un proceso de especialización fallido, dicho usuario no es capaz de obtener ningún resultado o información más allá de los que muestra el mensaje de error. Esto es especialmente frustrante si el proceso de especialización fracasa para dicho caso concreto y el usuario debe esperar un tiempo de procesamiento no despreciable antes de recibir el mensaje de error.

Es posible, como ya se ha dicho, que el error mostrado sea la indicación de que se ha alcanzado el límite de tiempo disponible (*timeout*), lo cual causa incluso mayor confusión: ¿están las entradas violando algún requisito de especialización como son la confluencia, terminación y propiedad FVP, causando un tiempo desmedido del proceso de especialización; o son las entradas correctas pero son demasiado complejas para encontrar una solución dentro del límite impuesto? ¿Puede repararse dicha situación? ¿Cómo? En un escenario como el planteado, es posible que el interesado necesite realizar varias pruebas que resulten en el consumo de todo el tiempo de procesamiento hasta el límite permitido tratando de encontrar una combinación que deje de causar un tiempo desproporcionado, de forma similar a una serie de pruebas de caja negra.

Es necesario otorgar al usuario herramientas que le cedan el control del proceso de especialización, permitiéndole seleccionar una mayor granularidad y permitiendo el refinamiento progresivo y la depuración de la especialización incluso (y especialmente) en los casos en los que el usuario ha incurrido en algún error al elegir decisiones de especialización o entradas distintas a las deseadas. El objetivo no es solamente amortiguar el tiempo invertido en procesos de especialización fallidos con información parcial útil, sino ayudar con esta información al interesado a efectuar cambios dirigidos en las entradas para acercarse a un proceso de especialización satisfactorio.

4.2 Diseño de la solución

4.2.1. Preámbulo y planteamiento

Un planteamiento que surge tras el análisis del problema es similar al que plantea un depurador o *debugger* cuando recorre la ejecución de un programa respecto a su código fuente: para analizar detenidamente un proceso que previamente se percibía como una caja negra monolítica, se detiene dicho proceso en ciertos puntos para exponer información que es relevante a dicho momento temporal, y se otorga al usuario control sobre el flujo de ejecución pausado para que pueda reanudarlo o realizar otras operaciones más complejas [30].

Para ver si dicho planteamiento es plausible, es necesario buscar si existen puntos de detención conceptualmente coherentes, seguros y compatibles con la implementación. Si se reconsideran las definiciones del Algoritmo 2 utilizado por Presto, puede observarse que la parte más interna del proceso es un bucle que obtiene términos a especializar de forma iterativa y los despliega nuevamente utilizando la técnica de *narrowing* (líneas de la 3 a la 7 del Algoritmo 1). De hecho, las iteraciones asociadas a términos que ya se muestran en la pestaña *Inspect* en la versión original de Presto se relacionan específicamente con las iteraciones del bucle mencionado; es decir, existe previamente para los usuarios finales de la herramienta una relación conceptual entre la sección mencionada y el "bucle principal" del proceso de especialización. Ya se dispone de un potencial punto de detención razonable: cada una de las iteraciones del subalgoritmo EQNPE.

El nuevo flujo de trabajo plantea la posibilidad de seleccionar un modo "incremental" con la interfaz de usuario que extraiga y muestre los resultados parciales tras cada iteración en la pantalla y otorgue al usuario la posibilidad de continuar, progresando a través de la ejecución del algoritmo "paso a paso".

Tras cada iteración se dispone de los conjuntos parciales de términos de especialización Q y Q', por lo que el usuario podría observar cuáles son los términos que han sido tratados en dicha iteración, y cuáles serán los utilizados por la siguiente iteración. Esto tiene varias implicaciones. El usuario podría ver si el proceso de especialización se acerca a una especialización deseable observando si los términos se vuelven demasiado generales. Por ejemplo, para un operador `f` en cierta iteración puede observarse que en su conjunto de términos se dispone de una llamada específica `f(A)`, pero en la siguiente iteración se observa una llamada `f(f(A))` que fuerza a `f` a volver a una versión genérica `f(X)`, donde la variable `X` es más general que cualquier término no variable `A`. El usuario podría entonces darse cuenta de que no le resulta de interés continuar dicho proceso de especialización sin alterar los datos de entrada o las decisiones iniciales.

En otro escenario distinto, un usuario cuyo proceso de especialización no finaliza debido a que se alcanza el límite máximo de tiempo de ejecución podría recibir información que antes no podía obtenerse al poder finalizar de forma satisfactoria una o más iteraciones pudiendo alcanzar un punto que causa el comportamiento temporal indeseado. El usuario podría en dicho caso, además, observar qué llamadas causan que se alcance dicho punto, permitiéndole en el mejor caso enmendar de forma efectiva las decisiones de especialización.

Respecto al control por parte del usuario, al detener el algoritmo entre iteraciones las únicas opciones previstas son o bien continuar el algoritmo o bien abortarlo en dicho momento. Aunque esta forma de trabajo incremental da pie a permitir la modificación de los datos "en caliente", en medio del algoritmo, esto no se puede permitir sin comprobar antes que se conservan las correspondientes condiciones teóricas del algoritmo original, lo que lamentablemente queda fuera del ámbito de este trabajo. Esta operación se manifiesta de forma intuitiva como una sencilla orden de "Continuar", y puede ser extendida dando la opción de continuar cierto número de iteraciones indicado por el usuario, aumentando de forma adicional la capacidad del usuario de seleccionar la granularidad deseada para cada sesión.

También ha tenido un gran peso en el diseño de la solución la valoración de los desarrolladores y usuarios actuales de Presto, que son expertos en el dominio de la especialización y en el lenguaje Maude. Una serie de reuniones con dichos expertos ha confirmado de forma explícita que la adición de un nuevo flujo de trabajo que permita al usuario detenerse entre iteraciones con el fin de analizar la información utilizada por el programa resulta trascendental para los usuarios actuales y futuros de la herramienta. Sus indicaciones han forjado el diseño de la solución destacando necesidades y detalles específicos

a distintos niveles, complementando de forma vital el análisis del dominio realizado por el autor de este trabajo.

4.2.2. Extendiendo NPER para su uso incremental

Para permitir la detención condicional del algoritmo NPER utilizado por Presto es necesario modificar la implementación existente, ya que esta es puramente secuencial y monolítica: una vez se lanza la llamada a ejecución, el intérprete de Maude se detendrá con el resultado al final de la ejecución. Otra alternativa podría ser la utilización de las herramientas de depuración de Maude para detener el algoritmo sin modificarlo [3], pero estas herramientas se encuentran pensadas para el desarrollo y no para el uso en producción: con el objetivo de otorgar las mayores prestaciones de rendimiento, flexibilidad, legibilidad, etc. resulta mejor opción soportar el nuevo flujo de trabajo revisando y extendiendo el código fuente Maude para introducir primitivas destinadas al uso habitual en producción, ya que esto es posible.

Un planteamiento intuitivo para poder "seccionar" el algoritmo existente es el seguido por las semicorrutinas [31], también conocidas como *generadores*, que se encuentran presentes en diversos lenguajes populares como C# o Python [32, 33]. Una semicorrutina es un caso especial de corrutina, un componente que generaliza el concepto de función que permite suspender la ejecución en ciertos puntos que permiten la posterior reentrada de la ejecución por uno de dichos puntos de entrada. Al contrario que una corrutina verdadera, al suspender la ejecución éstas delegan el flujo de control al llamante, perdiendo control de cuándo volverán a ejecutarse [31].

Tal y como se ve en los lenguajes nombrados, una corrutina se presenta como una función (o método) tradicional que, cuando empieza a ejecutarse, lo hace por el punto de entrada inicial típico que se esperaría de una función y, al alcanzar una de las llamadas *instrucciones yield* [32, 33], éstas devuelven un valor y retornan el control a la función llamante. Sin embargo, a diferencia de una instrucción *return*, la semicorrutina "recuerda" donde se detuvo la ejecución y, cuando ésta sea llamada de nuevo, seguirá por donde se quedó, utilizando la instrucción *yield* como punto de entrada en lugar del inicio estándar de la función [31–33].

Maude no dispone de semicorrutinas [3], pero siguiendo el marco lógico planteado y averiguando cuáles son los puntos de detención y cuál es la información que se devuelve en cada uno de ellos, puede plantearse un nuevo algoritmo con instrucciones *yield*. Más tarde, pueden observarse dichas instrucciones para determinar cuáles son todos los puntos de entrada y salida del flujo de control de la función, determinando a su vez cuáles son los posibles recorridos que pueden realizarse a través del código fuente entre los puntos de entrada y salida. Finalmente, puede utilizarse dicha información para crear una serie de funciones que seccionan la primera, tal que una serie de llamadas a dichos fragmentos resultan en un proceso semánticamente equivalente al primero.

En el caso específico del algoritmo NPER, existe un único punto de detención localizado dentro del subalgoritmo EQNPE. Véase la propuesta de algoritmo modificado, $\text{EQNPE}_{\text{ITER}}$ en el Algoritmo 3, donde el algoritmo principal NPE sigue siendo tal y como se describe en el Algoritmo 2 pero se ha sustituido la llamada a `EqNPE` por una a `EqNPEIter`. Las instrucciones *yield* se representan con la palabra reservada **dar**, como se observa en la línea 8. En ésta línea el algoritmo devuelve una indicación de que el proceso aún no ha finalizado (`false`), el conjunto de términos considerado en dicha iteración Q, el árbol de *narrowing* $\mathcal{T}$ con la información de depuración asociada y, finalmente, un conjunto de términos sin cubrir Q^*. Al final el algoritmo se devuelve una indicación de finalización (`true`), el resultado Q' y el árbol de *narrowing* ahora completado $\mathcal{T}$.

Algoritmo 3: Especialización de teorías ecuacionales incremental

```
1  función EqNPEIter(ℛ, Q, 𝒰):
2      Q ⟵ normalizar(Q)
3      Q* ⟵ ∅
4      𝒯 ⟵ ∅
5      repetir
6          dar (false, Q, 𝒯, Q*)
7          Q' ⟵ Q
8          𝒯 ⟵ 𝒯 ∪ desplegar(Q', ℛ, 𝒰)
9          ℒ ⟵ hojasNormalizadas(𝒯)
10         Q* ⟵ términos no cubiertos en ℒ
11         Q ⟵ abstraer(Q', 𝒯, ℛ)
12     hasta que Q' = Q
13     devolver (true, Q', 𝒯)
```

El conjunto de términos sin cubrir Q^* es una adición original al algoritmo incremental que aporta más información de la que se podía observar originalmente en la información de depuración de la pestaña *Inspect*, dando una visión de los datos vitales que maneja internamente el algoritmo, al ser los términos no cubiertos los que guían el flujo de control del algoritmo entre iteraciones.

Habiendo determinado los puntos de detención y entrada, es posible realizar un análisis de los recorridos del flujo de control dentro del Algoritmo 3. Son los siguientes:

1. Desde el inicio de la función en la línea 2 hasta la línea 6, ya que el bucle **repetir ... hasta que ...** evalúa su condición al final.

2. Desde la línea 6 y pasando por la condición $Q' = Q$ (punto fijo). Si es verdadera, se escapa del bucle hacia la línea 13. En caso contrario, se vuelve a las 6, donde el flujo de control es delegado de nuevo.

Estos dos recorridos determinan dos funciones separadas que pueden ser llamadas por una función conductora en un orden determinado para emular el comportamiento de la función original, como se muestra en el Algoritmo 4. Si se sigue el flujo impuesto por la función de ejemplo conductora, se puede observar que la traza final resulta semánticamente equivalente a la anterior, con la diferencia de que ahora se dispone de los datos intermedios en la función conductora, y de que el algoritmo original delega el flujo de control a una función externa capaz de hacer acciones como mostrar la información en una pantalla. Además de seccionar en funciones más pequeñas, también se han añadido datos a los retornos y a los parámetros formales de las funciones implicadas, al ser necesario ahora arrastrar de forma explícita datos que antes se encontraban en una variable dentro del mismo ámbito (*scope*), y que ahora quedan invisibles entre ámbitos distintos.

Ahora, puede finalmente extenderse este algoritmo de forma natural a todo el ámbito de NPER, incluyendo las instrucciones previas y posteriores a la llamada a $\text{EQNPE}_{\text{ITER}}$ al inicio de `EqNPEIterBegin` y tras el retorno final de `EqNPEIterNext`, resultando en un algoritmo "interactivo" seguro para teorías de reescritura, manteniendo y extendiendo las características ya existentes y preservando las propiedades formales de Presto.

Algoritmo 4: Especialización de teorías ecuacionales incremental, algoritmo seccionado

```
1  función EqNPEIterBegin(ℛ, Q, 𝒰):
2      Q ⟵ normalizar(Q)
3      Q* ⟵ ∅
4      𝒯 ⟵ ∅
5      devolver (false, Q, 𝒯, Q*)
6  función EqNPEIterNext(ℛ, Q, 𝒰, 𝒯, Q*):
7      Q' ⟵ Q
8      𝒯 ⟵ 𝒯 ∪ desplegar(Q', ℛ, 𝒰)
9      ℒ ⟵ hojasNormalizadas(𝒯)
10     Q* ⟵ términos no cubiertos en ℒ
11     Q ⟵ abstraer(Q', 𝒯, ℛ)
12     si Q' cubre a todo Q entonces
13         devolver (true, Q', 𝒯)
14     en otro caso
15         devolver (false, Q, 𝒯, Q*)
16     fin
                                          Ejemplo de función conductora
17 función EqNPEIterDriverExample(ℛ, Q, 𝒰):
18     (B, Q, 𝒯, Q*) ⟵ EqNPEIterBegin(ℛ, Q, 𝒰)
19     repetir
20         (B, Q, 𝒯, Q*) ⟵ EqNPEIterNext(ℛ, Q, 𝒰, 𝒯, Q*)
21         mostrarInformación(B, Q, 𝒯, Q*)
22     hasta que B = true
```

4.3 Implementación de la nueva herramienta incremental

Para implementar el flujo de trabajo descrito, es necesario modificar todos los componentes en la tubería lógica que forman cliente, servidor e intérprete de Maude como se observa en la Figura 3.4. Primero, el código de Maude que define la lógica de dominio de Presto debe modificarse de forma que pueda especializarse un programa de forma completa utilizando el algoritmo NPER incremental planteado, utilizando por ejemplo la línea de comandos directamente sobre el intérprete. A continuación, es necesario modificar el servidor para añadir rutas a la API o modificar las existentes para exponer la nueva funcionalidad. Finalmente, es necesario modificar la interfaz de usuario para que utilice toda esta nueva infraestructura y exponga la característica al usuario.

Recuérdense los modos de funcionamiento de la Tabla 3.2. Es necesario notar que no se da soporte a los modos con CFVP porque estos utilizan una versión depurada y modificada del algoritmo NPER que aprovecha la existencia de esta propiedad para evitar realizar un cálculo realmente iterativo durante la ejecución de EqNPE, separándose de lo planteado en el Algoritmo 1. Seccionar la implementación correspondiente para poder capturar datos intermedios requeriría introducir repetición de forma artificiosa, afectando negativamente a las prestaciones de la versión para CFVP, que es equivalente a la que tiene FVP pero busca ganar rendimiento. Por lo tanto, solamente se da soporte a los modos cero, uno, tres y cuatro.

Como se ha visto anteriormente, el algoritmo "interrumpible" NPER incremental necesita una función conductora. Esta función es la que toma la decisión de qué hacer con los datos intermedios capturados y de cuándo avanzar entre iteraciones. Como es el usua-

rio quién debe controlar el avance y la interfaz de usuario quien muestra los resultados, se ha decidido que sea el cliente quien implemente dicha lógica. Esto también concuerda con las buenas prácticas que dictan que el estado debe ir en el cliente en lugar de en el servidor web, y esto tiene especial relevancia teniendo en cuenta que no existía previamente ningún mecanismo de sesión que permita mantener este estado en el servidor.

4.3.1. Extensión de la lógica de dominio

La lógica de dominio se encuentra implementada íntegramente en un archivo llamado `presto.maude`. Con el objetivo de aislar los cambios y seguir un principio de extensión antes que modificación, se ha escrito un nuevo archivo distinto `presto-iter.maude` que define un nuevo módulo `PRESTO-ITER` que incluye al módulo `PRESTO`, y que define las nuevas funciones segmentadas utilizando las funciones ya definidas cuando sea posible, delegando la funcionalidad al código ya existente en medida de lo posible. La utilización no incremental de la herramienta utiliza las llamadas ya existentes importadas de `presto.maude`. Esta decisión busca aislar al máximo los posibles errores introducidos en este trabajo y afectar positivamente a la auditabilidad del código nuevo.

A continuación se muestra una vista general del archivo `presto-iter.maude`, con la signatura de los operadores:

```
fmod PRESTO-ITER is
    protecting PRESTO . --- Inclusión de las definiciones existentes en presto.maude

    --- (1) Definición de resultados parciales
    sort PartialResult .
    op <_;_;_> : Bool TermSet RootLeafSet          -> PartialResult  [ctor] .

    --- (2) Concatenación de resultados parciales
    op __ : PartialResult PartialResult -> PartialResult .
    op __ : RootLeafSet PartialResult -> PartialResult .

    --- (3a) API principal: puntos de entrada y preprocesamiento
    op evalIterBegin : Module Module String Nat Nat -> String . --- modos 0 y 1
    op evalIterBegin : Module Module Nat Nat -> String . --- modos 3 y 4

    --- (3b) API principal: continuación tras la primera iteración
    op evalIterNext : Module Module PartialResult Nat Nat Nat -> String .

    --- (4) Serialización de los datos de retorno
    op evalIter : Module Module PartialResult RuleSet Nat Bool -> String.

    --- (5) Extensión incremental del algoritmo NPER
    op embPEIter : Module TermSet Nat Nat -> PartialResult .
    op fvpPEIter : Module TermSet Nat Nat -> PartialResult .

    --- (6) Extensión incremental del algoritmo EqNPE
    op eqNPEIter : Module TermSet Bool Nat Nat -> PartialResult .
```

Primero, se observa la definición de un nuevo *sort* `PartialResult` (1) que se corresponde con la tupla $(B, Q, \mathcal{T}, Q^*)$ devuelta por las funciones. El primer dato se corresponde con un valor booleano que indica con `true` o `false` en caso de que el cómputo haya finalizado; en caso contrario; el segundo dato contiene el conjunto de términos Q que va actualizando el algoritmo entre iteraciones; y el tercer dato consiste en el árbol de *narrowing*, codificado con el *sort* `RootLeafSet`. Lo que en el algoritmo se muestra como el cuatro y último campo, correspondiente a otro conjunto de términos donde almacenar los términos no cubiertos Q^*, no aparece en la definición al extraerse más tarde de $\mathcal{T}$ en

`evalIter` y devolverse en el JSON codificado como parte de la información de depuración en el campo `debug`.

Obsérvese la API principal (3), que define un operador `evalIterBegin` que inicia la operación de especialización incremental y un operador `evalIterNext` que continúa la operación tras la primera iteración, que se llama de forma reiterada, siguiendo el patrón que plantearía un iterador. Los argumentos que reciben estos operadores, de forma similar a las llamadas `eval` originales, son los siguientes:

- `evalIterBegin`, sobrecarga primera, para los modos soportados para teorías ecuacionales (cero y uno).
    1. El módulo en el metanivel (`Module`) a ser especializado *con* dependencias transitivas respecto a sus inclusiones, es decir, elevado al metanivel como `upModule('NOMBRE-MODULO,true)`.
    2. El módulo en el metanivel (`Module`) a ser especializado *sin* dependencias transitivas respecto a sus inclusiones, es decir, elevado al metanivel como `upModule('NOMBRE-MODULO,false)`.
    3. Los términos de especialización iniciales como `String`, que serán analizados y convertidos en un `TermSet`. Existe otra sobrecarga idéntica pero que toma el `TermSet` directamente ya bien formado en este tercer parámetro y que es llamada implícitamente por esta sobrecarga que toma `String`.
    4. El modo de funcionamiento codificado como `Nat`, de acuerdo con la Tabla 3.2. Se soportan los valores cero y uno para esta sobrecarga.
    5. La iteración en la que detenerse (*exclusive*) como `Nat`, en caso de se quiera avanzar varias de golpe. Introducir uno (literal `1`) para detenerse tras el final de la iteración cero (`0`), y así sucesivamente.
- `evalIterBegin`, sobrecarga segunda, para los modos soportados para teorías de reescritura (tres y cuatro). Se diferencia en que no toma términos iniciales, ya que se infieren automáticamente para las teorías de reescritura.
    1. Igual que antes, el `Module` con dependencias y aplanado respecto a sus inclusiones.
    2. Igual que antes, el `Module` sin dependencias respecto a sus inclusiones.
    3. El modo de funcionamiento como `Nat`. Se soportan los valores 3 y 4 para esta sobrecarga.
    4. La iteración en la que detenerse (*exclusive*) como `Nat`.
- `evalIterMext`, para todos los modos.
    1. Igual que antes, el `Module` con dependencias aplanado.
    2. Igual que antes, el `Module` sin dependencias.
    3. El `PartialResult` devuelto por la última llamada, cuya iteración final deberá ser la inicial de ésta.
    4. El modo de funcionamiento como `Nat`.
    5. La iteración desde la que se retoma el algoritmo como `Nat`.
    6. La iteración en la que detenerse (*exclusive*) como `Nat`.

También se observan las definiciones de concatenación de los resultados parciales (2), que definen cómo se combinan los resultados parciales de varias operaciones, permitiendo escribir cómodamente la composición de varias iteraciones como si se trataran de los elementos de una lista. De esta forma, una llamada compleja similar a:

```
evalIterBegin(...,0) evalIterNext(...,0,1) evalIterNext(...,1,2)
```

resultaría en el `PartialResult` final resultante de la última iteración ejecutada (en este caso la uno, indicado por el 2 final) si se han pasado a dichas funciones los parámetros adecuados tal y como se han descrito.

A continuación, el operador `evalIter` (4) es llamado implícitamente por las otras funciones y se encarga de serializar los datos intermedios en una cadena codificada que puede ser enviada por la red hacia el cliente. Las llamadas de la API principal también llaman, de forma similar a sus análogas originales `eval`, a las versiones incrementales de `embPE` y `fvpPE`, llamadas aquí `embPEIter` y `fvpPEIter`. Éstas, como ocurría con las originales, llaman a `eqNPEIter`, extensión incremental de EQNPE, que es donde se encuentran las modificaciones principales de la solución diseñada.

El Fragmento 4.1 muestra una comparación de la diferencia en el código fuente entre la implementación original de EQNPE y la extensión incremental desarrollada. Siguiendo la recursividad original, se han insertado los constructores de los resultados parciales `PartialResult` en las partes clave del algoritmo que corresponden con la devolución de un valor (líneas 13 y 14) para que, junto con la concatenación, puedan "arrastar" durante la ejecución del algoritmo la información correspondiente para devolver en caso de una posible detención y permitir su posterior continuación. Una comprobación condicional de las variables `FromIter` y `ToIter`, de las cuales se incrementa la primera, añade una nueva rama en el grafo de control (líneas 10 y 11) que permite la detención condicional del algoritmo cuando se ha alcanzado la iteración objetivo. Las llamadas con puntos suspensivos (. . .) han sido parcialmente omitidas, ocultando los argumentos que reciben.

Por motivos de rendimiento, y porque no resulta práctico al no tener un programa completamente especializado aún, hasta que la operación no ha finalizado con un resultado definitivo se omite completamente cualquier invocación al algoritmo de compresión.

4.3.2. Modificación de la API REST del servicio web

El proceso anterior de ingeniería inversa ha revelado que la API REST expuesta por el servicio web en Java acepta una ruta `eval-program` que recibe una serie de argumentos por parte del cliente con la orden de especialización, el programa y (si corresponde) los términos iniciales. El código subyacente de esta ruta organiza los programas correspondientes y términos en un formato consumible directamente por el intérprete de Maude y adjunta una serie de comandos conductores con la llamada específica al operador correspondiente de la lógica de dominio, es decir, a una sobrecarga de `eval`, como en el ejemplo del Fragmento 3.2.

Se ha extendido esta ruta para poder aceptar los datos relacionados con el modo incremental, extendiendo los parámetros formales de la Tabla 3.1 para exponer ahora los correspondientes con la Tabla 4.1. La modificación es completamente retrocompatible: si no se adjunta ninguno de los nuevos parámetros, `iterative` asume el valor falso por defecto, lo que a su vez causa que se ignore la no existencia de los argumentos relacionados con la iteración (por ejemplo, `fromIter`, `toIter`...). Además, el texto creado y pasado al intérprete Maude utiliza las sobrecargas de `eval` originales al encontrarse desactivado el modo incremental, preservando de forma fiel y exacta el comportamiento original.

Sin embargo, si existe el argumento `iterative` y este contiene el valor verdadero, la nueva implementación asociada a esta ruta generará llamadas a los nuevos operadores. Si la iteración inicial es cero, es decir, se arranca el proceso desde el principio, se construye una llamada a `evalIterBegin` y, en caso contrario, a `evalIterNext`. Si se selecciona el ejemplo incluido en Presto "*Communication Protocol with Mod*" y se ejecuta en modo

Fragmento 4.1: Comparación de la implementación original e incremental de EQNPE

```
op eqNPE : Module TermSet Bool -> RootLeafSet .
eq eqNPE(M,QS,B) =
    [ 0, list2set(...) ] removeEquals(...)
    eqNPE'(M,list2set(...),B,1) .



op eqNPE' : Module TermSet Bool Nat -> RootLeafSet .
eq eqNPE'(M,QS,B,I) =


        if equals?(...) --- si Q' cubre a Q
        then [ I,QS ] unfold(...)
        else [ I, abstract(M,QS,getQSetCandidates(unfold(...))) ] unfold(...)
             eqNPE'(M,abstract(M,QS,getQSetCandidates(unfold(...))),B,I + 1)
        fi
        .
```

```
op eqNPEIter : Module TermSet Bool Nat Nat -> PartialResult .
eq eqNPEIter(M,QS,B,0,ToIter) =                                  --- inicio
    < false ; mt ; [ 0, list2set(...) ] removeEquals(...) >
    eqNPEIter'(M,list2set(...),B,1,ToIter) .
eq eqNPEIter(M,QS,B,FromIter,ToIter) =                           --- continuación
    eqNPEIter'(M,QS,B,FromIter,ToIter) .

op eqNPEIter' : Module TermSet Bool Nat Nat -> PartialResult .
eq eqNPEIter'(M,QS,B,FromIter,ToIter) =
    if (FromIter == ToIter) then < false ; QS ; mt >             --- detención (yield)
    else
        if equals?(...) --- si Q' cubre a Q
        then < true ; QS ; [ FromIter,QS ] unfold(...) >
        else < false ; QS ; [ FromIter, abstract(M,QS,getQSetCandidates(unfold(...))) ] unfold(...) >
             eqNPEIter'(M,abstract(M,QS,getQSetCandidates(unfold(...))),B,FromIter + 1,ToIter)
        fi
    fi .
```

Tabla 4.1: Parámetros formales de la ruta `POST eval-program` de la API del servicio web Presto tras las modificaciones

Nombre	¿Obligatorio?	Tipo	Descripción
`modname`	Sí	Texto	Nombre del módulo padre a especializar.
`program`	Sí	Texto	El programa Maude entero a especializar.
`terms`	Situacional	Texto	Los términos iniciales de especialización o los términos de la última iteración durante el modo incremental. Se requieren cuando se ejecuta una especialización de teoría ecuacional (modos 1, 2 y 3), o cuando se continúa en modo incremental.
`mode`	Sí	Entero	Un indicador del modo seleccionado, indicando si se realiza una especialización de teoría ecuacional, de reescritura o si se asume FVP.
`fullmaude`	Sí	Booleano	Verdadero si el programa requiere Full Maude para ejecutarse, o falso en caso contrario.
`iterative`	No	Booleano	Verdadero si se desea ejecutar el programa en modo incremental.
`fromIter`	Situacional	Entero	La iteración inicial. Únicamente admisible y obligatorio durante el modo incremental.
`toIter`	Situacional	Entero	La iteración final (exclusive). Únicamente admisible y obligatorio durante el modo incremental.
`rlset`	Situacional	Texto	El árbol de *narrowing* anterior. Únicamente admisible y obligatorio durante el modo incremental, excepto en la primera iteración.

incremental, el servicio web generará lo mostrado en la Figura 4.2 para el intérprete de Maude.

El modo fijado es el 3 en correspondencia con el solicitado por el usuario como parte de la selección de este ejemplo. La iteración objetivo es la segunda porque la iteración cero solamente muestra datos triviales de inicialización y de poco interés y, en consecuencia, el cliente siempre pide las iteraciones cero y la uno para la primera ronda de impresión de datos por pantalla para mostrar los resultados parciales. Sin embargo, este podría ser otro número, donde el valor `1` causa que se ejecute estrictamente una única iteración (la cero).

Si el cliente solicita reanudar la ejecución de este proceso, otra llamada a `eval-program` causará que el servicio web construya lo mostrado en la Figura 4.3. Nótese que la parte parcialmente omitida `<false ; ... ; ... >` se corresponde con el `PartialResult` anterior serializado, recibido desde el cliente, y que se corresponde con la tupla de datos técnicos que se arrastran entre llamadas y que permiten dibujar datos de depuración ricos y de alta calidad en los puntos de detención entre iteraciones.

Si el campo `done` indica el valor falso, el cliente sabe que debe indicar que el proceso no ha finalizado, y tiene la garantía de que el resto de campos se muestran como se ha descrito para el caso de una especialización no finalizada. Los campos `qset` y `rlset` contienen la información serializada que debe ser devuelta al servidor para la siguiente iteración, formando el `PartialResult` que se pasará a la siguiente llamada.

Respecto a los datos devueltos al cliente, se ha añadido un nuevo objeto `partial` que incluye los datos correspondientes al nuevo modo incremental. De la misma forma, algunos de los campos anteriores, como el relacionado con el algoritmo de compresión, se devuelven vacíos (por motivos de retrocompatibilidad) al no encontrarse disponibles

```
mod ELP-TOOL is pr PRESTO-ITER .
endm

fmod CAESAR-CIPHER is
    --- omitido: el módulo tal y como se ha introducido
endfm

mod CLI-SRV-PROTOCOL is
    --- omitido: el módulo tal y como se ha introducido
endm

set print format off .
set print conceal on .
print conceal fmod_is_sorts_.____endfm .
print conceal mod_is_sorts_._____endm .
rew in ELP-TOOL :
evalIterBegin(upModule('CLI-SRV-PROTOCOL,true),upModule('CLI-SRV-PROTOCOL,false),3,2) .

quit .
```

Fragmento 4.2: Comandos enviados al intérprete de Maude en el arranque de una especialización incremental

hasta que haya finalizado la especialización. Comparado con lo mostrado en el Fragmento 3.1, los datos devueltos durante una llamada de continuación de una especialización incremental se muestran con el ejemplo del Fragmento 4.4. Nótese la aparición de los campos adicionales y la falta de información para otros. Las señalizaciones en puntos suspensivos al final de los literales de cadena denotan que ésta es más larga, si bien parte ha sido omitida visualmente.

```
{
    "done":false,
    "ori":"fmod STRINGMATCH isELP-NL  inc BOOL .ELP-NL  inc BOOL .ELP-NL  sorts Boo"...,
    "eval":"fmod STRINGMATCH isELP-NL  inc BOOL .ELP-NL  inc BOOL .ELP-NL  sorts Bo"...,
    "evalren":"",
    "total":"",
    "ren":"",
    "debug":"Iter: 0 ELP-NLQSet: ifs match(0 1 nc, M:Message) then (C:CliName <- {S"...,
    "partial":
    {
        "qset":"('ifb_then_else_fi['_=:=_['1.Symbol,'$1:Symbol],'True.Boolean,'Fals"...,
        "rlset":"[ 0 , ('ifs_then_else_fi['match['__['0.Symbol,'__['1.Symbol,'nc.Mes"...
    }
}
```

Fragmento 4.4: Resumen de respuesta decodificada de la ruta `POST eval-program` en modo incremental de la API del servicio web Presto

4.3.3. Modificación del cliente web

El cliente web, que maneja la interfaz de usuario y las llamadas AJAX que consumen la API REST expuesta por el servicio web, ha sido modificado para utilizar las nuevas características expuestas por el servicio. Primero, se ha añadido a la ventana principal de la aplicación una casilla que permite activar y desactivar el modo incremental recién im-

```
mod ELP-TOOL is pr PRESTO-ITER .
endm

fmod CAESAR-CIPHER is
    --- omitido: los mismos módulos sin modificar

set print format off .
set print conceal on .
print conceal fmod_is_sorts_.____endfm .
print conceal mod_is_sorts_._____endm .
rew in ELP-TOOL :
evalIterNext(upModule('CLI-SRV-PROTOCOL,true),upModule('CLI-SRV-PROTOCOL,false),< false
; (('dec['M:Message,'s['s['0.Nat]]]) · ('enc['Q:Message,'s['s['0.Nat]]]) ·
    --- omitido: términos y árbol de narrowing anterior
>,3,2,3) .

quit .
```

Fragmento 4.3: Comandos enviados al intérprete de Maude en la continuación de una especialización incremental

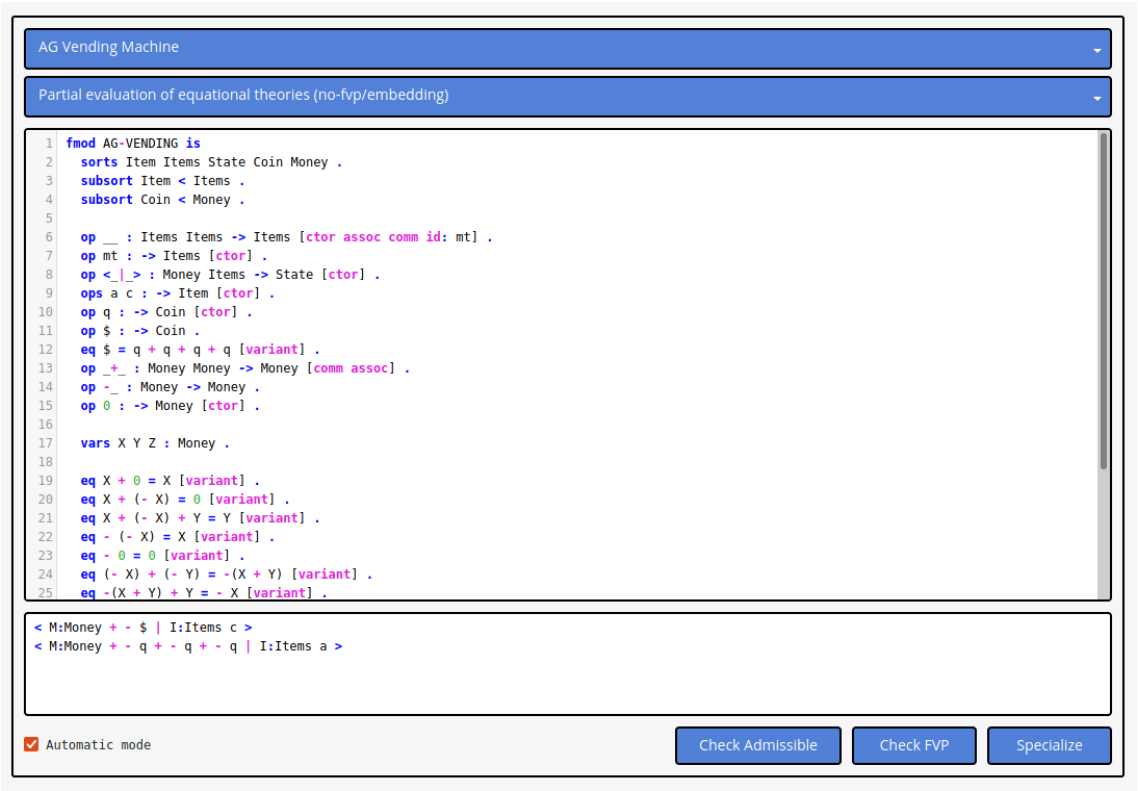
plementado. Tras las correspondientes discusiones sobre experiencia de usuario con los usuarios habituales de la herramienta y expertos del dominio, se ha decidido exponer el nuevo modo incremental con el nombre "Modo manual" en la interfaz, mientras que el comportamiento anterior es ahora el "Modo automático". Para conservar la compatibilidad, por defecto, la herramienta selecciona el "Modo automático", teniendo el usuario que activar las características incrementales de forma explícita si así lo desea. La nueva página principal de la interfaz de muestra en la Figura 4.1, en comparación con la interfaz original en la Figura 3.1. Obsérvese la casilla en la esquina inferior izquierda.

Si se selecciona el modo incremental, la aplicación muestra la pestaña de inspección (*Inspect*) con una advertencia de que algoritmo se encuentra en curso, en lugar de la pestaña de resultado de especialización (*Specialized*), como muestra la Figura 4.2. Las pestañas relacionadas con la compresión (renombrado) se encuentran desactivadas al no encontrarse dicha información disponible hasta alcanzar el final del proceso de especialización. Al completarse dicho proceso, se activan las pestañas con la información correspondientes y se muestra un mensaje de confirmación. Como mejora de calidad de vida, la nueva vista de inspección también muestra dos cajas de texto separadas: para los conjuntos de términos, mostrados en el cuadro superior, y para el árbol de *narrowing*, mostrado en el campo inferior. La vista anterior concatenaba toda esta información en una misma caja de texto.

El botón verde que muestra "*Continuar*" en la parte inferior permite retomar el proceso, mientras que la iteración actual se muestra en la parte superior, dentro del mensaje informativo. Junto al botón que permite retomar el proceso, un control numérico permite seleccionar el número de iteraciones que se desea avanzar, siendo una el valor por defecto. Cada una de las detenciones entre N iteraciones en las que se muestra información y se devuelve control al usuario se llama *ronda*. El número de iteraciones a avanzar introducido es recordado entre rondas, permitiendo avanzar en iteraciones de N en N más rápidamente.

En la caja de texto superior de la pestaña de inspección también aparece, junto a los conjuntos de términos utilizados por cada iteración, una nueva sección encabezada por "`Uncovered leaves`" que contiene la lista de los términos sin cubrir. Dicha información

Figura 4.1: Interfaz de usuario de Presto tras las modificaciones



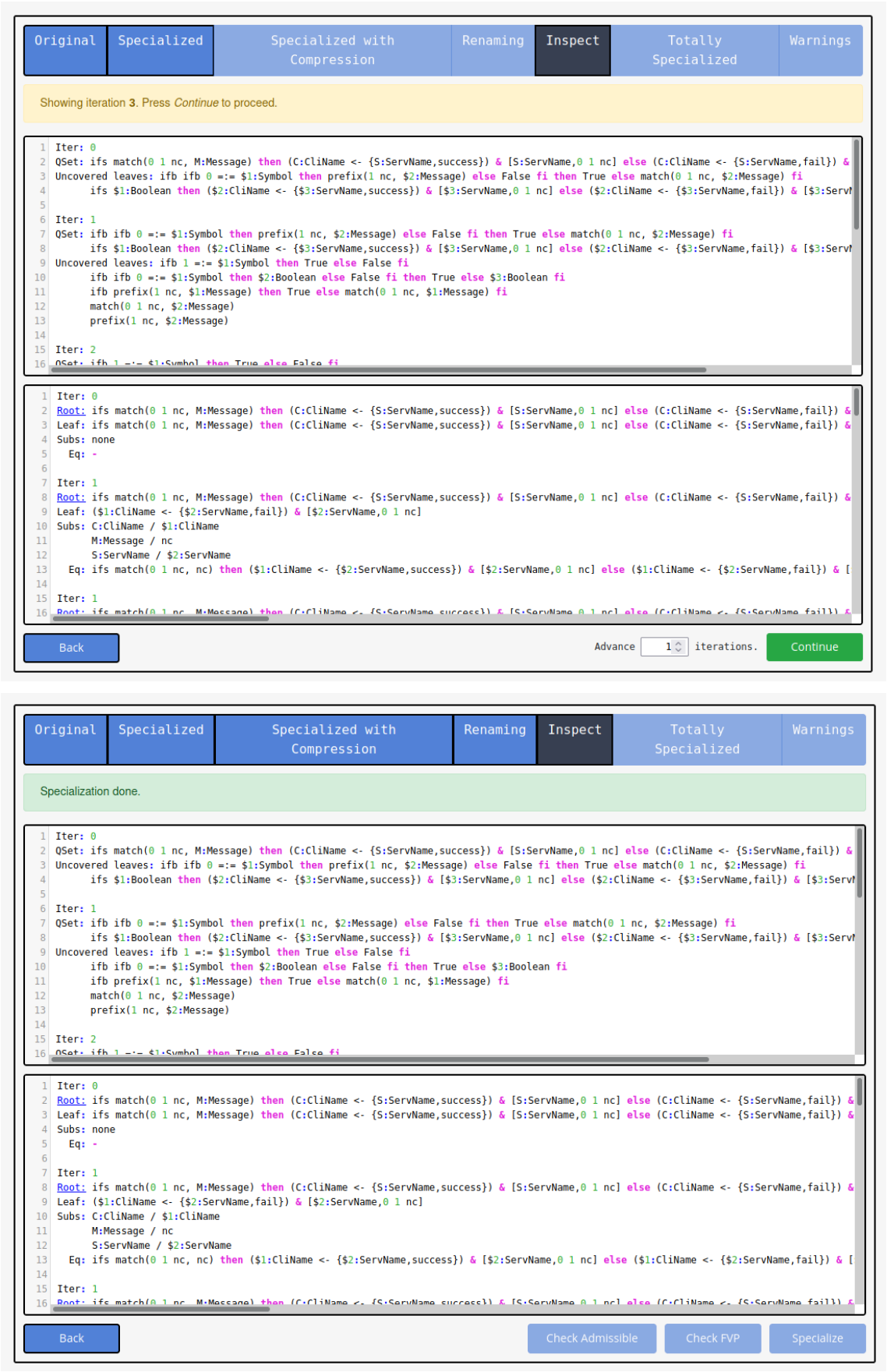
no se encuentra disponible para la última iteración porque requiere de ejecución adicional de la siguiente.

4.3.4. Exploración: marcado gráfico de los términos no cubiertos

De forma adicional a la implementación del modo incremental expuesta anteriormente, se ha realizado una exploración o prueba de concepto temprana de una de las posibilidades de línea de investigación que abre el marco de trabajo de la especialización de programas incremental: el marcado de programas parcialmente especializados. Durante el proceso de especialización, el programa parcialmente especializado no tiene una semántica correcta y no puede ser utilizado en todos los contextos que esperaría el usuario. Mostrarlo como resultado definitivo resulta inseguro, pero mostrarlo como un resultado más, con las correspondientes anotaciones, puede resultar un instrumento muy efectivo para visualizar de forma clara las operaciones realizadas por un algoritmo de especialización incremental dado.

En el caso del algoritmo NPER incremental, el programa parcialmente especializado no es definitivo debido a los llamados *términos sin cubrir*. En esta exploración, se marcan los términos sin cubrir en el propio programa parcialmente especializado de color rojo, permitiendo al usuario visualizar en qué partes están contenidos ciertos términos, y pudiendo así ver en qué localización del programa y en qué contexto se han escrito tales llamadas. Notablemente, no tiene la misma dificultad esclarecer qué hace dicho término cuando este se ve de forma aislada en una traza comparado con su visualización en una regla o ecuación completa junto con otras definiciones de la misma implementación con la que ya se encuentra familiarizado el usuario. Esto permitiría al usuario trazar estos términos sobre el fragmento correspondiente de la implementación original, revelando

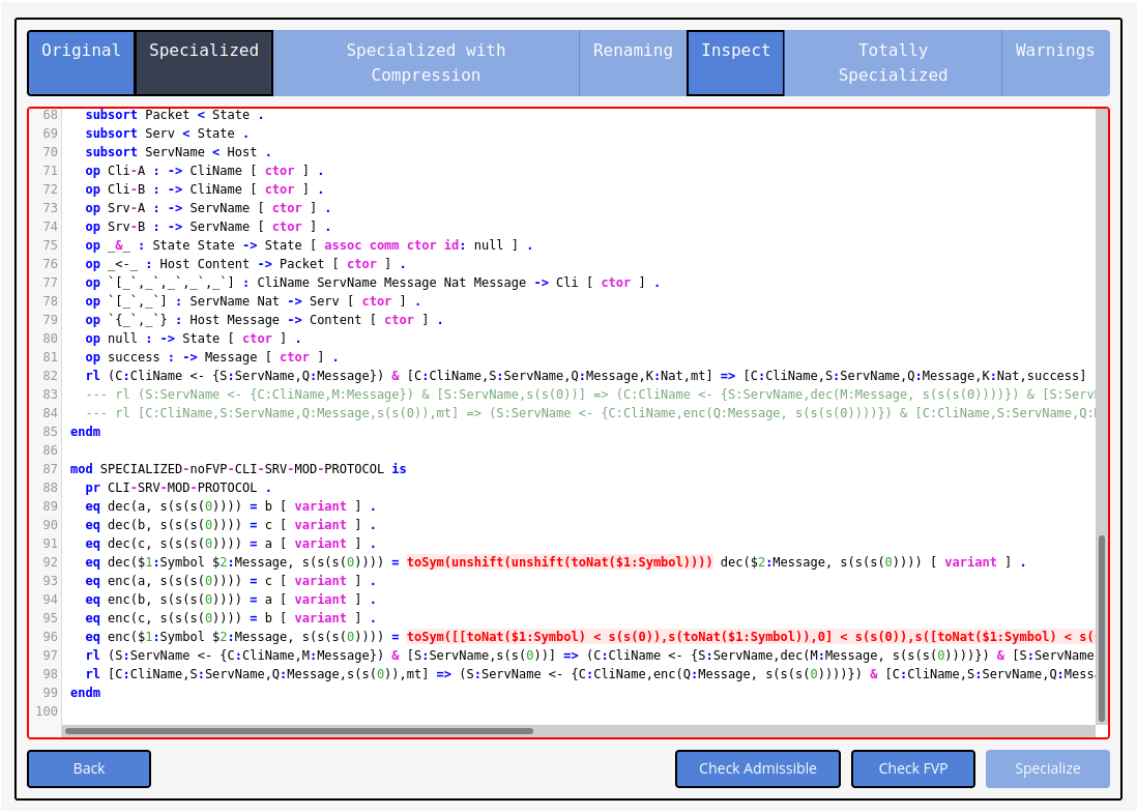
Figura 4.2: Pestaña de inspección en modo incremental tras las modificaciones



detalles como los motivos que llevaron a su implementación, o incluso el nombre de la persona que lo modificó por última vez en un sistema de control de versiones.

El resultado en la interfaz de usuario se muestra en la Figura 4.3. En ella, se marca en color rojo el borde de la caja de texto que contiene dicho código, con el fin de alertar al usuario de que la versión de programa especializado no es válido aún: es una visualización meramente informativa. El usuario debe avanzar por el proceso de especialización hasta su completitud para obtener un resultado completo, correcto y funcional.

Figura 4.3: Marcado gráfico de los términos no cubiertos



El resaltado semánticamente correcto de los términos sin cubrir en este contexto requiere el uso del intérprete de Maude para invocar al analizador semántico para poder comparar si los fragmentos de código Maude mostrados contienen instancias válidas de los términos sin cubrir. Por ejemplo, para evidenciar esta necesidad, considérese que para cada una de las variables las expresiones que se relacionan con ellas deben tener el mismo tipo que el requerido por dichas variables. Esto requiere la resolución de tipos que se encuentra integrada en los algoritmos internos del intérprete de Maude. Esta nueva línea de desarrollo requiere por lo tanto la modificación transversal de la API y la lógica de negocio además de la modificación de la interfaz de usuario, soportada por la correspondiente exploración de la base teórica que la sustenta.

Para mostrar de forma exploratoria y en clave de demostración esta posibilidad, sin embargo, se ha preferido una implementación más simple que utiliza el análisis sintáctico (*parsing*) únicamente en el lado del cliente, lo cual no siempre resulta en la selección de las instancias que realmente se corresponden con términos sin cubrir a nivel semántico. Sin embargo, en la práctica el resultado de este prototipo es suficiente, con una notable tasa de éxito en el marcado de las expresiones relacionadas. La recomendación del autor de este trabajo es la desactivación de esta característica en el caso de un despliegue a producción de la herramienta Presto modificada que pudiera exponer la herramienta a usuarios no expertos.

Para el marcado mediante análisis sintáctico utilizado en esta exploración se han utilizado expresiones regulares que utilizan la coincidencia perezosa (*lazy*) para los "agujeros" representados por las variables. Estas expresiones regulares coinciden únicamente en las partes derechas de las reglas y ecuaciones y tienen en cuenta el símbolo delimitador que bordea las variables por la derecha. Por ejemplo, dada la siguiente lista de términos:

```
<$1:Nat, $2:Nat>
test
```

se generaría la siguiente expresión regular:

```
/(?<=^[^\S\n]*(?:(?:eq\s.*?\s=(?=\s))|(?:rl\s.*\s=>(?=\s))).*?)((?:[^\S\n]<[^,]*?,\s[^>]*?>[^\S\n])|(?:[^\S\n]test[^\S\n]))/gim
```

Se puede explorar la expresión regular generada de forma interactiva en el siguiente enlace: https://regexr.com/6129n.

De forma similar, puede extenderse esta idea para más tipos de anotaciones sobre este marco de trabajo incremental. Pueden utilizarse colores, formas, comentarios, trozos de programa segmentados, etc. en diferentes visores ricos para poder mostrar de forma clara, sencilla y visual al usuario el impacto exacto que tienen sus decisiones de especialización o sus entradas en relación con el algoritmo, pudiendo aprender de forma interactiva el comportamiento de la herramienta ante diferentes situaciones.

4.4 Valoración personal de los resultados

El resultado final constituye una mejora de las ya eficaces funcionalidades que ofrecía la potente herramienta Presto mediante la adición de un nuevo flujo de trabajo que extiende el original con mayores opciones de control y mayor información y retroalimentación del proceso, respetando a su vez la semántica original y la compatibilidad manteniendo soporte exacto a la API tal y como existía antes de emprender este proyecto.

El nuevo modo incremental ofrece la posibilidad de analizar de forma dinámica y a ritmo del usuario la progresión del algoritmo NPER de especialización parcial dirigida por *narrowing*, permitiendo al usuario de forma versátil y pragmática avanzar y detenerse bajo demanda de forma intuitiva en la misma interfaz gráfica de usuaria de la que se disponía anteriormente, respetando los conceptos previos existentes, la nomenclatura y el lenguaje de diseño existentes. Además, se amplía la información de depuración ofrecida y se organiza de una mejor forma, aumentado la calidad de vida y la experiencia de uso, dando pie a un mejor entendimiento del comportamiento de Presto incluso para los tramos de algoritmo que aún no se han ejecutado, con información relativa a los términos no cubiertos que se utilizarán en la próxima iteración.

Las modificaciones realizadas han requerido de un profundo análisis de la algorítmica y la literatura relacionada, además de la ingeniería inversa de la implementación concreta ya existente, que no tenía ninguna documentación técnica asociada y sólo artículos científicos en los que se describen sus bases teóricas. Ha sido necesario tener en cuenta conceptos de mantenimiento del *software* y de integración, al haber sido necesario un ejercicio de precaución a la hora de mantener intacta la semántica original del complejo algoritmo NPER y de asegurar la compatibilidad de los flujos de trabajo que pudieran existir previamente.

De forma adicional, el acercamiento tomado sobre el problema y la solución diseñadas ofrecen de forma versátil una base sólida para posibles mejoras y líneas de investigación en el área de la especialización en lenguajes funcional-lógicos y en especial del lenguaje Maude: el flujo de trabajo puede alojar sin mayor inconveniente la modificación *en caliente* de los parámetros de especialización (o incluso del programa de entrada) de forma

reactiva a los datos mostrados, o incluso la señalización dinámica mediante colores y anotaciones de programas parcialmente especializados, como se trata en la exploración incluida en este capítulo. La investigación de los fundamentos teóricos relacionados con estas mejoras, que escapa al ámbito de este trabajo, puede utilizar el estado actual de Presto como marco de trabajo para el diseño y desarrollo de técnicas incrementales de evaluación parcial enfocadas al usuario.

4.4.1. Breve demostración del nuevo modo incremental

A continuación se utiliza un ejemplo para observar el comportamiento del nuevo modo incremental. El ejemplo en cuestión se encuentra en la sección anexa A.3 con el nombre `nested`, y consiste en una serie de llamadas anidadas para producir una cantidad de iteraciones considerables del algoritmo, aunque éstas son ligeras. En una de las llamadas, sin embargo, se ha insertado de forma intencionada una variación para "dificultar" la progresión del algoritmo, por lo que llegado a dicho punto el algoritmo se ralentiza y por consiguiente una de las iteraciones tarda notablemente más tiempo que las anteriores.

Para ejecutar este ejemplo, es necesario seleccionar el modo "*Partial evaluation of equational theories (no-fvp/embedding)*" e insertar el programa y los términos de especialización correspondientes. A continuación, debe desmarcarse la casilla *Automatic mode* para indicar que se desea utilizar el modo incremental, y presionar el botón *Specialize* para iniciar el proceso. Este ejemplo termina tras catorce iteraciones.

La herramienta mostrará la pestaña *Inspect* con una traza de los conjuntos de términos y los conjuntos de hojas normalizadas sin cubrir en la parte superior. Si se avanza hasta la iteración nueve la traza indica lo siguiente:

```
Iter: 0
QSet: f0(X:Nat, Y:Nat, Z:Nat, H:Nat, I:Nat, J:Nat, K:Nat)
Uncovered leaves: f1(f0($1:Nat, $2:Nat, $3:Nat, $4:Nat, $5:Nat, $6:Nat, $7:Nat), $3:Nat,
$4:Nat, $5:Nat, $6:Nat, $7:Nat)

Iter: 1
QSet: f0(X:Nat, Y:Nat, Z:Nat, H:Nat, I:Nat, J:Nat, K:Nat)
      f1(f0($1:Nat, $2:Nat, $3:Nat, $4:Nat, $5:Nat, $6:Nat, $7:Nat), $3:Nat, $4:Nat,
$5:Nat, $6:Nat, $7:Nat)
Uncovered leaves: f1($1:Nat, $2:Nat, $3:Nat, $4:Nat, $5:Nat, $6:Nat)

Iter: 2
QSet: f0(X:Nat, Y:Nat, Z:Nat, H:Nat, I:Nat, J:Nat, K:Nat)
      f1($1:Nat, $2:Nat, $3:Nat, $4:Nat, $5:Nat, $6:Nat)
Uncovered leaves: f2(f1($1:Nat, $2:Nat, $3:Nat, $4:Nat, $5:Nat, $6:Nat), $3:Nat, $4:Nat,
$5:Nat, $6:Nat)

    --- (NOTA no mostrada en la traza original: iteraciones omitidas)

Iter: 8
QSet: f0(X:Nat,Y:Nat,Z:Nat,H:Nat,I:Nat,J:Nat,K:Nat)
      f1($1:Nat, $2:Nat, $3:Nat, $4:Nat, $5:Nat, $6:Nat)
      f2($1:Nat, $2:Nat, $3:Nat, $4:Nat, $5:Nat)
      f3($1:Nat, $2:Nat, $3:Nat, $4:Nat)
      f4($1:Nat, $2:Nat, $3:Nat)
Uncovered leaves: f5(f4(1 + 1 + 1 + $1:Nat, $2:Nat, $3:Nat), f4($1:Nat, $2:Nat, 1 + 1 + 1
+ $3:Nat))

Iter: 9
QSet: f0(X:Nat,Y:Nat,Z:Nat,H:Nat,I:Nat,J:Nat,K:Nat)
      f1($1:Nat, $2:Nat, $3:Nat, $4:Nat, $5:Nat, $6:Nat)
      f2($1:Nat, $2:Nat, $3:Nat, $4:Nat, $5:Nat)
```

```
      f3($1:Nat, $2:Nat, $3:Nat, $4:Nat)
      f4($1:Nat, $2:Nat, $3:Nat)
      f5(f4(1 + 1 + 1 + $1:Nat, $2:Nat, $3:Nat), f4($1:Nat, $2:Nat, 1 + 1 + 1 + $3:Nat))

--- Specialization not yet complete ---
```

Si se observan los conjuntos de términos, se puede apreciar un patrón regular hasta que se alcanza la llamada `f5(f4(1 + 1 + 1 + ...), ...)`, que marca el inicio de una nueva tendencia y posterior despliegue de numerosas llamadas a `f5`. El análisis de la traza permite averiguar antes de la finalización del algoritmo que la presencia de las sumas `1 + 1 + 1 + ...` asimétricas (el módulo implementa aritmética de Presburger) tiene un impacto significativo sobre el comportamiento temporal del algoritmo en relación con el resto del programa. Esto, combinado con la posibilidad de ver las llamadas que se están desplegando, permite al usuario tomar decisiones como abortar y modificar el programa sin que el algoritmo haya finalizado.

De hecho, si esta llamada en concreto hubiera contenido una llamada a una ecuación con un comportamiento temporal no deseable en la especialización, el algoritmo hubiera alcanzando el límite de tiempo máximo (con el correspondiente mensaje de error) en la iteración ocho, mostrando datos que, de otra forma, hubieran quedado completamente ocultos. Lo mismo se aplica para las otras llamadas, aunque en iteraciones distintas en cada caso.

4.4.2. Revisitando el caso de estudio

Considérese de nuevo el ejemplo de la sección 3.3.4 en el que se ha especializado un comprobador de modelos CTL para un sencillo modelo de ejemplo relacionado con un horno industrial. Tanto el programa como los términos de prueba completos se hallan disponibles en la sección anexa A.1.

Imagínese que el término de especialización introducido, en lugar del mostrado originalmente, fuera el siguiente:

```
(<...>, 1) |= (AG (Not (P:Proposition And (P:Proposition Implies (EX Q:Proposition)))))
```

Nótese que todas las variables son del mismo tipo y que a pesar de que las dos primeras variables son iguales la última es distinta. La propiedad incompleta representada por esta fórmula se puede representar aproximadamente como "*El horno no puede satisfacer la propiedad Q directamente tras participar en un estado que verificó P.*".

Para estas entradas, la especialización en modo automático no finaliza dentro de los límites de tiempo normales establecidos en el actual despliegue de la herramienta en producción (3 minutos). Esto implica que la ejecución consume varios minutos y termina con un error, sin otorgar ninguna información al usuario.

Si se utiliza el modo incremental, sin embargo, aparece la pantalla correspondiente a la iteración uno, mostrando información nueva que antes quedaba oculta. Un usuario de Presto será capaz de evaluar la traza resultante para obtener valiosas pistas del progreso del proceso a partir de la información mostrada en relación con el conocimiento del algoritmo.

CAPÍTULO 5
Generación de casos de prueba

La programación es un proceso que usualmente suele llevar a errores que impiden obtener el comportamiento deseado por el usuario. Esto puede ocurrir al desarrollar los programas como tal, o incluso mientras se describen los modelos y especificaciones que sirven para verificar o realizar ciertas operaciones relacionadas con los métodos formales sobre otros programas. La disciplina de la ingeniería del software ha visto nacer una serie de técnicas diversas y a veces complejas que buscan detectar, comprender y eliminar los errores que ocurren de forma orgánica durante el desarrollo de los productos de software, tanto de forma proactiva como reactiva, dando lugar a procesos como la *depuración* de programas (*debugging*) o la realización de pruebas (*testing*) [1].

Con el objetivo de asistir en los procesos de especialización realizados con la herramienta Presto, se complementa el trabajo realizado sobre Presto con un generador de casos de pruebas en forma de términos bien tipados que representan llamadas a ejecución y que pueden reducirse o reescribirse con el objetivo de probar programas Maude, tanto si se trata del programa original como del especializado. Los generadores de casos de prueba desarrollados en este trabajo son programas Maude cuya ejecución en Maude devolverá los casos de prueba bajo demanda.

5.1 Análisis del problema

Como se ha visto en los capítulos anteriores, la especialización de programas requiere que se fijen ciertos parámetros de entrada a valores conocidos estáticamente y esto tiene como consecuencia que el programa especializado resultante ya no acepte las entradas no contempladas como lo hacía el original, a cambio de las optimizaciones logradas. Aunque el usuario que realiza la especialización tiene diversos mecanismos de control como la posibilidad de marcar y desmarcar ecuaciones con el atributo `variant` y de decidir cuáles son los términos de entrada a utilizar como datos estáticos, en programas lo suficientemente complejos puede llegar a resultar complicado decidir si ciertos términos o casos límite son cubiertos por el programa especializado tal y como lo eran en el original.

Además, cuando se realiza un proceso que obtiene optimizaciones ligadas a mejoras de comportamiento temporal y prestaciones, como es el caso de la especialización, puede resultar de interés realizar pruebas de rendimiento (*benchmarking*) con el objetivo de observar las ganancias relativas que producen diferentes acercamientos a la especialización óptima de un mismo programa original, o de enfocar los detalles de un proyecto de especialización con la finalidad de maximizar las ganancias de rendimiento buscadas.

Son muchas las propiedades que pueden probarse de un programa. En este contexto concreto, es posible limitarse respecto a lo que es deseable probar para un programa Maude especializado, si bien generalmente se busca simultáneamente garantizar:

- **Completitud** (sobre las entradas consideradas): que se obtengan las soluciones esperadas para aquellos términos que se consideran válidos para el programa especializado.

- **Corrección:** que las soluciones obtenidas para aquellos términos válidos sean correctas.

- **Mejora de rendimiento:** que los términos válidos se resuelvan con mejores prestaciones que en el programa original, dentro de unos umbrales que el usuario considere aceptables.

Para poder entender cómo probar de la mejor forma estas propiedades es necesario introducir el concepto de *cobertura*, es decir, la proporción de código respecto del total que se ejecuta con uno o varios casos de prueba dados. En Maude, la "ejecución" de un programa pasa por la consideración de un *término inicial* que causa que se invoquen reglas y ecuaciones que, a su vez, causarán más llamadas, formando un árbol de computación. Por lo tanto, maximizar la cobertura en Maude pasa por encontrar términos iniciales tales que ocasionen árboles que ejecuten la mayor cantidad posible de ecuaciones y reglas del programa, buscando por lo tanto potenciar la *amplitud* de dicho árbol, es decir, que cada ecuación y regla llamada explore el mayor número posible de las diferentes ecuaciones y reglas que puede llegar a llamar.

Teniendo en cuenta que las reglas y ecuaciones en Maude pueden ser condicionales, generar términos iniciales que aseguren cierta cobertura requiere generar series de valores básicos que cumplan o violen estas restricciones de forma organizada, cubriendo las diferentes ramas de decisión del flujo de control, de forma similar a como se haría en otros lenguajes. La exploración manual de estas posibilidades para programas lo suficientemente complejos tiene un aspecto práctico limitado, haciendo deseable alguna forma de automatización que permita generar de forma rápida, sencilla y con bajo coste cierto número considerable de términos iniciales variados que busquen la *amplitud*.

Revisitando el caso de estudio del comprobador de modelos CTL, es posible ahora valorar la necesidad de generar casos de prueba en masa. Si se quisiera realizar una prueba de rendimiento o *benchmark* para probar que una especialización mejora el rendimiento respecto al programa original, se necesitarían crear estructuras de Kripke y fórmulas de tamaño considerable y con ciertas propiedades, ya que para probar el rendimiento se necesitan cargas de trabajo representativas que incurran en tiempos de ejecución no triviales. Con este fin, puede resultar de interés generar términos iniciales que exploren el árbol de llamadas también en *profundidad*, incurriendo en largas cadenas de llamadas que exploren a fondo la repetición del programa, es decir, sus llamadas recursivas y estructuras con repetición, comparables a los bucles de un programa imperativo.

En lugar de empezar a escribir términos de gran tamaño a mano, un posible acercamiento natural a este problema pasa por escribir un segundo programa que genere estructuras de Kripke con un gran número de estados de forma automática. Sin embargo, hacer esto para cada posible programa que se desee probar puede resultar complicado porque requiere una comprensión detallada del programa original, además de un tiempo de desarrollo nada despreciable para dicho generador. ¿Puede automatizarse este proceso a un mayor nivel de abstracción, ahorrando tiempo y costes para el desarrollador?

5.2 Diseño de la solución

Para desarrollar una solución para el problema planteado, primero se ha formalizado un esquema general en forma de canal o *pipeline*, sobre el que se introducen de forma incremental componentes que acercan los datos progresivamente desde un extremo hacia el otro.

Al inicio se dispone de los siguientes componentes:

- **Programa original:** un programa Maude cualquiera que se desee probar, ya sea especializado o no. Dada la complejidad del lenguaje Maude, esta entrada podría restringirse a cierto subconjunto de los programas Maude que se limite a utilizar ciertas características específicas del lenguaje, pero no otras.
- **Casos de prueba:** un conjunto de términos Maude sintácticamente y semánticamente válidos para el programa original dado, tal que si se carga el programa original en un intérprete de Maude y, al menos, uno de estos términos, se obtienen las correspondientes reducciones y reescrituras. Además, estos términos deberían cumplir las propiedades deseables nombradas anteriormente.

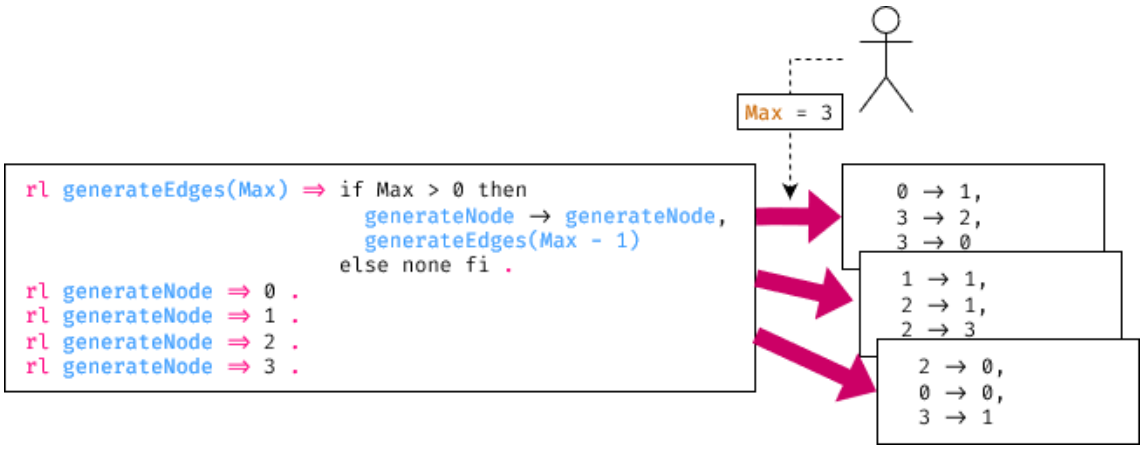
En este capítulo se describe cómo podemos añadir nuevos componentes recorriendo este esquema hacia atrás, es decir, primero buscando una forma de obtener los casos de prueba finales y, más tarde, otro componente de abstracción más alta que pueda generar dicho componente, y así sucesivamente hasta poder aceptar el programa original.

5.2.1. Generador de casos de prueba

Los casos de prueba finales son simplemente un conjunto de términos válidos de Maude que siguen cierta sintaxis descrita por el programa original, siguiendo el mismo patrón, pero completada con datos arbitrarios. Es por lo tanto posible escribir otro programa que genere de forma automática esos términos utilizando dicha sintaxis y cierto componente de aleatoriedad para generar en lote los datos básicos que necesiten los términos.

Dadas las propiedades de metalenguaje de Maude [3], es posible escribir un programa Maude que pueda cargar el programa original y utilizar la información de sus operadores, ecuaciones, reglas, etc. para generar términos que puedan ser aceptados por el mismo. El componente de aleatoriedad puede ser modelado con reglas de reescritura, de forma que las búsquedas sobre dicho programa generarían una serie de soluciones correspondientes a los diferentes casos de prueba de salida. Si esta búsqueda se limita, se obtendrán una serie de casos de prueba aparentemente arbitrarios, pero si esta se ejecuta en profundidad se obtendrá el conjunto de todos los posibles casos de prueba posibles, otorgando la máxima libertad y completitud al usuario. Además, el lenguaje de estrategias de Maude [3] permite controlar la aplicación de las reglas de reescritura para obtener

Figura 5.1: Visión esquemática de un generador de casos de prueba parametrizado

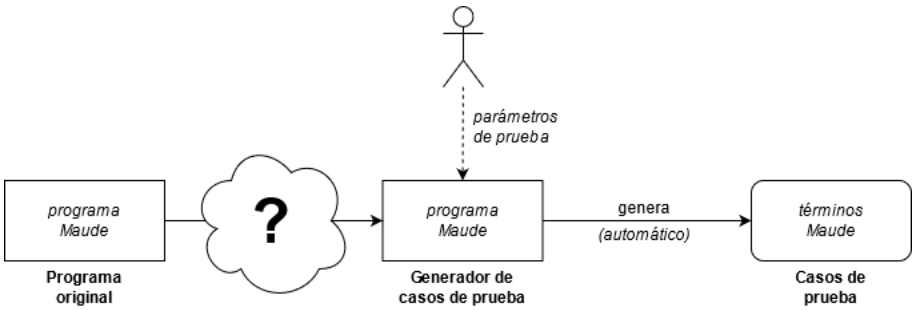


resultados específicos sin necesidad de modificar el código fuente, aumentado aún más las libertades del usuario y evitando el acoplamiento.

Dicho programa Maude capaz de generar términos podría además estar parametrizado, es decir, podría aceptar una serie de datos introducidos por el usuario relacionados con los parámetros de las pruebas que se van a realizar, como la talla deseada de los términos generados, o las características buscadas de los datos básicos presentes en ellos. Esto es adicional a los propios parámetros aceptados por el intérprete de Maude para las búsquedas y al lenguaje de estrategias, por lo que se trata de una posible extensión de las capacidades de control del usuario. En la Figura 5.1 se puede observar un esquema de alto nivel que ilustra esta idea.

Teniendo estas observaciones en cuenta, es posible ahora introducir un nuevo componente definido como sigue:

- **Generador de casos de prueba:** un programa Maude que carga al programa original y utiliza conocimiento específico de este para generar términos sintácticamente y semánticamente válidos. Opcionalmente, puede aceptar datos del usuario para modificar o restringir la naturaleza de los términos generados. En este caso, diremos que el generador se encuentra *parametrizado* y los datos introducidos se llamarán *parámetros de prueba*.



Sin embargo, aunque este generador de casos de prueba puede generar un gran número de casos de prueba de forma completamente automática, utiliza la semántica fuertemente acoplada al programa original, tratándose por lo tanto de una solución específica que no sirve para cualquier programa de entrada. Es necesario por lo tanto continuar depurando esta solución, o bien modificando el generador de casos de prueba para que sea más genérico, o bien introduciendo un nuevo componente capaz de *generar un generador* de casos de prueba a partir de un programa original arbitrario de entrada.

5.2.2. Metagenerador de casos de prueba

Crear un generador de casos de prueba adecuado resulta complicado, ya que es necesario tener un buen entendimiento sobre el problema específico que trata el programa original, así como su dominio. En muchos casos, es posible que haga falta información no directamente disponible en el programa original, como las relacionadas con las restricciones obvias del dominio en cuestión o la valoración de la adecuación de cierta parametrización, y es incluso posible que sea necesario cierto componente de intuición a la hora de tomar decisiones relacionadas. En la práctica, es necesario por lo tanto permitir al usuario, en cierta medida, intervenir en el proceso de elaboración del generador de casos de prueba.

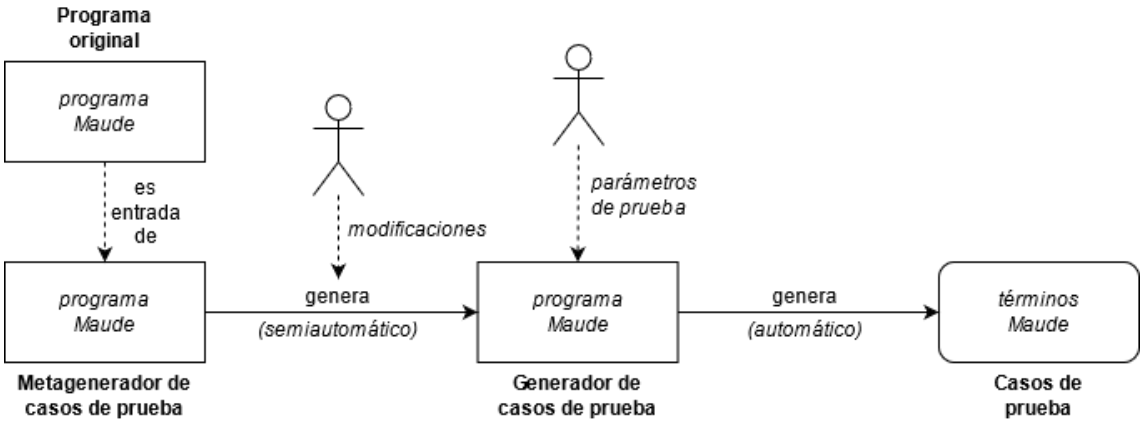
Por otro lado, teniendo en cuenta las capacidades de metalenguaje de Maude, es posible obtener información rica y variada de un programa dado como, por ejemplo, el programa original, y esta información puede utilizarse para generar ciertas estructuras útiles o necesarias para el generador de casos de prueba. La potencial necesidad de intervención del usuario en el proceso de construcción del generador y el hecho de que Maude puede asistir parcialmente en dicho proceso dirige los esfuerzos a un proceso *semiautomático*.

En el mejor de los casos, un análisis avanzado resultaría en un generador de casos de prueba que tiene en cuenta la semántica del problema original de forma adecuada y suficiente y que, por lo tanto, no requiere ninguna intervención manual. En el peor de los casos, un análisis del programa original sería lo suficientemente inconcluso como para que el usuario deba construir el generador de casos de prueba de forma completamente manual. De forma intuitiva, esto favorece un planteamiento de plantilla (*template*): un generador de generadores crea de forma automática un generador incompleto, y el usuario entonces completa esta plantilla manualmente con su conocimiento del dominio.

Se define por lo tanto un nuevo y último componente:

- **Metagenerador de casos de prueba:** un programa Maude que toma como entrada un programa cualquiera y produce como salida una plantilla para un generador de casos de prueba específico para este. El nuevo componente puede verse como un *generador de generadores* de casos de prueba semiautomático.

Figura 5.2: Representación esquemática del proceso de generación de casos de prueba utilizando el metagenerador



El proceso resultante no es, efectivamente, solamente semiautomático, sino que además resulta flexible y extensible. Presentar al usuario un archivo con un programa plantilla le otorga la libertad de modificarlo de forma completamente ilimitada en su entorno

de desarrollo preferente, y utilizando otras herramientas si es necesario. El usuario puede decidir copiar, eliminar o modificar las definiciones según le sea conveniente para cada escenario posible. Este proceso resulta familiar e intuitivo teniendo en cuenta que el usuario de esta herramienta ya conoce el lenguaje Maude, y que las definiciones generadas se basan en el vocabulario y el conocimiento que aparece en la implementación del programa original.

Este diseño también deja un amplio margen para que el análisis pueda ser de varios tipos: estático, dinámico, de caja blanca, de caja negra... sin necesidad de cambiar la arquitectura, ya que las únicas diferencias se verían en una plantilla resultante de mayor o menor calidad. En la sección siguiente sobre la implementación se delimita de forma más específica el alcance y límite del análisis, dejando vía libre a que este pudiera ser extendido o mejorado en el futuro.

5.3 Implementación

El metagenerador desarrollado en este trabajo es un programa Maude que utiliza las características de metanivel de Maude para aceptar un módulo y todos los que este importa (`upModule` [3]), obteniendo su metarepresentación, que luego es analizada de forma sistemática como se describe más adelante. Durante este análisis se genera de forma dinámica, utilizando también la metarepresentación, un nuevo módulo cuyo nombre será `GENERATE-` seguido del nombre del módulo original y que contendrá una serie de operadores y reglas de escritura.

Este nuevo módulo es una plantilla de generador de casos de prueba, aunque aquí la palabra "plantilla" no implica que el programa no pueda ejecutarse directamente: el nuevo módulo generado es un módulo Maude que puede llegar a estar bien formado y ser cargado por el intérprete directamente. Aunque siempre se intenta que esto ocurra, en ciertos casos concretos en los que se utilizan operaciones predefinidas (*built-in*) del intérprete, el módulo generado podría no ser ejecutable sin intervención manual. Es posible que en casos lo suficientemente sencillos la plantilla generada no sea solamente ejecutable, sino que ya sea el generador más adecuado y no requiera modificaciones.

Junto con la lectura de las subsecciones siguientes, puede encontrarse el código fuente del metagenerador en la sección anexa A.4.

5.3.1. Visión general del proceso

Maude expone el módulo `META-LEVEL`, cuyo uso tiene dedicada una sección específica en el manual [3], que contiene el operador `upModule` para elevar módulos al metanivel, obtendiendo su metarepresentación. Esta metarepresentación puede ser explorada utilizando ajuste de patrones de forma habitual, como se haría con cualquier otra estructura, tras consultar la sintaxis de los operadores correspondientes en dicho manual. Se han definido una serie de operadores cuyo nombre empieza por `make`, que recorren progresivamente esta metarepresentación de forma recursiva desde la estructura raíz hacia los diferentes componentes más pequeños que la componen. Para ser específicos, no se recorre todo el árbol sino solamente ciertas partes que son relevantes para el análisis, realizando en realidad varios recorridos dirigidos de varios subárboles, omitiendo por consiguiente ciertas ramas.

Por motivos de rendimiento y simplicidad, se han juntado este recorrido y la generación del nuevo módulo: durante el propio recorrido recursivo, los operadores implicados

devuelven parte del nuevo módulo, y esto implica que, al volver a la llamada raíz, el recorrido habrá terminado y el valor de retorno será el nuevo módulo finalizado.

Finalmente, el nuevo módulo generado, que es una representación en el metanivel, se convierte a una serie de caracteres que forman código fuente Maude válido, y se imprime a la salida estándar el fichero Maude resultante. Durante esta fase se enriquece la salida con un comentario informativo y sangrado para facilitar la lectura del código. En resumen, se realiza un proceso de lo que suele llamarse *pretty-printing* del módulo generado.

Se puede resumir por lo tanto el comportamiento del metagenerador en los siguientes pasos de alto nivel:

1. **Elevación al metanivel:** se utiliza el operador ya definido por Maude `upModule(` *nombre del módulo* `, true)` para obtener la metarepresentación del módulo de entrada y toda la jerarquía de módulos heredados por este, aplanados en un único metamódulo de gran tamaño.

2. **Recorrido del metamódulo:** se recorren recursivamente en ciertas subestructuras de la representación obtenida con los operadores `make`, generando al mismo tiempo la metarepresentación del nuevo módulo correspondiente a la plantilla del generador de casos de prueba.

3. ***Pretty-printing* del nuevo módulo:** se transforma la metarepresentación del nuevo módulo a código fuente Maude con comentarios y sangrado, utilizando el operador `toString`. Finalmente, se imprime a la salida estándar con `print`.
   Ambos operadores han sido implementados por el autor para este trabajo.

5.3.2. Estrategia de análisis

Recordando las características básicas del lenguaje Maude, una teoría define una serie de operadores constructor de tipos y una serie de operadores que actúan como funciones que transforman una serie de datos de entrada en un dato de salida [3]. Luego, a nivel de implementación, una serie de ecuaciones o reglas logran que se realicen las reducciones o reescrituras convenientes. Cuando se generan casos de prueba, se desean recorrer combinaciones de términos sintácticamente válidos, es decir, anidamientos de los diferentes operadores unos dentro de otros, formando expresiones complejas y válidas respecto a sus tipos. Para poder explorar estas combinaciones solamente se necesita, por lo tanto, las definiciones de los operadores (introducidas por `op`), que incluyen la sintaxis y la información de tipos, así como los atributos y axiomas asociados.

Por ejemplo, considérese las siguientes definiciones de operadores:

```
op false : -> Bool [ctor] .
op true : -> Bool [ctor] .
op _and_ : Bool Bool -> Bool .
op _or_ : Bool Bool -> Bool .
```

Es posible generar combinaciones de aquellos términos que pueden reducirse o reescribirse al *sort* `Bool`, sin tener las ecuaciones y reglas que verdaderamente implementan las operaciones descritas:

```
false
true
true and false
```

```
false or false
(false and true) or true
...
```

Como se puede observar, las expresiones que se pueden formar con estos operadores son infinitas, y pueden explorarse con varias estrategias: aunque aquí se presentan varias combinaciones aleatorias, también pueden explorarse minuciosamente todas las posibles en anchura, hasta cierta profundidad límite. Esto otorga una base para realizar una primera generación de términos de prueba basada únicamente en las posibles entradas, útil para pruebas de caja negra. Esto implica también que la cobertura del programa original se intenta obtener a base de la generación exhaustiva de combinaciones, con la esperanza de que las suficientes combinaciones descubran posibles casos límite de interés.

Un sencillo programa Maude que genera términos como los descritos arriba es el siguiente:

```
1 mod GENERATE-BOOL is
2     op generateBool : -> Bool .
3
4     rl generateBool => true .
5     rl generateBool => false .
6     rl generateBool => generateBool and generateBool .
7     rl generateBool => generateBool or generateBool .
8 endm
```

Fragmento 5.1: Ejemplo de generador de combinaciones de términos

Las reglas de reescritura de Maude permiten describir estas expresiones de forma intuitiva, centrada en su sintaxis, tal y como lo haría una gramática EBNF [34]. Además, como ya se ha visto anteriormente, el lenguaje de estrategias permite a un usuario controlar cómo se aplicar estas reglas utilizando comandos de nivel de intérprete, sin cambiar este código fuente.

Para generar este módulo de forma automática a partir de uno con los operadores, es necesario definir diferentes operaciones y conceptos sobre los elementos del lenguaje Maude. Para explicar el algoritmo con sencillez, se introduce el concepto intuitivo e informal de patrones con agujeros (*holes*) que se rellenan para formar términos. Dada una definición de operador, como por ejemplo `op [_;_] : Nat Bool -> Tuple .`, debido al significado especial del carácter "_" [3], es conocido que pueden construirse términos válidos que invoquen a las respectivas ecuaciones y reglas si se escriben cadenas que siguen el patrón `[ ○ ; ○ ]`, donde ○ es un "agujero" que representa una subexpresión que devuelve el tipo esperado para su posición. Dada una definición de operador c, sea su representación como *patrón* con agujeros el resultado de $patrón(c)$. Es implica que un patrón rellenado completamente con valores, como por ejemplo `[ 2 ; true ]`, es un *término* según las definiciones habituales en Maude.

Para cada agujero existe un *sort* correspondiente a devolver para superar la comprobación de tipos. Esto es, siguiendo el ejemplo anterior, para el primero se debe devolver `Nat` y para el segundo `Bool`. Esto implica la generación de una subexpresión de forma recursiva. Con el objetivo de delegar la generación de una expresión capaz de devolver tal valor, se *rellena* cada agujero con la llamada a una regla llamada `generate` seguida del nombre del *sort* correspondiente. En este caso, el resultado de dicha operación es `[ generateNat ; generateBool ]`. Además, la definición también esclarece que el *sort*

devuelto como resultado de esta expresión es `Tuple`. Por ello, es posible añadir una nueva regla en el generador, declarando así una nueva forma de generar `Tuples`. De forma correspondiente, el nombre deberá ser `generateTuple` para que pueda ser utilizada en los agujeros de otras expresiones que requieran una subexpresión de este *sort*:

```
rl generateTuple => [ generateBool ; generateNat ] .
```

De esta forma, para cada definición de operador (`op`) del programa original puede generarse una regla. Más tarde esta relación se extiende desde el *uno a uno* al *uno a muchos*, añadiendo la posibilidad de generar más de una regla por cada definición.

Sea C el conjunto de todas las definiciones de operador c_i del programa original, extendiendo el ejemplo propuesto, puede definirse el siguiente algoritmo:

Algoritmo 5: Generación de reglas que generan combinaciones de unos operadores dados

Entrada: Un conjunto de definiciones de operador C.
Salida: Un conjunto de reglas R que generan las combinaciones exhaustivas de los términos válidos según los operadores en C.

$R \leftarrow \emptyset$
para $c \leftarrow C$ **hacer**
  $p \leftarrow$ patrón(c)
  $h \leftarrow \emptyset$
  para $x \longleftarrow$ agujeros(p) **hacer**
    $h \leftarrow h \cup$ (”`generate`” + nombreDelSort(x))
  fin
  $t \leftarrow$ rellenar(p, h)
  $n \leftarrow$ ”`generate`” + sortDevuelto(c)
  $r \leftarrow$ crear regla (`rl` n `=>` t `.`)
  $R \leftarrow R \cup r$
fin
devolver R

Los operadores utilizados como entrada se extraen del conjunto de operadores del módulo original. Tras la ejecución del algoritmo, las reglas resultantes se añaden a un nuevo módulo de sistema, junto con las correspondientes definiciones de operador (`op`) para que puedan ser utilizadas. La signatura de tipos de dichas definiciones no toma ningún parámetro de entrada y devuelve el *sort* a generar, con el mismo nombre que las reglas, `generate`*Sort*, coincidiendo con el ejemplo presentando en el Fragmento 5.1.

Teniendo en cuenta este algoritmo, se puede observar que el análisis es: estático, porque no ejecuta el programa original; y que realiza una sobreaproximación, ya que intenta devolver todas las posibles reglas válidas al usuario para que sea este el encargado de filtrar aquellas que le resulten de utilidad. A pesar de las modificaciones y extensiones descritas en las subsecciones siguientes, el análisis conserva estos rasgos.

Comparación con otras herramientas

Es relevante comparar la estrategia seguida con la implementado por la herramienta MSTCG creada por Adrián Riesco[35], basada en ejecución simbólica y con un propósito similar de generar automáticamente casos de prueba para programas Maude. Esta herramienta realiza un acercamiento notablemente más sofisticado que utiliza ejecución simbólica para detectar y marcar las diferentes definiciones de cierto programa Maude,

garantizando así cierto nivel de cobertura determinado a partir del conocimiento que relaciona el código Maude con las diferentes condiciones que deben cumplir sus variables para poder ser alcanzado. A continuación, la herramienta utiliza *narrowing* para evaluar las expresiones alcanzables y finalmente determinar conjuntos de valores que permiten alcanzar de forma exhaustiva los diferentes estados del programa original. El resultado final es un conjunto reducido de datos concretos que resultan en un recorrido dirigido que explora de forma completa el programa original.

El proceso presentado en este trabajo, sin embargo, es más comparable con una técnica que busca combinaciones de entradas aceptables a nivel semántico, resultando en sendos conjuntos de datos de menor ambición respecto a la cobertura, de forma similar al comportamiento de los llamados *fuzzers* de caja blanca [36]. También se puede comparar esta aproximación a la realizada por el generador de casos de prueba automático aleatorio basado en especificaciones *QuickCheck*, desarrollado originalmente para Haskell por Koen Claessen y John Hughes [37] y más tarde extendido a otros lenguajes.

A diferencia de técnicas como las de A. Riesco, o Claessen y Hughes, el objetivo de los juegos de datos objeto de este trabajo no es la depuración de programas (garantizando automáticamente un buen cubrimiento de los patrones de llamada). Por el contrario, tanto el programa original como el programa especializado se asumen correctos y lo que se desea obtener son juegos de datos pesados y dirigidos a evaluar experimentalmente la optimización producida por el evaluador parcial Presto y el mayor rendimiento del programa especializado. Más allá de las pruebas, la técnica aquí planteada utiliza una forma más simple y rápida de generación de datos, donde para cierto tiempo de trabajo prima la cantidad de términos generados. Esto permite extenderse a otras áreas como la generación de lotes de datos para *benchmarking*. El usuario de Presto es un usuario experto que requiere un gran control sobre el generador automático para parametrizarlo a su conveniencia y controlar la generación de forma rigurosa y versátil para el uso que se haya planteado.

5.3.3. Importación y operadores intrínsecos (*built-in*)

En el lenguaje Maude, los módulos soportan una característica llamada importación, con la que un módulo puede traer las definiciones de otro, distribuyendo una teoría compleja en varias unidades lógicas con dependencias explícitas y jerarquizadas. En el código fuente, esta característica se observa con las directivas `including`, `extending` y `protecting` [3].

Esta característica es relevante en el proceso descrito anteriormente porque cambia la forma en la que se eleva el módulo original al metanivel. La operación proporcionada por Maude `upModule` pide un segundo argumento además del nombre del módulo a elevar, que es un valor booleano que indica si se desean elevar de forma transitiva los módulos importados por el módulo objetivo. Como es bastante habitual que un programa Maude se encuentre dividido en varios módulos, es importante que se decida que sí que se desean cargar los módulos importados transitivamente para asegurar la utilidad de la herramienta.

Sin embargo, esta elección conlleva un nuevo problema: ¿qué ocurre si el usuario ha decidido importar un módulo del preludio, como `NAT`, que utiliza operadores predefinidos (*built-in*) del intérprete? Al contrario que los operadores definidos por el usuario, que siempre definen todos los posibles constructores de tipos de forma explícita, los operadores intrínsecos a veces dependen de literales soportados por el lenguaje o bien de datos externos. Por ejemplo, un número natural (`Nat`) es formado por un literal, como `13`, y al existir un soporte intrínseco por parte del lenguaje no hay operadores con que recorrer

con los posibles infinitos literales. Resulta imposible crear un conjunto de reglas completo `generateNat` con el algoritmo anterior.

Es relativamente sencillo para un usuario importar otros módulos que acaben utilizando utilidades del preludio en algún punto de su cadena de importaciones. Es vital por lo tanto tratar los operadores intrínsecos de forma explícita. La estrategia que se ha tomado, ante la gran cantidad de operadores intrínsecos ofrecidos por el preludio de Maude, es la de implementar una restricción general que evite generar reglas para aquellos operadores que cumplen una serie de condiciones que se asocian con el soporte intrínseco del intérprete y que, por lo tanto, rompen las invariantes esperadas por el algoritmo desarrollado.

Las reglas utilizadas para decidir si un operador tiene soporte intrínseco son las siguientes:

- Se considera intrínseco si tiene el atributo `special`, lo que indica que se requiere el uso de una función escrita en el lenguaje de implementación del propio intérprete (*hook* nativo).
- Se considera intrínseco si retorna el *sort* `Universal`, que es el *top sort*. Es decir, `Universal` es un *sort* tal que todos los demás son un *subsort* de este. La justificación de esta regla es que resulta imposible crear un valor para este *sort*, ya que este podría ser cualquier valor y, en consecuencia, se desconoce qué subconjunto de toda la explosión combinatoria de valores posibles es relevante para cada caso concreto.
- Se considera intrínseco si retorna un *sort* prohibido explícitamente, como `Nat` o `Bool`.

Puede observarse con más detalle la implementación en el operador `makeRlsBuiltIn`, implementado en el módulo `MAUDE-CHECK-BASE` del metagenerador (sección anexa A.4).

Esta decisión tiene sin embargo una desventaja: aunque el algoritmo anterior permite generar plantillas que siempre son ejecutables desde un primer momento, si se utilizan de forma directa o indirecta operadores intrínsecos del preludio ya no resulta posible asegurar que la plantilla generada sea un programa completamente ejecutable. Por ejemplo, imagínese que una regla necesita llamar a `generateNat`, pero no existe implementación para ésta por las razones argumentadas anteriormente. Sería necesario que el usuario desarrollara entonces una o varias reglas específicas para su caso capaces de generar números naturales.

El impacto de esta desventaja se reduce añadiendo reglas específicas y desarrolladas a mano para ciertos casos comunes, actuando como una opción "por defecto" que facilita el trabajo de obtener un generador completo cuando esta implementación ya es adecuada. Estas reglas se añaden en un módulo a parte llamado `GENERATION-HELPERS`. Por ejemplo, para los números naturales se ofrece una regla `generateNat` que se reescribe a `random(counter)`, definido en el preludio, que genera un número natural aleatorio. El módulo ofrece este tipo de implementaciones *ad-hoc* para los *sorts* `Nat`, `Float`, `Rat`, `Int`, `Bool`, `Char` y `String`.

5.3.4. Refinamiento con heurísticas

Con el objetivo de mejorar la calidad de las plantillas generadas, se ha modificado el algoritmo desarrollado para permitir el procesamiento de operadores según una serie de heurísticas. Cada una de éstas comprueba si un operador $c \in C$ cumple una serie de

condiciones que lo alinean con cierto caso particular y, en caso de verificarse, éstas generan las reglas correspondientes siguiendo un algoritmo especializado. Véase el operador `makeRlsH` implementado en el módulo `MAUDE-CHECK-HEURISTICS` del metagenerador (sección anexa A.4), para más información.

Operadores que representan constructores de secuencias, multiconjuntos o conjuntos

En el lenguaje Maude, las declaraciones de secuencias, multiconjuntos y conjuntos son similares:

```
sort Items .
sort Item .
subsort Item < Items .

op empty : -> Items [ctor] .

op _._ : Items Items -> Items [ctor assoc id(empty)] . --- secuencia
op _;_ : Items Items -> Items [ctor assoc comm id(empty)] . --- multiconjunto
op _,_ : Items Items -> Items [ctor assoc comm idem id(empty)] . --- conjunto
```

Esto es, se refiere a aquellos operadores que tienen esta forma:

op $\boxed{P}$: $\boxed{S}$ $\boxed{S}$ -> $\boxed{S}$ [assoc id($\boxed{I}$) $\boxed{\textit{atributos...}}$] .

De forma habitual, en la plantilla se generarían las reglas siguientes para el operador `_,_`:

```
rl generateItems => generateItems , generateItems .
```

Esto es poco ideal porque resulta en una explosión combinatoria al explorarse el espacio de resultados con el comando `search`, que busca en amplitud. Para una búsqueda en amplitud, además de intuitivo, resulta conveniente generar lo siguiente, que recuerda a una gramática EBNF recursiva por la derecha [34]:

```
rl generateItems => generateItem , generateItems .
rl generateItems => empty .
```

Esto se logra con una heurística. De la misma forma que puede realizarse esta aproximación, pueden añadirse nuevas heurísticas al metagenerador que refinen progresivamente las plantillas generadas.

5.3.5. Extensión de la generación con las declaraciones de *subsorts*

A veces, un *sort* puede estar definido pero no tener ningún constructor de tipos que lo genere, porque existe una definición de *subsort*, o varias de ellas, que indican todos los posibles valores que puede adoptar dicho *sort*. Por ejemplo, considérese el siguiente escenario:

```
sorts Zero' NzNat' Nat' .   --- NzNat' es un natural no cero
subsort Zero' < Nat' .
subsort NzNat' < Nat' .
```

```
op 0 : -> Zero' .
op s : Nat' -> NzNat' .
```

Aquí, esta definición de los números naturales en notación de Peano no define ningún operador que genere un `Nat'`. ¿Significa esto que es imposible generar términos que resulten en este tipo? No, porque existen declaraciones de *subsort*. Es necesario por lo tanto extender el algoritmo inicial para no dejar *sorts* sin reglas, siguiendo en la misma línea de la sobreaproximación a una plantilla en la que el usuario pueda encontrar todas las posibles reglas útiles.

Para ello, se recorren todas las declaraciones de *subsorts* del módulo original, de forma independiente, y se generan reglas tal que el súper-*sort* genere al *sort* derivado, para cada declaración `subsort`. De esta forma, del ejemplo anterior se derivarán de forma adicional las siguientes reglas:

```
rl generateNat' => generateZero' .
rl generateNat' => generateNzNat' .
```

5.4 Valoración personal de los resultados

Se puede ilustrar la necesidad de la generación de términos de prueba iniciales revisitando el mismo ejemplo presentado en la Sección 3.3.4, un comprobador de modelos que utiliza la lógica de árboles de computación (CTL). Este ejemplo es ilustrativo porque las entradas necesarias para ejecutar este programa no son sencillas de generar a mano debido a su complejidad intrínseca, especialmente si éstas son grandes, por lo que se enfatizan los beneficios obtenidos con una posible automatización en la generación de los términos iniciales.

Es necesario crear estructuras de Kripke y fórmulas de tamaño considerable y con ciertas propiedades, de forma automática. En lugar de escribir un generador capaz de hacer esto a mano, se va a ejecutar el metagenerador de casos de prueba con el módulo `OVEN` presentado anteriormente, que introduce las proposiciones `hot`, `open` y `working` para modelar un horno. Este módulo importa el comprobador de modelos CTL, por lo que traerá todas las definiciones sobre el manejo de fórmulas CTL y estructuras de Kripke.

El resultado de la ejecución del metagenerador resulta en la plantilla de generador etiquetada `generate-oven-template` y disponible en la sección anexa A.5. Esta plantilla ya es un programa Maude ejecutable y bien formado que genera ciertas estructuras como, por ejemplo, las fórmulas, de forma correcta. Sin embargo, las estructuras de Kripke generadas no son válidas al ser el grafo subyacente no conexo, es decir, este puede contener varios componentes fuertemente conexos separados. Es esperable que a la plantilla le falte tal restricción porque esta necesidad no se ha expresado de ninguna forma en el programa original. Por lo tanto, es natural que estas restricciones dejadas al criterio humano luego requieran intervención del usuario cuando se ejecutan procesos automatizados.

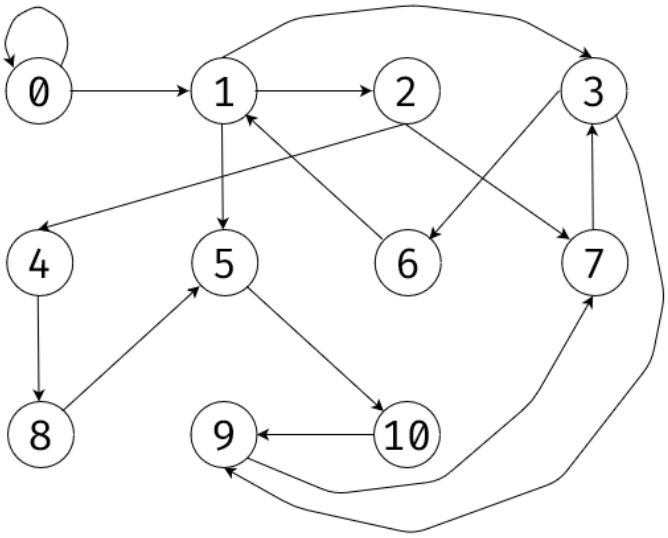
Se ha modificado la plantilla para asegurar que las estructuras de Kripke generadas son conexas y tienen una forma "representativa", es decir, que parecen naturalmente intrincadas, con bucles o estados que pueden ser visitados desde varios lugares. Para ello se ha utilizado la operación módulo para conectar los estados entre sí con cierta pseudoaleatoriedad controlada según su índice. Además, se ha añadido una cota explícita a algunas de las reglas para poder controlar la talla de una estructura de Kripke en función de su número de estados, demostrando la capacidad de parametrización descrita ante-

riormente. Finalmente, el resultado (programa `generate-oven`) se encuentra en la sección anexa A.6.

Es interesante concretar algunas métricas relativas a este ejemplo. De la plantilla se han extraído 28 reglas útiles. De ellas, se han modificado 8 y se han mantenido sin modificar 20 (71 %). De estas 8 modificaciones, 5 (18 % del total) se consideran triviales ya que se han realizado para aceptar la cota de estados y pasarla a otras reglas, mientras que 3 reglas (11 % del total) han sufrido cambios semánticos importantes para adaptarlas a los criterios de generación de las estructuras de Kripke.

En conclusión, el generador de casos de prueba resultante es capaz de generar estructuras de Kripke adecuadas y comparables a aquellas que resultan de interés en escenarios planteados de forma manual, como la que se muestra primero de forma gráfica y luego textual a continuación:

Figura 5.3: Estructura de Kriple sin etiquetas generada con `rewrite generateSystem(11) .`.




```
result System: < 0 -> 0, 0 -> s(0), s(0) -> s(s(0)), s(0) -> s(s(s(0))),
s(0) -> s(s(s(s(s(0))))), s(s(0)) -> s(s(s(s(0)))), s(s(0)) -> s(s(s(s(s(s(s(0)))))))),
s(s(s(0))) -> s(s(s(s(s(s(0)))))), s(s(s(0))) -> s(s(s(s(s(s(s(s(s(0)))))))),
s(s(s(s(0)))) -> s(s(s(s(s(s(s(s(0)))))))), s(s(s(s(s(0))))) ->
s(s(s(s(s(s(s(s(s(s(0))))))))))), s(s(s(s(s(s(0)))))) -> s(0), s(s(s(s(s(s(s(0)))))))) ->
s(s(s(0))), s(s(s(s(s(s(s(s(0)))))))) -> s(s(s(s(s(0))))), s(s(s(s(s(s(s(s(s(0)))))))))
-> s(s(s(s(s(s(s(0)))))))), s(s(s(s(s(s(s(s(s(s(0))))))))))) ->
s(s(s(s(s(s(s(s(s(0)))))))))) ; empty >
```

Al igual que aquí la talla solicitada es 11, puede solicitarse una estructura de cualquier otra talla, sin límite más allá que el impuesto por los recursos subyacentes disponibles.

Además, también es posible la generación de fórmulas. El siguiente comando nos provee con 25 fórmulas rápidamente:

```
Maude> search [25] generateFormula =>* F:Formula .

Solution 1 (state 16)
F:Formula --> False

Solution 2 (state 17)
F:Formula --> True

Solution 3 (state 18)
```

```
F:Formula --> hot

--- ...

Solution 13 (state 508)
F:Formula --> True Xor (E[True U True Xor hot])

--- ...

Solution 25 (state 1715)
F:Formula --> E[True U working]
```

El metagenerador no solamente resulta de utilidad por el hecho de que las plantillas puedan agilizar el proceso de crear generadores de términos de prueba: un punto importante del metagenerador es que este recorre de forma exhaustiva el espacio de operadores y *sorts* definidos por el usuario en su programa. Así, resulta trivial para un usuario discernir cuáles son todos los casos potencialmente relevantes para él, a nivel del dominio, sin necesidad de realizar una comprensión del código fuente Maude, que es potencialmente más complicada. Por ejemplo, tal vez alguien no sea consciente de que una fórmula puede llegar a formarse con el operador `EX` y, sin utilizar el metagenerador, sería necesario recorrer todo el código fuente del programa original para confirmar tal detalle, atendiendo a detalles de implementación en lugar de mantenerse en un alto nivel de abstracción. Para ese momento podrían haberse generado ya miles de términos de prueba sin percatarse el interesado de que ninguno de ellos contiene el combinador `EX`, y perdiendo por lo tanto cobertura del programa original.

En definitiva, se considera que el metagenerador cumple con los objetivos propuestos dentro del alcance esperado. En este capítulo se plantean, además un amplio campo de mejoras que pueden aumentar la calidad de la herramienta en distintas direcciones, abriendo nuevas oportunidades para los desarrolladores e investigadores de la comunidad Maude.

CAPÍTULO 6
Conclusiones

La herramienta Presto permite la especialización automática, versátil y eficiente de teorías ecuacionales así como de teorías de reescritura del lenguaje Maude mediante su algoritmo NPER, basado en la técnica de *narrowing*. Esto permite obtener versiones extremadamente optimizadas para programas de complejidad y tamaño no triviales como un comprobador de modelos basado en la lógica CTL. Una arquitectura basada en un servicio web REST permite la ejecución de la lógica de dominio cómodamente a través de la red utilizando una interfaz gráfica de usuario que se ejecuta en cualquier navegador.

El desarrollo de un nuevo esquema de especialización automática incremental basado en la extensión del algoritmo NPER realizada en este trabajo ha permitido extender la implementación de la herramienta Presto de forma retrocompatible, armónica, responsable y proporcional con la implementación original para incluir un nuevo flujo de trabajo que permite la detención condicional, controlada y bajo demanda del proceso de especialización en ciertos puntos clave, permitiendo analizar información relacionada con los resultados parciales obtenidos por el algoritmo para acercarse a su resultado objetivo. Este nuevo flujo de trabajo se basa en la segmentación del algoritmo monolítico original en varias funciones que delegan el flujo de control al usuario, quien puede controlar al nivel de granularidad deseado el progreso del algoritmo mediante la nueva interfaz proporcionada. Este nuevo modo de funcionamiento es ortogonal respecto a las decisiones de especialización, integrándose con el uso previo de la herramienta y respetando las líneas conductoras de su diseño.

El marco de trabajo desarrollado permite la exploración de nuevas líneas de trabajo que utilicen el mismo tipo de control o extiendan la misma implementación, como la posibilidad de poder modificar las decisiones de especialización *en caliente* sin necesidad de reiniciar el proceso, o la construcción y anotación dinámica y progresiva de los programas especializados.

Por otra parte, el diseño de un nuevo paradigma de generación de términos Maude para pruebas experimentales, basado en un metagenerador capaz de generar generadores parametrizables vinculados a programas Maude, permite la creación en masa de términos aceptables por el programa objetivo que cubren de forma ordenada y predecible sus operadores definidos. El uso de comandos del lenguaje Maude permite la generación bajo demanda de términos de entrada de tamaño significativo para programas reales siguiendo los patrones deseados, reduciendo de forma notable un coste que, de forma manual, sería inasumible. La presentación de la herramienta como un programa Maude discreto que genera archivos editables en Maude permite su uso de forma ilimitadamente extensible e intuitiva en contextos que van más allá de los planteados originalmente, desligándose de la herramienta Presto y cubriendo un gran abanico de potenciales necesidades que puedan darse en la comunidad Maude.

Bibliografía

[1] P. Bourque, R. Dupuis, A. Abran, J.W. Moore, and L. Tripp. The guide to the software engineering body of knowledge. *IEEE Software*, 16(6):35–44, 1999. doi: 10.1109/52.805471.

[2] P. Gibbins. Chapter 13 - what are formal methods? In *The Software Life Cycle*, pages 278–290. Butterworth-Heinemann, 1990. ISBN 978-0-408-03741-9. doi: 10.1016/B978-0-408-03741-9.50020-9.

[3] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott. *Maude Manual (Version 3.1)*, October 2020. URL http://maude.lcc.uma.es/maude31-manual-html/maude-manual.html.

[4] M. Alpuente, A. Cuenca-Ortega, S. Escobar, and J. Meseguer. A partial evaluation framework for order-sorted equational programs modulo axioms. *J. Log. Algebraic Methods Program.*, 110, 2020. doi: 10.1016/j.jlamp.2019.100501.

[5] M. Alpuente, S. Escobar, J. Sapiña, and D. Ballis. Presto website. http://safe-tools.dsic.upv.es/presto/, 2021. (last visited 2021-06-01).

[6] M. Alpuente, S. Escobar, J. Sapiña, and D. Ballis. *Presto Quick Start Guide*. VRAIN, Universitat Politècnica de València, and DMIF, University of Udine, 2021. URL http://safe-tools.dsic.upv.es/presto/quickstart.pdf.

[7] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, September 1993. ISBN 978-0-13-020249-9.

[8] M. Leuschel, M. Varea, M. Fontaine, B. Martens, J. Jørgensen, D. de Schreye, R. Glück, M.H. Sorensen, A. de Waal, and G. Vidal. ECCE website. http://wyvern.cs.uni-duesseldorf.de/ecce/index.php, 2021. (last visited 2021-06-03).

[9] M. Leuschel and J.F. Morales. Logen repository on GitHub. https://github.com/leuschel/logen, 2021. (last visited 2021-06-03).

[10] E.M. Clarke, O. Grumberg, and D. Peleg. *Model Checking*. MIT Press, 1999. ISBN 978-0-26-203270-4.

[11] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: A tutorial. *J. UCS*, 12:1618–1650, January 2006.

[12] R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. A declarative debugger for Maude functional modules. *Electronic Notes in Theoretical Computer Science*, 238(3):63–81, 2009. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2009.05.013.

[13] S. Escobar, C. Meadows, and J. Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design*

V, FOSAD 2007/2008/2009 Tutorial Lectures, pages 1–50, January 2007. ISBN 978-3-642-03828-0. doi: 10.1007/978-3-642-03829-7_1.

[14] S. Antoy. Programming with narrowing: A tutorial. *Journal of Symbolic Computation*, 45(5):501–522, 2010. ISSN 0747-7171. doi: 10.1016/j.jsc.2010.01.006.

[15] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. *Electronic Notes in Theoretical Computer Science*, 117:417–441, 2005. ISSN 1571-0661. doi: 10.1016/j.entcs.2004.06.020.

[16] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Model Checking Strategy-Controlled Rewriting Systems (System Description). In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:18, 2019. ISBN 978-3-95977-107-8. doi: 10.4230/LIPIcs.FSCD.2019.34.

[17] M. Scott. *Programming language pragmatics (4th ed.)*. Morgan Kaufmann, January 2015. ISBN 978-0-12-410409-9.

[18] M. Alpuente, D. Ballis, S. Escobar, J. Meseguer, and J. Sapiña. Narrowing-based optimization of rewrite theories. In *7th Int'l Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE)*, 2020.

[19] H. Comon-Lundh and S. Delaune. The finite variant property: How to get rid of some algebraic properties. In *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005. doi: 10.1007/978-3-540-32033-3_22.

[20] Eclipse Foundation. Jakarta RESTful Web Services website. https://projects.eclipse.org/projects/ee4j.jaxrs, 2021. (last visited 2021-06-19).

[21] Java Community Process. Java API for RESTful Web Services (JAX-RS 2.1) specification. https://jcp.org/en/jsr/detail?id=370, August 2017.

[22] The Bootstrap team. Bootstrap website. https://getbootstrap.com/, 2021. (last visited 2021-06-19).

[23] OpenJS Foundation. jQuery website. https://jquery.com/, 2021. (last visited 2021-06-19).

[24] OpenJS Foundation. jQuery documentation: ajax. https://api.jquery.com/Jquery.ajax/, 2021. (last visited 2021-06-19).

[25] Mozilla. Mozilla Developer Network documentation: encodeURIComponent. https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent, 2021. (last visited 2021-06-19).

[26] Marijn Haverbeke. CodeMirror website. https://codemirror.net/, 2021. (last visited 2021-06-19).

[27] Oracle. Java documentation: Overriding and hiding methods. https://docs.oracle.com/javase/tutorial/java/IandI/override.html, 2021. (last visited 2021-06-19).

[28] Microsoft. C# reference: override modifier. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override, 2020. (last visited 2021-06-19).

[29] M. Leuschel and S. Gruner. Abstract conjunctive partial deduction using regular types and its application to model checking. In *Logic Based Program Synthesis and Transformation*, pages 91–110. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-45607-0. doi: 10.1007/3-540-45607-4_6.

[30] J.B. Rosenberg. *How debuggers work: algorithms, data structures, and architecture*. John Wiley & Sons, Inc., 1996.

[31] D.E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997. ISBN 0201896834.

[32] Microsoft. C# reference: yield (contextual keyword). https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/yield, 2020. (last visited 2021-06-21).

[33] Microsoft. Python 3 reference: yield statement. https://docs.python.org/3/reference/simple_stmts.html#the-yield-statement, 2021. (last visited 2021-06-21).

[34] ISO/IEC 14977:1996. Information technology — Syntactic metalanguage — Extended BNF, 1996.

[35] A. Riesco. Using semantics specified in maude to generate test cases. In *Theoretical Aspects of Computing – ICTAC 2012*, pages 90–104. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32943-2. doi: 10.1007/978-3-642-32943-2_7.

[36] V. Pham, M. Böhme, and A. Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 543–553. Association for Computing Machinery, 2016. ISBN 9781450338455. doi: 10.1145/2970276.2970316.

[37] K. Claessen and J. Hughes. Quickcheck website. http://www.cse.chalmers.se/~rjmh/QuickCheck/, 2021. (last visited 2021-06-25).

APÉNDICE A
Programas y términos de prueba

A.1 (ctl) Comprobador de modelos CTL

ctl.maude

```
fmod TRUTH' is
    sort Bool' .

    ops True False : -> Bool' .

    op _==_ : Bool' Bool' -> Bool' [comm] .

    eq True  == True  = True [variant] .
    eq False == False = True [variant] .
    eq False == True  = False [variant] .
endfm

fmod NAT' is
    pr TRUTH' .

    sorts Zero' Nat' NzNat' .
    subsorts Zero' NzNat' < Nat' .

    op 0 : -> Zero' [ctor] .
    op s : Nat' -> NzNat' [ctor] .

    ops 1 2 3 4 5 6 7 8 9 10 : -> NzNat' [ctor] .

    op _+_ : NzNat' Nat' -> NzNat' [assoc comm prec 33] .
    op _+_ : Nat' Nat' -> Nat' [ditto] .

    op _==Nat_ : Nat' Nat' -> Bool' [comm] .

    var N M : Nat' .

    eq N + 0    = N [variant] .
    eq N + s(M) = s(N + M) [variant] .

    eq 0    ==Nat 0    = True [variant] .
    eq 0    ==Nat s(N) = False [variant] .
    eq s(N) ==Nat s(M) = N ==Nat M [variant] .

    eq 1  = s(0) [variant] .
    eq 2  = s(s(0)) [variant] .
    eq 3  = s(s(s(0))) [variant] .
    eq 4  = s(s(s(s(0)))) [variant] .
    eq 5  = s(s(s(s(s(0))))) [variant] .
    eq 6  = s(s(s(s(s(s(0)))))) [variant] .
    eq 7  = s(s(s(s(s(s(s(0))))))) [variant] .
```

```
45      eq 8  = s(s(s(s(s(s(s(s(0)))))))) [variant] .
46      eq 9  = s(s(s(s(s(s(s(s(s(0))))))))) [variant] .
47      eq 10 = s(s(s(s(s(s(s(s(s(s(0)))))))))) [variant] .
48  endfm
49
50  fmod PROPOSITIONAL-CALCULUS is
51      pr TRUTH' .
52
53      sorts Proposition Formula .
54      subsort Proposition < Formula .
55      subsort Bool' < Formula .
56
57      ops True False : -> Bool' .
58      op Not_        : Formula -> Formula [prec 53] .
59      op _And_       : Formula Formula -> Formula [assoc comm prec 55] .
60      op _Or_        : Formula Formula -> Formula [assoc comm prec 59] .
61      op _Xor_       : Formula Formula -> Formula [assoc comm prec 57] .
62      op _Implies_   : Formula Formula -> Formula [gather(e E) prec 61] .
63
64      vars A B C : Formula .
65      eq True And A       = A [variant] .
66      eq False And A      = False [variant] .
67      eq A And A          = A [variant] .
68      eq False Xor A      = A [variant] .
69      eq A Xor A          = False [variant] .
70      eq A And (B Xor C)  = A And B Xor A And C [variant] .
71      eq Not A            = A Xor True [variant] .
72      eq A Or B           = A And B Xor A Xor B [variant] .
73      eq A Implies B      = Not (A Xor A And B) [variant] .
74  endfm
75
76  fmod PROPOSITIONAL-SATISFACTION is
77      pr PROPOSITIONAL-CALCULUS .
78
79      sort Statements .
80
81      op _|=_ : Statements Formula -> Bool' .
82
83      var Z : Statements . vars A B : Formula .
84      eq Z |= (A And B)   = (Z |= A) And (Z |= B) [variant] .
85      eq Z |= (A Xor B)   = (Z |= A) Xor (Z |= B) [variant] .
86      eq Z |= True        = True [variant] .
87      eq Z |= False       = False [variant] .
88  endfm
89
90  fmod CTL-CALCULUS is
91      pr PROPOSITIONAL-CALCULUS .
92
93      op AX_    : Formula -> Formula [prec 63] .
94      op EX_    : Formula -> Formula [ctor prec 63] .
95      op AF_    : Formula -> Formula [prec 63] .
96      op EF_    : Formula -> Formula [prec 63] .
97      op AG_    : Formula -> Formula [prec 63] .
98      op EG_    : Formula -> Formula [prec 63] .
99      op A[_U_] : Formula Formula -> Formula [prec 63] .
100     op E[_U_] : Formula Formula -> Formula [ctor prec 63] .
101
102     op NotEG_ : Formula -> Formula [ctor prec 63] .
103
104     vars K L : Formula .
105     eq AX K     = Not (EX Not K) [variant] .
106     eq AF K     = NotEG (Not K) [variant] .
107     eq EF K     = E[True U K] [variant] .
108     eq AG K     = Not (EF (Not K)) [variant] .
```

```
109     eq EG K     = Not (NotEG K) [variant] .
110     eq A[K U L] = (Not (E[Not L U ((Not K) And (Not L))])) And (NotEG L) [variant] .
111 endfm
112
113 fmod CTL-SYSTEM is
114     pr PROPOSITIONAL-CALCULUS .
115     pr NAT' .
116
117     sorts State Transition Label .
118     subsort Nat' < State .
119
120     op [_:_] : State Proposition -> Label [ctor] .
121     op _->_  : State State -> Transition [ctor] .
122
123     sort TransitionSet .
124     subsort Transition < TransitionSet .
125     op empty     : -> TransitionSet [ctor] .
126     op _,_       : TransitionSet TransitionSet -> TransitionSet [ctor assoc comm id:
    empty prec 121 format (d r os d)] .
127     op transFrom : State TransitionSet -> TransitionSet .
128
129     sort LabelSet .
130     subsort Label < LabelSet .
131     op empty : -> LabelSet [ctor] .
132     op _,_   : LabelSet LabelSet -> LabelSet [ctor assoc comm id: empty prec 121 format
    (d r os d)] .
133
134     sort System .
135     op <_;_>     : TransitionSet LabelSet -> System [ctor] .
136
137     vars TS TS' : TransitionSet . vars T : Transition .
138     var LS : LabelSet . vars L L' : Label . vars P P' : Proposition .
139     vars S S' A B : State .
140
141     op _==_ : State State -> Bool' [comm] .
142     eq S == S' = S ==Nat S' [variant] .
143
144     op _==_ : Proposition Proposition -> Bool' [comm] .
145     eq P == P  = True [variant] .
146
147     op _==_ : Label Label -> Bool' [comm] .
148     eq [S : P] == [S' : P'] = (S == S') And (P == P') [variant] .
149
150     op _in_ : Label LabelSet -> Bool' .
151     eq L in (L', LS) = (L == L') Or (L in LS) [variant] .
152     eq L in empty    = False [variant] .
153
154     op _in_ : Label System -> Bool' .
155     eq L in < TS ; LS > = L in LS [variant] .
156
157     op if : Bool' TransitionSet TransitionSet -> TransitionSet [frozen(2 3)] .
158     eq if(True,  TS, TS') = TS [variant] .
159     eq if(False, TS, TS') = TS' [variant] .
160
161     op transFrom : State TransitionSet -> TransitionSet .
162     eq transFrom(S, (A -> B, TS)) = if(A == S,
163                                        (A -> B, transFrom(S, TS))
164                                     ,
165                                        transFrom(S, TS)
166                                     ) [variant] .
167     eq transFrom(S, empty)        = empty [variant] .
168 endfm
169
170 fmod CTL-SATISFACTION is
```

```
    pr CTL-SYSTEM .
    pr CTL-CALCULUS .
    pr PROPOSITIONAL-SATISFACTION .

    op (_,_) : System State -> Statements .

    var M : System . var S : State .
    vars K L : Formula . var P : Proposition .
    eq (M,S) |= P               = [S : P] in M [variant] .
    eq (M,S) |= (EX K)          = anyTransition(M, S, K) [variant] .
    eq (M,S) |= (E[K U L])      = ((M,S) |= L) Or (
                                      ((M,S) |= K) And anyTransition(M, S, E[K U L]) )
[variant] .
    eq (M,S) |= (NotEG K)       = ((M,S) |= (Not K)) Or
                                      allTransitions(M, S, NotEG K) [variant] .

    op anyTransition  : System State Formula -> Bool' .
    op allTransitions : System State Formula -> Bool' .

    vars TS TSf : TransitionSet . var S' : State .
    var LS : LabelSet .

    eq anyTransition(< TS ; LS >, S, K) =
        anyTransitionHelper(transFrom(S, TS), < TS ; LS >, K) [variant] .
    eq allTransitions(< TS ; LS >, S, K) =
        allTransitionsHelper(transFrom(S, TS), < TS ; LS >, K) [variant] .
    eq allTransitions(< empty ; LS >, S, K) = False [variant] .

    op anyTransitionHelper : TransitionSet System Formula -> Bool' .
    eq anyTransitionHelper((S -> S', TSf), < S -> S', TS ; LS >, K) =
        ((< TS ; LS >, S') |= K) Or anyTransitionHelper(TSf, < TS ; LS >, K) [variant] .
    eq anyTransitionHelper(empty, < TS ; LS >, K) = False [variant] .

    op allTransitionsHelper : TransitionSet System Formula -> Bool' .
    eq allTransitionsHelper((S -> S', TSf), < S -> S', TS ; LS >, K) =
        ((< TS ; LS >, S') |= K) And allTransitionsHelper(TSf, < TS ; LS >, K) [variant]
.
    eq allTransitionsHelper(empty, < TS ; LS >, K) = True [variant] .
endfm
```

A.1.1. Especialización para el caso del horno industrial

Extiéndase el programa anterior con lo siguiente:

```
fmod OVEN is
    protecting CTL-SATISFACTION .

    op open : -> Proposition .
    op working : -> Proposition .
    op hot : -> Proposition .

    eq open == working  = False [variant] .
    eq open == hot      = False [variant] .
    eq working == hot   = False [variant] .
endfm
```

Términos para la especialización primera:

```
(< 1 -> 2, 2 -> 1, 2 -> 3, 3 -> 4, 4 -> 2, 4 -> 5, 5 -> 4, 5 -> 1 ; [1 : open], [3 :
working], [4 : hot], [5 : open], [5 : hot] >, S:State) |= (AG (Not (open And (open
Implies (EX P:Proposition)))))
```

Términos para la especialización segunda (incremental que no finaliza):

```
(< 1 -> 2, 2 -> 1, 2 -> 3, 3 -> 4, 4 -> 2, 4 -> 5, 5 -> 4, 5 -> 1 ; [1 : open], [3 :
working], [4 : hot], [5 : open], [5 : hot] >, 1) |= (AG (Not (P:Proposition And
(P:Proposition Implies (EX Q:Proposition)))))
```

A.2 (specialized-oven) Especializado: comprobador de modelos para un horno industrial

```
fmod TRUTH' is
  inc BOOL .
  sort Bool' .
  op False : -> Bool' .
  op True : -> Bool' .
  op _==_ : Bool' Bool' -> Bool' [ comm ] .
endfm

fmod PROPOSITIONAL-CALCULUS is
  inc BOOL .
  pr TRUTH' .
  sorts Formula Proposition .
  subsort Bool' < Formula .
  subsort Proposition < Formula .
  op False : -> Bool' .
  op Not_ : Formula -> Formula [ prec 53 ] .
  op True : -> Bool' .
  op _And_ : Formula Formula -> Formula [ assoc comm prec 55 ] .
  op _Implies_ : Formula Formula -> Formula [ prec 61 gather (e E) ] .
  op _Or_ : Formula Formula -> Formula [ assoc comm prec 59 ] .
  op _Xor_ : Formula Formula -> Formula [ assoc comm prec 57 ] .
endfm

fmod NAT' is
  inc BOOL .
  pr TRUTH' .
  sorts Nat' NzNat' Zero' .
  subsort NzNat' < Nat' .
  subsort Zero' < Nat' .
  op 0 : -> Zero' [ ctor ] .
  op 1 : -> NzNat' [ ctor ] .
  op 10 : -> NzNat' [ ctor ] .
  op 2 : -> NzNat' [ ctor ] .
  op 3 : -> NzNat' [ ctor ] .
  op 4 : -> NzNat' [ ctor ] .
  op 5 : -> NzNat' [ ctor ] .
  op 6 : -> NzNat' [ ctor ] .
  op 7 : -> NzNat' [ ctor ] .
  op 8 : -> NzNat' [ ctor ] .
  op 9 : -> NzNat' [ ctor ] .
  op _+_ : Nat' Nat' -> Nat' [ assoc comm prec 33 ] .
  op _+_ : NzNat' Nat' -> NzNat' [ assoc comm prec 33 ] .
  op _==Nat_ : Nat' Nat' -> Bool' [ comm ] .
  op s : Nat' -> NzNat' [ ctor ] .
  eq 1 = s(0) [ variant ] .
  eq 10 = s(s(s(s(s(s(s(s(s(0)))))))))) [ variant ] .
  eq 2 = s(s(0)) [ variant ] .
  eq 3 = s(s(s(0))) [ variant ] .
  eq 4 = s(s(s(s(0)))) [ variant ] .
  eq 5 = s(s(s(s(s(0))))) [ variant ] .
  eq 6 = s(s(s(s(s(s(0)))))) [ variant ] .
```

```
52    eq 7 = s(s(s(s(s(s(s(0)))))))  [ variant ] .
53    eq 8 = s(s(s(s(s(s(s(s(0))))))))  [ variant ] .
54    eq 9 = s(s(s(s(s(s(s(s(s(0)))))))))  [ variant ] .
55    eq 0 + N:Nat' = N:Nat' [ variant ] .
56    eq N:Nat' + s(M:Nat') = s(N:Nat' + M:Nat') [ variant ] .
57  endfm
58
59  fmod CTL-SYSTEM is
60    inc BOOL .
61    pr PROPOSITIONAL-CALCULUS .
62    pr NAT' .
63    sorts Label LabelSet State System Transition TransitionSet .
64    subsort Label < LabelSet .
65    subsort Nat' < State .
66    subsort Transition < TransitionSet .
67    op <_;_> : TransitionSet LabelSet -> System [ ctor ] .
68    op _->_ : State State -> Transition [ ctor ] .
69    op _==_ : Label Label -> Bool' [ comm ] .
70    op _==_ : Proposition Proposition -> Bool' [ comm ] .
71    op _==_ : State State -> Bool' [ comm ] .
72    op _`,_ : LabelSet LabelSet -> LabelSet [ assoc comm ctor id: (empty).LabelSet prec
  121 format (d r os d) ] .
73    op _`,_ : TransitionSet TransitionSet -> TransitionSet [ assoc comm ctor id:
  (empty).TransitionSet prec 121 format (d r os d) ] .
74    op _in_ : Label LabelSet -> Bool' .
75    op _in_ : Label System -> Bool' .
76    op `[_:_`] : State Proposition -> Label [ ctor ] .
77    op empty : -> LabelSet [ ctor ] .
78    op empty : -> TransitionSet [ ctor ] .
79    op if : Bool' TransitionSet TransitionSet -> TransitionSet [ frozen (2 3) ] .
80    op transFrom : State TransitionSet -> TransitionSet .
81  endfm
82
83  fmod CTL-CALCULUS is
84    inc BOOL .
85    pr PROPOSITIONAL-CALCULUS .
86    op AF_ : Formula -> Formula [ prec 63 ] .
87    op AG_ : Formula -> Formula [ prec 63 ] .
88    op AX_ : Formula -> Formula [ prec 63 ] .
89    op A`[_U_`] : Formula Formula -> Formula [ prec 63 ] .
90    op EF_ : Formula -> Formula [ prec 63 ] .
91    op EG_ : Formula -> Formula [ prec 63 ] .
92    op EX_ : Formula -> Formula [ ctor prec 63 ] .
93    op E`[_U_`] : Formula Formula -> Formula [ ctor prec 63 ] .
94    op NotEG_ : Formula -> Formula [ ctor prec 63 ] .
95  endfm
96
97  fmod PROPOSITIONAL-SATISFACTION is
98    inc BOOL .
99    pr PROPOSITIONAL-CALCULUS .
100   sort Statements .
101   op _|=_ : Statements Formula -> Bool' .
102 endfm
103
104 fmod CTL-SATISFACTION is
105   inc BOOL .
106   pr CTL-SYSTEM .
107   pr CTL-CALCULUS .
108   pr PROPOSITIONAL-SATISFACTION .
109   op _`,_ : System State -> Statements .
110   op allTransitions : System State Formula -> Bool' .
111   op allTransitionsHelper : TransitionSet System Formula -> Bool' .
112   op anyTransition : System State Formula -> Bool' .
113   op anyTransitionHelper : TransitionSet System Formula -> Bool' .
```

```
endfm

fmod OVEN is
  inc BOOL .
  pr CTL-SATISFACTION .
  op hot : -> Proposition .
  op open : -> Proposition .
  op working : -> Proposition .
endfm

fmod SPECIALIZED-noFVP*-OVEN is
  pr OVEN .
  op f0 : State Proposition -> [Formula] .
  eq < 1 -> 2,2 -> 1,2 -> 3,3 -> 4,4 -> 2,4 -> 5,5 -> 1,5 -> 4 ; [1 : open],[3 :
working],[4 : hot],[5 : hot],[5 : open] >,S:State |= (AG Not (open And (open Implies (EX
P:Proposition)))) = f0(S:State, P:Proposition) [ variant ] .
  eq f0(0, $1:Proposition) = True [ variant ] .
  eq f0(s(0), hot) = False [ variant ] .
  eq f0(s(0), open) = False [ variant ] .
  eq f0(s(0), working) = True [ variant ] .
  eq f0(s(s(0)), hot) = False [ variant ] .
  eq f0(s(s(0)), open) = False [ variant ] .
  eq f0(s(s(0)), working) = True [ variant ] .
  eq f0(s(s(s(0))), hot) = False [ variant ] .
  eq f0(s(s(s(0))), open) = False [ variant ] .
  eq f0(s(s(s(0))), working) = True [ variant ] .
  eq f0(s(s(s(s(0)))), hot) = False [ variant ] .
  eq f0(s(s(s(s(0)))), open) = False [ variant ] .
  eq f0(s(s(s(s(0)))), working) = True [ variant ] .
  eq f0(s(s(s(s(s(0))))), hot) = False [ variant ] .
  eq f0(s(s(s(s(s(0))))), open) = False [ variant ] .
  eq f0(s(s(s(s(s(0))))), working) = True [ variant ] .
  eq f0(s(s(s(s(s(s($1:Nat')))))), $2:Proposition) = True [ variant ] .
endfm
```

A.3 (nested) Ejemplo: llamadas anidadas

Términos de especialización:

```
f0(X:Nat,Y:Nat,Z:Nat,H:Nat,I:Nat,J:Nat,K:Nat)
```

Programa:

nested.maude

```
fmod NESTED is
  sort Nat .

  ops 0 1 : -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat [ctor assoc comm id: 0] .

  vars X Y Z H I J K : Nat .

  op f0 : Nat Nat Nat Nat Nat Nat Nat -> Nat .
  eq f0(0,Y,Z,H,I,J,K) = 0 [variant] .
  eq f0(1 + X,Y,Z,H,I,J,K)  = f1(f0(X,Y,Z,H,I,J,K),Z,H,I,J,K) [variant] .

  op f1 : Nat Nat Nat Nat Nat Nat -> Nat .
  eq f1(0,Y,Z,H,I,J) = 0 [variant] .
  eq f1(1 + X,Y,Z,H,I,J)  = f2(f1(X,Y,Z,H,I,J),Z,H,I,J) [variant] .

  op f2 : Nat Nat Nat Nat Nat -> Nat .
```

```
18   eq f2(0,Y,Z,H,I) = 0 [variant] .
19   eq f2(1 + X,Y,Z,H,I)  = f3(f2(X,Y,Z,H,I),Z,H,I) [variant] .
20
21   op f3 : Nat Nat Nat Nat -> Nat .
22   eq f3(0,Y,Z,H) = 0 [variant] .
23   eq f3(1 + X,Y,Z,H)  = f4(f3(X,Y,Z,H),Z,H) [variant] .
24
25   op f4 : Nat Nat Nat -> Nat .
26   eq f4(0,Y,Z) = 0 [variant] .
27   eq f4(1 + 1 + 1 + X,1 + 1 + 1 + Y,1 + 1 + 1 + Z) = f5(f4(1 + 1 + 1 + X,Y,Z),f4(X,Y,1 +
   1 + 1 + Z)) [variant] .
28
29   op f5 : Nat Nat -> Nat .
30   eq f5(0,Y) = 0 [variant] .
31   eq f5(1 + X,Y) = f6(f5(X,Y)) [variant] .
32
33   op f6 : Nat -> Nat .
34   eq f6(0) = 0 [variant] .
35   eq f6(1 + X) = f6(X) [variant] .
36
37 endfm
```

A.4 (maude-check) Metagenerador de casos de pruebas

maude-check.maude

```
1  load file.maude
2
3  fmod MAUDE-CHECK-BASE is
4      pr META-LEVEL .
5
6      var Q : Qid .
7      var H : Header .
8      var M : Module .
9      vars S SC SP : Sort .
10     var Ss : SortSet .
11     var T : Term .
12     var Op : OpDecl .
13     var Ops : OpDeclSet .
14     var SubS : SubsortDecl .
15     var SubSs : SubsortDeclSet .
16     vars Ty Ty' : Type .
17     var Tys : TypeList .
18     var K : Kind .
19     vars At At' : Attr .
20     var Ats : AttrSet .
21     var N : String .
22     var Hooks : HookList .
23
24     --- MODULE
25
26     op makeMod : Header -> Module .
27     eq makeMod(H) = makeMod(H, upModule(H, true)) .
28
29     op makeMod : Header Module -> Module .
30     eq makeMod(H, M) =
31         mod makeModName(H) is
32             (protecting H .)
33             sorts none .
34             none    --- SubsortDeclSet
35             makeOps(getSorts(M), M)
36             none    --- MembAxSet
37             none    --- EquationSet
```

```
38              makeRlsForOps(getOps(M), M)
39              makeRlsForSubsorts(getSubsorts(M), M)
40          endm .
41
42      op makeModName : Header -> Header .
43      eq makeModName(H) = qid("GENERATE-" + string(H)) .
44
45      --- OPERATORS
46
47      op makeOps : SortSet Module -> OpDeclSet .
48      eq makeOps(S ; Ss, M) = makeOpForSort(S, M) makeOps(Ss, M) .
49      eq makeOps(none, M) = none .
50
51      op makeOpForSort : Sort Module -> OpDecl .
52      eq makeOpForSort(S, M) = (op makeName(S) : nil -> getKind(M, S) [none] .) .
53
54      --- RULES
55
56      op makeRlsForOps : OpDeclSet Module -> RuleSet .
57      eq makeRlsForOps(Op Ops, M) = makeRlsForOp(Op, M) makeRlsForOps(Ops, M) .
58      eq makeRlsForOps(none, M) = none .
59
60      op makeRlsForOp : OpDecl Module -> RuleSet .
61      eq makeRlsForOp(Op, M) = makeRlsBuiltIn(Op, M) .
62
63      op makeRlsBuiltIn : OpDecl Module -> RuleSet .
64      eq makeRlsBuiltIn((op Q : Tys -> Ty [special(Hooks) Ats] .), M) = none .
65      eq makeRlsBuiltIn((op Q : Tys -> 'Universal [Ats] .), M) = none .
66      eq makeRlsBuiltIn((op Q : Tys -> 'Bool [Ats] .), M) = none .
67      eq makeRlsBuiltIn((op Q : Tys -> 'Nat [Ats] .), M) = none .
68      eq makeRlsBuiltIn(Op, M) = makeRlsTrivial(Op, M) [owise] .
69
70      op makeRlsTrivial : OpDecl Module -> RuleSet .
71      eq makeRlsTrivial((op Q : nil -> Ty [Ats] .), M) = makeConstantRl(Ty, Q, M) .
72      eq makeRlsTrivial(Op, M) = makeRlsH(Op, M) [owise] .
73
74      op makeRlsH : OpDecl Module -> RuleSet .
75
76      op makeRlsForSubsorts : SubsortDeclSet Module -> RuleSet .
77      eq makeRlsForSubsorts(SubS SubSs, M) = makeRlsForSubsort(SubS, M)
    makeRlsForSubsorts(SubSs, M) .
78      eq makeRlsForSubsorts(none, M) = none .
79
80      op makeRlsForSubsort : SubsortDecl Module -> RuleSet .
81      eq makeRlsForSubsort((subsort SC < SP .), M) =
82          makeRl(SP, getKind(M, SP), makeConstant(SC, makeName(SC))).
83
84      op makeConstantRl : Type Qid Module -> Rule .
85      eq makeConstantRl(Ty, Q, M) = makeRl(Ty, getKind(M, Ty), makeConstant(Ty, Q)) .
86
87      op makeRuleCallingRl : Type Qid NeTypeList Module -> Rule .
88      eq makeRuleCallingRl(Ty, Q, Tys, M) = makeRl(Ty, getKind(M, Ty), makeTerm(Q, Tys,
    M)).
89
90      op makeRl : Type Type Term -> Rule .
91      eq makeRl(Ty, Ty', T) =
92          (rl makeConstant(Ty', makeName(Ty)) => T [ none ] .) [owise] .
93
94      op makeTerm : Qid NeTypeList Module -> Term .
95      eq makeTerm(Q, Tys, M) = Q [ makeTermList(Tys, M) ] .
96
97      op makeTermList : NeTypeList Module -> NeTermList .
98      eq makeTermList(Ty Tys, M) = makeConstant(getKind(M, Ty), makeName(Ty)),
    makeTermList(Tys, M) .
```

```
    eq makeTermList(nil, M) = empty .

    op makeConstant : Type Qid -> Constant .
    eq makeConstant(Ty, Q) = qid(string(Q) + "." + string(Ty)) .

    --- UTILITY

    op makeName : Type -> Qid .
    eq makeName(Ty) = qid("generate" + string(Ty)) .

    op immSubsortsOf : Sort SubsortDeclSet -> SortSet .
    eq immSubsortsOf(S, (subsort SC < SP .) SubSs) =
        if S == SP then SC else none fi ;
        immSubsortsOf(S, SubSs) .
    eq immSubsortsOf(S, none) = none .

    op anyTypeMatches : Type TypeList -> Bool .
    eq anyTypeMatches(Ty, Ty' Tys) = Ty == Ty' or-else anyTypeMatches(Ty, Tys) .
    eq anyTypeMatches(Ty, nil) = false .
endfm

fmod MAUDE-CHECK-HEURISTICS is
    pr MAUDE-CHECK-BASE .

    var M : Module .
    var Q : Qid .
    var T : Term .
    vars Ty Ty' : Type .
    var S : Sort .
    var Ss : SortSet .
    var SubSs : SubsortDeclSet .
    vars Tys Tys' : TypeList .
    var Ats : AttrSet .

    --- set-like/list-like
    eq makeRlsH((op Q : Ty' Ty -> Ty [assoc id(T) Ats] .), M) =
        collectionLikeRls(immSubsortsOf(Ty', getSubsorts(M)), Q, Ty, M)
        makeRl(Ty, getKind(M, Ty), T) .

    op collectionLikeRls : SortSet Qid Type Module -> RuleSet .
    eq collectionLikeRls(S ; Ss, Q, Ty, M) =
        makeRuleCallingRl(Ty, Q, S Ty, M)
        collectionLikeRls(Ss, Q, Ty, M) .
    eq collectionLikeRls(none, Q, Ty, M) = none .

    --- generic
    ceq makeRlsH((op Q : Tys -> Ty [Ats] .), M) =
        makeRuleCallingRl(Ty, Q, Tys, M)
        if Ty' Tys' := Tys .

    eq makeRlsH((op Q : Tys -> Ty [Ats] .), M) = none [owise] .
endfm

fmod MAUDE-CHECK-DOWN is
    pr META-LEVEL .
    pr LEXICAL .

    var H OrigH : Header .
    var M : Module .
    var Is : ImportList .
    vars S S' : Sort .
    var Ss : SortSet .
    var SubSs : SubsortDeclSet .
    var Q : Qid .
```

```
163     var K : Kind .
164     var Ty : Type .
165     var Tys : TypeList .
166     var Ats : AttrSet .
167     var Ops : OpDeclSet .
168     var MAs : MembAxSet .
169     vars T T' : Term .
170     var EqC : EqCondition .
171     var Eqs : EquationSet .
172     var RlC : Condition .
173     var Rls : RuleSet .
174     var ME : ModuleExpression .
175     var Ch : Char .
176     var Str : String .
177
178     op toString : SModule Header -> String .
179     ceq toString(M, OrigH) =
180         fileHeader(OrigH)
181         + "mod " + string(H) + " is\n"
182         + toStringImports(M, Is)
183         + toStringSortSet(M, Ss)
184         + toStringSubsorts(M, SubSs)
185         + toStringOps(M, Ops)
186         + toStringMembers(M, MAs)
187         + toStringEqs(M, Eqs)
188         + toStringRls(M, Rls)
189         + "endm\n"
190     if (mod H is Is sorts Ss . SubSs Ops MAs Eqs Rls endm) := M .
191
192     op toString : FModule Header -> String .
193     ceq toString(M, OrigH) =
194         fileHeader(OrigH)
195         + "fmod " + string(H) + " is\n"
196         + toStringImports(M, Is)
197         + toStringSortSet(M, Ss)
198         + toStringSubsorts(M, SubSs)
199         + toStringOps(M, Ops)
200         + toStringMembers(M, MAs)
201         + toStringEqs(M, Eqs)
202         + "endfm\n"
203     if (fmod H is Is sorts Ss . SubSs Ops MAs Eqs endfm) := M .
204
205     op fileHeader : Header -> String .
206     eq fileHeader(OrigH) =
207           "---load " + toLower(string(OrigH)) + ".maude\n\n"
208         + "***(\n\tGenerated by MAUDE-CHECK -- Daniel Galán Pascual, 2020\n"
209         + "\tThis is a template generated automatically. Please complete to fit your
    needs.\n)***\n" .
210
211     op toStringImports : Module ImportList -> String .
212     eq toStringImports(M, (protecting ME .) Is) =
213         "\tprotecting " + toStringModuleExpr(M, ME) + " .\n" + toStringImports(M, Is) .
214     eq toStringImports(M, (extending ME .) Is) =
215         "\textending " + toStringModuleExpr(M, ME) + " .\n" + toStringImports(M, Is) .
216     eq toStringImports(M, (including ME .) Is) =
217         "\tincluding " + toStringModuleExpr(M, ME) + " .\n" + toStringImports(M, Is) .
218     eq toStringImports(M, (nil).ImportList) = "" .
219
220     op toStringModuleExpr : Module ModuleExpression -> String .
221     eq toStringModuleExpr(M, S) = string(S) .
222
223     op toStringSortSet : Module SortSet -> String .
224     eq toStringSortSet(M, S ; Ss) = "\tsort " + string(S) + " .\n" + toStringSortSet(M,
    Ss) .
```

```
225     eq toStringSortSet(M, (none).SortSet) = "" .
226
227     op toStringSubsorts : Module SubsortDeclSet -> String .
228     eq toStringSubsorts(M, (subsort S < S' .) SubSs) =
229         "\tsubsort " + string(S) + " < " + string(S) + " .\n" + toStringSubsorts(M,
    SubSs) .
230     eq toStringSubsorts(M, (none).SubsortDeclSet) = "" .
231
232     op toStringOps : Module OpDeclSet -> String .
233     eq toStringOps(M, (op Q : Tys -> Ty [Ats] .) Ops) =
234         "\top " + string(Q) + " : " + toStringTypes(M, Tys) + "-> " + toStringType(M,
    Ty) + toStringAttributes(M, Ats) + " .\n" + toStringOps(M, Ops) .
235     eq toStringOps(M, (none).OpDeclSet) = "" .
236
237     op toStringTypes : Module TypeList -> String .
238     eq toStringTypes(M, Ty Tys) = toStringType(M, Ty) + " " + toStringTypes(M, Tys) .
239     eq toStringTypes(M, (nil).TypeList) = "" .
240
241     op toStringType : Module Type -> String .
242     eq toStringType(M, S) = string(S) .
243     eq toStringType(M, K) = unescapeKindSymbols(string(K)) .
244
245     op unescapeKindSymbols : String -> String .
246     eq unescapeKindSymbols(Str) = "[" + substr(Str, 2, sd(length(Str), 4)) + "]" .
247
248     op toStringMembers : Module MembAxSet -> String .
249     eq toStringMembers(M, (none).MembAxSet) = "" .
250     --- TODO
251
252     op toStringEqs : Module EquationSet -> String .
253     eq toStringEqs(M, (eq T = T' [Ats] .) Eqs) =
254         "\teq " + toStringTerm(M, T) + " = " + toStringTerm(M, T) +
    toStringAttributes(M, Ats) + " .\n" + toStringEqs(M, Eqs) .
255     eq toStringEqs(M, (ceq T = T' if EqC [Ats] .) Eqs) =
256         "\tceq " + toStringTerm(M, T) + " = " + toStringTerm(M, T) + " if " +
    toStringEqCondition(M, EqC) + toStringAttributes(M, Ats) + " .\n" + toStringEqs(M, Eqs)
    .
257     eq toStringEqs(M, (none).EquationSet) = "" .
258
259     op toStringTerm : Module Term -> String .
260     eq toStringTerm(M, T) = printTokens(metaPrettyPrint(M, (none).VariableSet, T,
261                                         mixfix with-parens flat format number rat))
    .
262
263     op toStringAttributes : Module AttrSet -> String .
264     eq toStringAttributes(M, (none).AttrSet) = "" .
265     --- TODO
266
267     op toStringEqCondition : Module EqCondition -> String .
268     eq toStringEqCondition(M, (nil).EqCondition) = "" .
269     --- TODO
270
271     op toStringRls : Module RuleSet -> String .
272     eq toStringRls(M, (rl T => T' [Ats] .) Rls) =
273         "\trl " + toStringTerm(M, T) + " => " + toStringTerm(M, T') +
    toStringAttributes(M, Ats) + " .\n" + toStringRls(M, Rls) .
274     eq toStringRls(M, (crl T => T' if RlC [Ats] .) Rls) =
275         "\tcrl" + toStringTerm(M, T) + " => " + toStringTerm(M, T) + " if " +
    toStringRlCondition(M, RlC) + toStringAttributes(M, Ats) + " .\n" + toStringRls(M, Rls)
    .
276     eq toStringRls(M, (none).RuleSet) = "" .
277
278     op toStringRlCondition : Module Condition -> String .
279     eq toStringRlCondition(M, (nil).Condition) = "" .
```

```
    --- TODO

    op toLower : String -> String .
    eq toLower(Str) = toLower(Str, 0) .

    op toLower : String Nat -> String .
    ceq toLower(Str, Idx:Nat) = toLowerChar(substr(Str, Idx:Nat, 1)) + toLower(Str,
Idx:Nat + 1)
        if Idx:Nat < length(Str) .
    eq toLower(Str, Idx:Nat) = "" [owise] .

    op toLowerChar : Char -> Char .
    eq toLowerChar(Ch) = if ascii(Ch) >= 65 and ascii(Ch) <= 90 then
                            char(ascii(Ch) + 32)
                         else
                            Ch
                         fi .
endfm

mod MAUDE-CHECK-IO is
    pr MAUDE-CHECK-DOWN .
    pr STD-STREAM .

    op io : -> Oid .

    var S : String .

    op print : String -> Configuration .
    eq print(S) =
        <>
        write(stdout, io, S) .
endm

mod MAUDE-CHECK is
    pr MAUDE-CHECK-HEURISTICS .
    pr MAUDE-CHECK-IO .

    var H : Header .

    op generateMeta : Header -> Module .
    eq generateMeta(H) = makeMod(H) .

    op generate : Header -> Configuration .
    eq generate(H) = print(toString(makeMod(H), H)) .
endm
```

A.5 (generate-oven-template) Plantilla del generador de casos de prueba generada automáticamente

```
load oven.maude
---Load files containing the original definitions.

***(
  Generated by MAUDE-CHECK -- Daniel Galán Pascual, 2020
  This is a template generated automatically. Please complete to fit your needs.
)***
mod GENERATE-OVEN is
  protecting OVEN .
  protecting GENERATION-HELPERS .
  op generateBool : -> [Bool] .
```

```
12    op generateBool' : -> [Formula] .
13    op generateFormula : -> [Formula] .
14    op generateLabel : -> [LabelSet] .
15    op generateLabelSet : -> [LabelSet] .
16    op generateNat' : -> [State] .
17    op generateNzNat' : -> [State] .
18    op generateProposition : -> [Formula] .
19    op generateState : -> [State] .
20    op generateStatements : -> [Statements] .
21    op generateSystem : -> [System] .
22    op generateTransition : -> [TransitionSet] .
23    op generateTransitionSet : -> [TransitionSet] .
24    op generateZero' : -> [State] .
25    rl generateBool' => False .
26    rl generateBool' => True .
27    rl generateBool' => (generateNat' ==Nat generateNat') .
28    rl generateBool' => (generateBool' == generateBool') .
29    rl generateBool' => (generateLabel == generateLabel) .
30    rl generateBool' => (generateProposition == generateProposition) .
31    rl generateBool' => (generateState == generateState) .
32    rl generateBool' => (generateLabel in generateLabelSet) .
33    rl generateBool' => (generateLabel in generateSystem) .
34    rl generateBool' => (generateStatements |= generateFormula) .
35    rl generateBool' => allTransitions(generateSystem, generateState, generateFormula) .
36    rl generateBool' => allTransitionsHelper(generateTransitionSet, generateSystem,
      generateFormula) .
37    rl generateBool' => anyTransition(generateSystem, generateState, generateFormula) .
38    rl generateBool' => anyTransitionHelper(generateTransitionSet, generateSystem,
      generateFormula) .
39    rl generateFormula => generateBool' .
40    rl generateFormula => generateProposition .
41    rl generateFormula => (AF generateFormula) .
42    rl generateFormula => (AG generateFormula) .
43    rl generateFormula => (AX generateFormula) .
44    rl generateFormula => (A[generateFormula U generateFormula]) .
45    rl generateFormula => (EF generateFormula) .
46    rl generateFormula => (EG generateFormula) .
47    rl generateFormula => (EX generateFormula) .
48    rl generateFormula => (E[generateFormula U generateFormula]) .
49    rl generateFormula => (NotEG generateFormula) .
50    rl generateFormula => (Not generateFormula) .
51    rl generateFormula => (generateFormula And generateFormula) .
52    rl generateFormula => (generateFormula Implies generateFormula) .
53    rl generateFormula => (generateFormula Or generateFormula) .
54    rl generateFormula => (generateFormula Xor generateFormula) .
55    rl generateLabel => ([generateState : generateProposition]) .
56    rl generateLabelSet => (empty) .LabelSet .
57    rl generateLabelSet => generateLabel .
58    rl generateLabelSet => (generateLabel, generateLabelSet) .
59    rl generateNat' => generateNzNat' .
60    rl generateNat' => generateZero' .
61    rl generateNat' => (generateNat' + generateNat') .
62    rl generateNzNat' => (1) .NzNat' .
63    rl generateNzNat' => (10) .NzNat' .
64    rl generateNzNat' => (2) .NzNat' .
65    rl generateNzNat' => (3) .NzNat' .
66    rl generateNzNat' => (4) .NzNat' .
67    rl generateNzNat' => (5) .NzNat' .
68    rl generateNzNat' => (6) .NzNat' .
69    rl generateNzNat' => (7) .NzNat' .
70    rl generateNzNat' => (8) .NzNat' .
71    rl generateNzNat' => (9) .NzNat' .
72    rl generateNzNat' => (generateNzNat' + generateNat') .
73    rl generateNzNat' => s(generateNat') .
```

```
74    rl generateProposition => hot .
75    rl generateProposition => open .
76    rl generateProposition => working .
77    rl generateState => generateNat' .
78    rl generateStatements => (generateSystem, generateState) .
79    rl generateSystem => (< generateTransitionSet ; generateLabelSet >) .
80    rl generateTransition => (generateState -> generateState) .
81    rl generateTransitionSet => (empty) .TransitionSet .
82    rl generateTransitionSet => generateTransition .
83    rl generateTransitionSet => (generateTransition, generateTransitionSet) .
84    rl generateTransitionSet => if(generateBool', generateTransitionSet,
 generateTransitionSet) .
85    rl generateTransitionSet => transFrom(generateState, generateTransitionSet) .
86    rl generateZero' => (0) .Zero' .
87  endm
88
89  mod GENERATION-HELPERS is
90    including BOOL .
91    protecting RANDOM .
92    protecting COUNTER .
93    protecting CONVERSION .
94    protecting INT .
95    protecting STRING .
96    op generateBool : -> Bool .
97    op generateChar : -> Char .
98    op generateCharPrintable : -> Char .
99    op generateFloat : -> Float .
100   op generateFloatBetween : Float Float -> Float .
101   op generateInt : -> Int .
102   op generateIntBetween : Int Int -> Int .
103   op generateNat : -> Nat .
104   op generateNatBetween : Nat Nat -> Nat .
105   op generateRat : -> Rat .
106   op generateRatBetween : Rat Rat -> Rat .
107   op generateString : Nat -> String .
108   op generateStringPrintable : Nat -> String .
109   eq generateString(0) = "" .
110   eq generateString(N:NzNat) = (generateChar + generateString(sd(N:NzNat,1))) .
111   eq generateStringPrintable(0) = "" .
112   eq generateStringPrintable(N:NzNat) = (generateCharPrintable
 +generateStringPrintable(sd(N:NzNat, 1))) .
113   rl generateBool => false .
114   rl generateBool => true .
115   rl generateChar => char(generateNatBetween(0, 127)) .
116   rl generateCharPrintable => char(generateNatBetween(32, 126)) .
117   rl generateFloat => float((random(counter) / 4294967295)) .
118   rl generateInt => (- random(counter)) .
119   rl generateInt => random(counter) .
120   rl generateNat => random(counter) .
121   rl generateRat => rat(generateFloat) .
122   rl generateFloatBetween(Min:Float, Max:Float) => (Min:Float +(generateFloat
 *(Max:Float - Min:Float))) .
123   rl generateIntBetween(Min:Int, Max:Int) => (Min:Int +(generateInt rem((Max:Int + 1) -
 Min:Int))) .
124   rl generateNatBetween(Min:Nat, Max:Nat) => (Min:Nat +(generateNat rem((Max:Nat + 1) -
 Min:Nat))) .
125   rl generateRatBetween(Min:Rat, Max:Rat) => rat(generateFloatBetween(float(Min:Rat),
 float(Max:Rat))) .
126 endm
```

A.6 (generate-oven) Generador de casos de prueba adaptado manualmente

```
load oven.maude

mod GENERATE-OVEN is
        protecting OVEN .
        protecting RANDOM .
        protecting COUNTER .

        var States State : Nat .

        op generateBool' : -> [Formula] .
        op generateFormula : -> [Formula] .
        op generateLabel : Nat -> [LabelSet] .
        op generateLabelSet : Nat -> [LabelSet] .
        op generateProposition : -> [Formula] .
        op generateState : Nat -> [State] .
        op generateStatements : Nat -> [Statements] .
        op generateSystem : Nat -> [System] .
        op generateTransition : Nat Nat -> [TransitionSet] .
        op generateTransitionSet : Nat Nat -> [TransitionSet] .
        op generate : Nat -> [Bool'] .

        rl generateBool' => False .
        rl generateBool' => True .
        rl generateFormula => generateBool' .
        rl generateFormula => generateProposition .
        rl generateFormula => (AF generateFormula) .
        rl generateFormula => (AG generateFormula) .
        rl generateFormula => (AX generateFormula) .
        rl generateFormula => (A[generateFormula U generateFormula]) .
        rl generateFormula => (EF generateFormula) .
        rl generateFormula => (EG generateFormula) .
        rl generateFormula => (EX generateFormula) .
        rl generateFormula => (E[generateFormula U generateFormula]) .
        rl generateFormula => (Not generateFormula) .
        rl generateFormula => (generateFormula And generateFormula) .
        rl generateFormula => (generateFormula Implies generateFormula) .
        rl generateFormula => (generateFormula Or generateFormula) .
        rl generateFormula => (generateFormula Xor generateFormula) .
        rl generateLabel(States) => ([generateState(States) : generateProposition]) .
        rl generateLabelSet(States) => (empty) .LabelSet .
        rl generateLabelSet(States) => generateLabel(States), generateLabelSet(States) .
        rl generateProposition => hot .
        rl generateProposition => open .
        rl generateProposition => working .
        rl generateState(States) => toCustomNat(random(counter) rem States) .
        rl generateStatements(States) => (generateSystem(States), generateState(States))
.
        rl generateSystem(States) => (< generateTransitionSet(0, States) ;
generateLabelSet(States) >) .
        rl generateTransition(State, States) => (toCustomNat(State) ->
toCustomNat((State + State) rem States)) .
        rl generateTransitionSet(State, States) =>
            if State >= States then
                empty
            else
                generateTransition(State, States),
                if (State rem 2 =/= 0) then
                    toCustomNat(State quo 3) -> toCustomNat(State)
                else empty fi,
```

```
57                  generateTransitionSet(State + 1, States)
58              fi .
59
60          rl generate(States) => (generateStatements(States) |= generateFormula) .
61
62          op toCustomNat : Nat -> Nat' .
63          eq toCustomNat(s(N:Nat)) = ( s(toCustomNat(N:Nat)) ).Nat' .
64          eq toCustomNat(0) = (0).Nat' .
65  endm
```