Universidad Zaragoza
1542

Trabajo Fin de Máster

Model checking paramétrico de *workflows*

Autor/es

Eduardo González López de Murillas

Director/es

Francisco Javier Fabra Caro

Grupo de Integración de Sistemas Distribuidos y Heterogéneos (GIDHE)
Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura

2012

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Trabajo Fin de Máster

Máster en Ingeniería de Sistemas e Informática

Model Checking paramétrico de *workflows* científicos

Eduardo González López de Murillas

Director: Francisco Javier Fabra Caro

Grupo de Integración de Sistemas Distribuidos y Heterogéneos (GIDHE)
Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Curso 2011/2012
Junio 2012

Model Checking paramétrico de *workflows* científicos

Resumen

La computación científica ha ganado un creciente interés en los últimos años en áreas afines a las ciencias de la vida. Los *workflows* científicos son un tipo especial de *workflow* que se utilizan en escenarios de grandes dimensiones y gran complejidad computacional como modelos climáticos, estructuras biológicas, química, cirugía o simulación de desastres, por ejemplo, y cuya ejecución es un proceso que consume una gran cantidad de tiempo y recursos. Uno de los objetivos principales de la computación científica ha sido la mejora progresiva a través de la introducción de nuevos paradigmas y tecnologías para poder abordar desafíos cada vez más complejos, siendo uno de estos paradigmas la adición de aspectos semánticos a los *workflows*. Disponer de una serie de herramientas y técnicas que posibiliten el análisis del comportamiento del *workflow* antes de su ejecución resulta de gran interés. El objetivo de ese análisis es poder garantizar un comportamiento adecuado y correcto, así como verificar la correcta gestión y utilización de los recursos involucrados. El análisis debería permitir la predicción de la calidad de los resultados, así como identificar aquellos parámetros que son necesarios para obtener los resultados esperados. Desde el punto de vista del usuario, la incorporación de aspectos semánticos permite a los científicos realizar una navegación, interrogación, integración y composición de conjuntos de datos y servicios mucho más eficiente.

Sin embargo, el análisis del estado del arte en el área de la semántica aplicada a los modelos en la computación científica muestra carencias significativas en el grado de madurez y aplicación de este enfoque, así como la carencia de técnicas y herramientas para su aplicación. Es necesario, por tanto, proponer y desarrollar nuevas técnicas de modelado y análisis que puedan manejar dichos aspectos semánticos.

En este Trabajo Fin de Máster se aborda el análisis, diseño y desarrollo de un método y una herramienta de *model checking* basados en la introducción de aspectos y anotaciones semánticas tanto en los modelos como en las propiedades que deben verificarse. Como resultado, la herramienta COMBAS (*COmprobador de Modelos BAsado en Semántica*) proporciona un entorno de integración para la verificación de este tipo de modelos y la navegación por las estructuras resultantes del proceso. Para la descripción de los modelos de *workflows* científicos se ha utilizado una clase de Redes de Petri de alto nivel anotadas con información semántica en RDF, las U-RDF-PN. A lo largo de este trabajo se ha abordado la adición de las técnicas, metodologías y modelos necesarios para extender el *framework* con análisis *paramétrico*, que consiste en un análisis mucho más potente y expresivo mediante la utilización de parámetros cuyo valor es indeterminado al inicio del proceso, de forma que es posible estudiar el comportamiento del *workflow* respecto a los posibles valores de dichos parámetros. Para restringir los valores de los parámetros en cada uno de los caminos de ejecución del *workflow* se utiliza el concepto de guardas, expresadas en lógica proposicional, en el modelo del *workflow*. Para ello, es necesario estudiar primero qué herramientas permiten tratar dichas proposiciones, por lo que se analizan los Satisfiability Modulo Theories (SMTs), el estado actual de los estándares relacionados, la flexibilidad de los *solvers* disponibles y las herramientas que soporten la semántica que se va a aplicar.

Finalmente, la viabilidad y usabilidad del enfoque propuesto se ha demostrado mediante su aplicación al análisis del workflow EBI InterProScan, verificando propiedades de interés para el científico sin necesidad de implementar, desplegar ni ejecutar el *workflow*.

Índice

Capítulo 1 | **Introducción**

Los *workflows* científicos han ganado una relevancia notable en los últimos años, especialmente en áreas relacionadas con las ciencias de la vida. Estos *workflows*, que se han aplicado a tareas de cómputo en el mundo científico, representan un tipo especial de *workflow* por las particularidades del campo de aplicación. En general, engloban tareas de un alto coste computacional y tratan datos de gran tamaño, en aplicaciones tan diversas como el modelado climático, simulación de reacciones químicas o biológicas o cirugía, entre otros. Con el tiempo, se ha detectado la necesidad de desarrollo en este campo, y se han incorporado nuevas técnicas y paradigmas al mundo de los *workflows* científicos.

Es habitual encontrar *workflows* científicos compuestos por tareas muy costosas desde el punto de vista computacional. Su ejecución puede alargarse en el tiempo y requerir una gran cantidad de recursos. Es por esto, por lo que resulta de gran interés disponer de técnicas y herramientas que permitan analizar el comportamiento del *workflow* antes de su ejecución. El objetivo de dicho análisis es garantizar un funcionamiento adecuado del *workflow* y una óptima utilización de recursos. Además, la detección temprana de errores permite reducir costes en presupuesto y tiempo.

Uno de los enfoques principales que se ha abordado en la computación científica es la inclusión de técnicas y tecnologías semánticas a los modelos, y cómo esto permitirá tratar el problema planteado mediante técnicas de verificación. El uso de aspectos semánticos aporta grandes ventajas en cuanto al modelado, ya que permite una expresividad mucho mayor al poder especificar, no sólo propiedades estructurales, sino también información sobre los datos y el flujo de los mismos en el *workflow*.

El análisis del estado del arte muestra cómo en la actualidad existen trabajos que hacen uso de la semántica en los *workflows* con fines como la mejora en la descripción y descubrimiento de servicios, así como de recursos u otros *workflows*. Sin embargo, queda manifiesta una carencia muy significativa en cuanto al nivel de madurez y el desarrollo de técnicas y herramientas que integren el enfoque basado en la adición de aspectos semánticos, y que permitan su análisis posterior.

La introducción de aspectos semánticos en los *workflows* requiere nuevos modelos y técnicas de análisis que sean capaces de manejar este tipo de sistemas con información semántica. El formalismo de las Redes de Petri Unarias Anotadas con RDF (U-RDF-PN en adelante), es un subtipo de las redes de Petri que consideran este tipo de sistemas (las ventajas de utilizar Redes de Petri para modelar *workflows* han sido ampliamente tratadas en la literatura [29, 16]). En ellas, se usan anotaciones semánticas utilizando *Resource Description Framework*, RDF [18]. Las transiciones de la red de Petri corresponderían a acciones del sistema que cambien el estado del *workflow*. Dichas transiciones cuentan con anotaciones que representan la especificación formal de la tarea, incluyendo la descripción de las entradas y salidas. Algunas de las salidas podrían corresponder a datos generados u obtenidos en tiempo de ejecución, por tanto se hace necesario usar parámetros simbólicos para su representación. A su vez, pueden incluir referencias a precondiciones, que serán expresiones que especifican la tarea utilizando los parámetros de entrada. Para

tratar dichas precondiciones, se utilizan herramientas de *Satisfiability Modulo Theories* (SMT). Éstas nos permiten trabajar con predicados lógicos, de acuerdo a una combinación de teorías. Así, estas herramientas permiten saber si una colección de proposiciones lógicas es satisfacible o no. El análisis del *workflow* se lleva a cabo usando técnicas de *Model Checking* que definen el modo en que el modelo será verificado, a través de su grafo de alcanzabilidad, usando consultas expresadas en términos de la Computation Tree Logic (CTL).

1.1 Problema a resolver y objetivos

A pesar de la existencia de herramientas independientes, hasta ahora no se disponía de ningún entorno completo que integrase todas las fases del proceso de verificación, y mucho menos que facilitara la automatización de las mismas. Es esta necesidad la que se trata de solucionar con el objetivo principal de este Trabajo Fin de Máster, el desarrollo de *framework* COMBAS (*COmprobador de Modelos BAsado en Semántica*), un entorno completo de modelado y verificación que considera el formalismo de las U-RDF-PN con anotaciones en RDF, haciendo más sencillo el proceso y permitiendo al diseñador del sistema comprobar la validez de sus *workflows* gráficamente de forma automática y muy intuitiva.

En este trabajo se introduce una extensión significativa en las funcionalidades de dicho *framework*, en concreto la inclusión de análisis paramétrico, que consiste en un análisis mucho más potente y expresivo, utilizando parámetros de valor indeterminado al inicio del proceso. De esta forma, es posible estudiar el comportamiento del *workflow* respecto a los posibles valores de dichos parámetros.

Para restringir dichos valores en cada uno de los caminos de ejecución, se utiliza el concepto de *guardas*, expresadas en lógica proposicional en el modelo del *workflow*. Para realizar esto es necesario estudiar las herramientas que permiten tratar dichas proposiciones. Por tanto, se analizarán los Satisfiability Modulo Theories (SMTs), el estado actual de los estándares relacionados, la flexibilidad de los *solvers* disponibles, y las herramientas que soporten semántica que se van a aplicar.

Finalmente, la viabilidad del enfoque y herramienta propuestos se demostrará mediante su aplicación a un caso real en el ámbito de la computación científica, como es el *workflow* EBI InterProScan para el procesamiento de secuencias de proteínas.

1.2 Organización de la memoria

El resto de esta memoria está compuesta por varios capítulos y un anexo, y queda organizada de la siguiente forma:

- **Capítulo 2**: Se introducen los conceptos y estándares más importantes en los que se basa este trabajo.
- **Capítulo 3**: Se hace un análisis del estado del arte, se presentan las alternativas existentes en el campo de la verificación de *workflows* científicos y se identifican algunas de sus carencias y necesidades.
- **Capítulo 4**: Se expone el *framework* COMBAS, presentando su funcionamiento, diseño y algunos detalles de su implementación.
- **Capítulo 5**: Se expone un caso de aplicación real, analizando un *workflow* de procesado de proteínas, evidenciando la utilidad de COMBAS.

- **Capítulo 6**: Se presentan las conclusiones extraídas de este trabajo. A su vez, se indican las principales líneas de trabajo para continuar el desarrollo en el futuro.
- **Anexo A**: Se proporcionan datos adicionales acerca de las anotaciones semánticas, ficheros y datos que complementan la red de Petri del ejemplo mostrado en el Capítulo 5.

Capítulo 2 | **Puesta en contexto**

En este capítulo se presentan algunos de los conceptos en los que se basa este trabajo, y que se hace necesario conocer para comprender el fundamento del mismo. Se introduce el formalismo de las U-RDF-PN, se explica de modo escueto el fundamento de RDF, así como los SMT, se hablará brevemente de la lógica temporal y por último se presenta la técnica de *model checking*.

2.1 U-RDF-PNs

Las redes de Petri unarias con anotaciones en RDF (U-RDF-PN en adelante) representan un formalismo de modelado de sistemas anotados semánticamente. Éste puede ser utilizado tanto para modelar procesos de negocio, *workflows* científicos o cualquier otro tipo de sistema. Su principal ventaja es que consideran tanto el control como el flujo de datos, para lo cual se utilizan anotaciones semánticas en RDF.

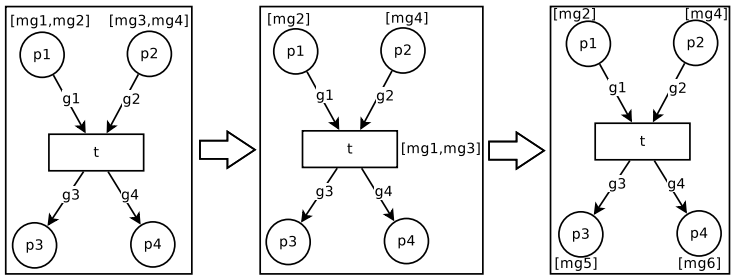

Figura 2.1: Ejemplo de *binding* en una U-RDF-PN.

Basadas en las *high level Petri nets*, constituyen una extensión en forma de anotaciones presentes en lugares, transiciones y arcos de las mismas. Las anotaciones en los lugares representan el marcado, de forma que substituimos los *tokens* de las redes de Petri por grafos RDF (RDFG). Estos grafos contienen una serie de tripletas RDF. A su vez, los arcos están anotados con patrones RDF (RDFGP), los cuales indican cómo deben ser los grafos RDF de los lugares de origen para que se pueda realizar el emparejamiento (*binding*).

En la Figura 2.1 se puede apreciar el proceso de binding en un caso sencillo. Vemos que la transición t tiene dos arcos de origen, que están anotados con los patrones RDF $g1$ y $g2$, y cuyos lugares de origen son $p1$ y $p2$ respectivamente. Dichos lugares están marcados con grafos RDF, $mg1$ y $mg2$ en $p1$ y $mg3$ y $mg4$ en $p2$. Por tanto, teniendo en cuenta que tenemos que coger un grafo RDF de cada lugar, existen cuatro combinaciones posibles a la hora de hacer binding: 1) $mg1$ y $mg3$; 2) $mg1$ y $mg4$; 3) $mg2$ y $mg3$ y 4) $mg2$ y $mg4$. Cualquiera de estas combinaciones podría darse, siempre y cuando satisfagan el patrón RDF correspondiente a su arco. Un patrón RDF es, al fin y al cabo, un grafo RDF, en el cual ciertas componentes de las tripletas pueden ser variables en lugar de URIs, literales o *bnodes*. Dichas variables son las que se emparejan con valores concretos de los grafos RDF. Así, si existe una variable común entre varios arcos de

entrada de una misma transición, para hacer *binding* dichas variables han de tomar el mismo valor. Volviendo al ejemplo, supongamos que tomamos la opción $mg1$ y $mg3$, y que el *binding* es posible. Entonces se procede al disparo de la transición t, que a su vez dispone de dos arcos de salida, con los patrones $g3$ y $g4$ que van a los lugares $p3$ y $p4$ respectivamente. Las variables de dichos patrones serán substituidas por los valores tomados en el *binding* de los arcos de entrada, y se colocará el grafo resultante en el lugar de salida. Así, vemos que se generan dos nuevos grafos, $mg5$ y $mg6$, que satisfacen los patrones $g3$ y $g4$ respectivamente.

Las transiciones de la red de Petri se corresponden con acciones del sistema modelado que cambian el estado del *workflow*. La información semántica adjunta a dichas transiciones corresponde a la especificación formal de la tarea, incluyendo la descripción de los parámetros de entrada y salida. Dado que algunas de las salidas pueden corresponder a datos recibidos o computados en tiempo de ejecución, se requiere el uso de parámetros simbólicos para representarlos. Por otro lado, hay que considerar las precondiciones de las tareas, que se describirán como expresiones que incluyen los parámetros presentes en la especificación de dicha tarea. Así, encontramos las precondiciones definidas en las transiciones como condiciones necesarias, previas al disparo de una transición. El tratamiento de estas expresiones requiere el uso de los *SMT solvers* que se presentarán más adelante.

Este formalismo es el utilizado en este trabajo para modelar los *workflows* que se van a analizar, sobre los que se aplicarán las técnicas de *model checking* adaptadas. Para más información sobre el formalismo, consultar [21], donde se presentan las U-RDF-PN con un mayor nivel de detalle. En nuestro caso, sólo consideraremos modelos acíclicos, es decir, modelos cuya estructura sea de tipo DAG (Directed acyclic graph). Esto es así porque de esta forma evitamos la posibilidad de que el grafo de alcanzabilidad sea infinito. Aunque lo pueda parecer, esta restricción no representa una desventaja notable, ya que, por norma general, en nuestro dominio de aplicación, *workflows* científicos, es habitual que los *workflows* sean acíclicos. Esta consideración ya ha sido tenida en cuenta en otras aproximaciones al manejo de *workflows*, como DAGMan [13], un *meta-scheduler* para Condor que maneja dependencias entre tareas dentro de un *workflow*.

2.2 *Resource Description Framework* (RDF)

El Resource Description Framework (RDF) es un modelo de metadatos, el cual forma parte de la familia de especificaciones del World Wide Web Consortium (W3C). Se ha convertido en un método de descripción de conceptos o modelado de información muy utilizado en recursos Web, usando una gran variedad de formatos y sintaxis. RDF está basado en la idea de identificar cosas usando identificadores Web (Uniform Resource Identifiers, o URIs), y describir recursos en términos de simples tripletas (conocidas como tripletas RDF) las cuales contienen sujetos, propiedades y valores para esas propiedades. El sujeto es el recurso, aquello que es descrito. El predicado representa la propiedad o relación que se quiere expresar del sujeto. En último lugar, el objeto es el valor de la propiedad que se describe del sujeto o recurso. Es posible combinar RDF con otras tecnologías tales como RDFS y OWL para añadir expresividad.

RDF Schema (RDFS) es un lenguaje basado en RDF que proporciona elementos para construir ontologías, permitiendo estructurar los recursos descritos en RDF. Es muy extensible, y está formado por clases que facilitan la clasificación de recursos o la creación de relaciones entre ellos.

Otro lenguaje de ontologías es el Web Ontology Language (OWL), el cual proporciona una mayor expresividad que RDFS gracias a la diversidad de directivas y clases que introduce. Existen diversas variantes de OWL, que incluyen subconjuntos del mismo, tales como OWL Lite, OWL DL o OWL Full. En [4] se puede encontrar más información sobre estos lenguajes de descripción de ontologías.

2.3 SMT

Los *Satisfiability Modulo Theories solvers* (SMT *solvers*) son herramientas que permiten trabajar con predicados lógicos, y resolver problemas de decisión usando una serie de teorías expresadas en lógica de primer orden con igualdad. La lógica de primer orden se distingue de la lógica proposicional en que la primera hace uso de variables cuantificables. Así, algunos ejemplos de teorías usadas con frecuencia son la teoría de números reales, la de enteros, o incluso de otras estructuras de datos como listas, vectores, vectores de *bits*, etc.

En realidad, los SMT representan generalizaciones del problema de satisfacibilidad booleana (*Boolean SAT*). Sin embargo, permiten una expresividad mucho mayor que la disponible con fórmulas booleanas. Esto es así porque es posible tratar problemas a un nivel más alto, por ejemplo en el conjunto de los enteros, en lugar de hacerlo a nivel de *bits* con variables booleanas.

En el ámbito que presentamos, se hace necesario el uso de los *SMT solvers* a la hora de procesar las pre y postcondiciones presentes en el modelo.

2.4 Lógica temporal

La lógica temporal presenta proposiciones lógicas cuyo valor es condicionado por el tiempo. A diferencia de la lógica clásica, en la cual las proposiciones se evalúan a un valor estático, una expresión en lógica temporal podrá cambiar de valor dependiendo del momento en el que se evalúe. El uso de este tipo de lógica proporciona una expresividad mucho mayor, y ofrece una herramienta extremadamente útil a la hora de interrogar los modelos de nuestro sistema. Se distinguen, principalmente, dos corrientes dentro de la lógica temporal: Linear Temporal Logic (LTL) y Computation Tree Logic (CTL). En el caso que nos atañe, nos centraremos en la lógica CTL, ya que es la que mejor se adapta al tipo de consultas que queremos expresar. Computation Tree Logic (CTL) es un tipo de lógica temporal que considera la existencias de varias líneas temporales paralelas, que representan posibles evoluciones futuras, ramificaciones. Esto encaja con la idea de grafo de alcanzabilidad, que presenta ramificaciones según elecciones que se tomarán en el sistema. Es esta lógica la que nos permitirá expresar fórmulas que usaremos para consultar el comportamiento del sistema.

2.5 *Model checking*

La comprobación de modelos, o *model checking*, es una técnica de verificación formal usada para verificar la corrección de sistemas, especialmente software o hardware. La validación supone una parte muy importante dentro de la fase de diseño. Con el fin de realizar dicha validación, existen diversas técnicas, entre las que destacan las de simulación, testeo, o verificación formal. Las técnicas de simulación y testeo exploran algunos de los posibles comportamientos del sistema estudiado, en ciertos entornos. Sin embargo, dejan abierta opción a la existencia de fallos en casos no explorados. Las técnicas de verificación formal, por el contrario, realizan una comprobación exhaustiva del conjunto de escenarios posibles, partiendo de una especificación dada. Así, cuando un diseño es correcto según una técnica de verificación formal, queda garantizado que el diseño está libre de fallos. Como es lógico, la detección de fallos queda supeditada a la corrección de la especificación proporcionada.

Se denominan verificaciones formales a los métodos basados en lógica para comprobar la corrección de un sistema. A su vez existen diversas técnicas dentro de las verificaciones formales. Nos centraremos en la técnica de *model checking*. La idea principal del *model checking* consiste en describir un sistema y sus estados por medio de una especificación formal, y aplicar un

verificador lógico automático que verifique su corrección. Para esto, debemos contar con tres elementos básicos:

- Especificación formal: modelo M de una determinada teoría lógica.
- Propiedad a verificar: Fórmula ϕ de un lenguaje lógico.
- Método de verificación: *model checking*, que comprobará si un modelo M satisface la propiedad ϕ. $M| = \phi$

En el trabajo que se presenta, se utiliza la técnica de *model checking* sobre fórmulas en lógica temporal. Por lo tanto, el modelo M se corresponde con un sistema de transiciones (un grafo de alcanzabilidad que habrá sido generado a partir de una U-RDF-PN), la fórmula será enunciada en un lenguaje de lógica temporal (CTL en nuestro caso, extendido con semántica y proposiciones lógicas de primer orden), y la técnica de verificación comprobará si el modelo verifica dicha fórmula.

Capítulo 3 | **Estado del arte**

La revisión del trabajo relacionado para abordar los objetivos de este Trabajo Fin de Máster se centra en tres áreas bien diferenciadas. En primer lugar, se analizan los principales trabajos y herramientas que consideran la adición de aspectos semánticos a los *workflows* científicos. En segundo lugar, se abordarán los principales estándares y enfoques para realizar *model checking*. La adición de anotaciones paramétricos sobre el modelo inicial de las U-RDF-PN implica la utilización de SMTs, por lo que, finalmente, se estudian los diferentes mecanismos para atacar este problema.

3.1 Semántica en *Workflows* científicos

En el Capítulo 1, se ha hablado de la importancia que han cobrado los *workflows* científicos en la actualidad. Esto ha motivado el desarrollo en este área, introduciendo nuevas técnicas y paradigmas, tratando de mejorar el comportamiento de dichos *workflows*.

En este trabajo, tratamos de resaltar la importancia de realizar tareas de verificación sobre este tipo de *workflows*, dada su gran complejidad computacional y coste en tiempo y recursos. Estas características proporcionan motivos más que suficientes para invertir esfuerzo en asegurar su corrección, de forma previa a la ejecución de los experimentos, con el fin de ahorrar costes en presupuesto y tiempo que podrían ocasionar errores de diseño de los *workflows*.

Como se ha mencionado, diversas técnicas han sido aplicadas a este campo. Entre ellas, destaca la inclusión de aspectos semánticos en la descripción de los *workflows*. Esto ha aportado numerosas ventajas, como mejoras en la descripción y descubrimiento de servidos, recursos o *workflows*, o el análisis de datos de *provenance*.

Una de las posibles aplicaciones de la utilización de aspectos semánticos es el modelado de *workflows* científicos. [5] fue una de las primeras propuestas en el ámbito de la investigación, que combina metadatos y servicios *Web* ofreciendo soporte para *workflows* científicos y presenta cómo los descriptores semánticos son incorporados a la tecnología de servicios.[7] analiza las ventajas de usar aspectos semánticos en *workflows* científicos, y pone de manifiesto su utilidad a la hora de descubrir y ejecutar *workflows*, la reutilización de los mismos, y cómo esta información semántica puede ser utilizada como datos de *provenance*. Vemos que estas propuestas introducen técnicas de tecnología de servicios al ámbito de los *workflows* científicos.

En [28] se incluye información de calidad en *workflows* científicos, utilizando una aproximación basada en anotaciones RDF de ontologías OWL DL. [24] propone soporte de *reasoning* en *workflows* científicos, y utiliza dicha técnica en descubrimiento de recursos. Destaca el planteamiento del proyecto myExperiment [14]. Éste, presenta una red social basada en tecnologías de la *Web* semántica que facilita la búsqueda y compartición de *workflows* científicos de gran utilidad en investigación. Como parte de este proyecto, [27] propone una ontología OWL DL que permite que los datos sean publicados en un formato RDF estándar mejorando su descubrimiento.

Existe una gran iniciativa, dentro del análisis de *workflows* científicos, centrada en el estudio de datos de *provenance*. Dichos datos son de una gran relevancia en el mundo científico,

permitiendo la reproducción de experimentos, detección de comportamientos anómalos y proporcionando información de gran utilidad a la hora de verificar la autenticidad de los datos. En este campo se ha realizado un gran esfuerzo en la captura y la compartición de los datos, que son obtenidos como resultados de la ejecución de los experimentos. Existen trabajos que tratan de mejorar el uso y análisis de dichos datos, como [30], que introduce el uso de base de datos relacionales para su análisis, o [11] que presenta un nuevo lenguaje de consulta para expresar preguntas sobre dichos datos.

Cabe destacar la reciente aparición de una propuesta de entorno de verificación llamado CosyVerif [1]. Esta iniciativa, propuesta por el grupo MeFoSyLoMa [2], consiste en un entorno que trata de integrar diversos formalismos de especificación y verificación de sistemas dinámicos, además de herramientas de visualización que faciliten su uso. A día de hoy, soportan autómatas y Redes de Petri como formalismos de modelado, e integran técnicas de análisis estructural, por medio de cálculo de invariantes, así como análisis de comportamiento, usando técnicas de construcción de grafos de alcanzabilidad simbólicos, *unfolding* o simulaciones estocásticas, entre otras. Aunque dicho entorno se encuentra en sus fases iniciales, y pretende ser extendido con más formalismos y técnicas, no ofrece soluciones inmediatas a las carencias detectadas anteriormente, como verificación en *workflows* científicos y modelado de flujo de datos.

Por tanto, vemos que no hay estudios que traten de analizar el comportamiento de *workflows* antes de conocer los resultados de su ejecución y recursos requeridos. En [21] se presenta el formalismo de las Redes de Petri Unarias Anotadas (U-RDF-PNs) y la técnica de *Model Checking* RDF, que son aplicados a procesos de negocio. Es este el formalismo en el que nos basamos para la realización de este trabajo, aplicándolo a los *workflows* científicos y mostrando su validez en este campo.

3.2 Estándares y herramientas para el *Model checking*

En la Tabla 3.1 se presentan las principales (por su utilización en la literatura y su grado de madurez) herramientas de *Model Checking*, que pueden clasificarse según varios criterios. En la segunda columna se presenta qué técnica o técnicas de *Model Checking* utiliza cada herramienta para realizar la verificación. La tercera columna lista los lenguajes o formalismos de modelado que soportan, siendo el lenguaje utilizado para especificar el sistema que se verificará. La cuarta columna indica qué lenguajes se pueden usar para especificar propiedades a verificar sobre el modelo. La quinta columna muestra el lenguaje o lenguajes de programación utilizados en la implementación de la herramienta. Por último, se muestran las plataformas sobre las que se puede ejecutar.

Una característica importante de los trabajos presentados en la tabla es que nos hemos concentrado en aquellas alternativas más representativas que soporten CTL como lenguaje de propiedades. Esto es así ya que CTL representa el lenguaje de lógica temporal que mejor se ajusta a la naturaleza de los modelos que utilizamos. CTL permite expresar consultas que aprovechan la ramificación del modelo, en nuestro caso una estructura de Kripke anotada.

Si revisamos la tabla vemos que ninguna de las alternativas admite como lenguaje de modelado las Redes de Petri, excepto la herramienta CosyVerif, de la que ya se ha hablado en la sección 3.1. Esto es un dato muy significativo, ya que la utilización de cualquiera de las herramientas para analizar modelos expresados en términos de las U-RDF-PN requeriría un trabajo

[1] Entorno de verificación software CosyVerif. Disponible en http://www.cosyverif.org/

[2] Grupo MeFoSyLoMa, asociación formada por los laboratorios Cedric (Cnam), IBISC (Univ. Evry), LACL (Univ. Paris 12), LIP6 (UPMC), LIPN (Univ. Paris 13), LRDE (Epita), LSV (École Normale Supérieure de Cachan) y LTCI (TELECOM ParisTech). Más información disponible en http://www.mefosyloma.fr/

Nombre	**Técnica de *Model Checking***	**Lenguaje de modelado**	**Lenguaje de propiedades**	**Lenguaje de programación usado**	**Plataforma**
APMC [19]	Aproximado y probabilístico	Reactive Modules	PCTL, PLTL	C	Unix
ARC [1]	Plano	AltaRica	μ-calculus, CTL*	ANSI C	Unix
BANDERA [10]	Análisis de código	Java	CTL, LTL	Java	Windows y Unix
CADENCE SMV [26]	Plano	Cadence SMV, SMV, Verilog	CTL, LTL	?	Windows y Unix
CWB-NC [9]	Plano y temporizado	CCS, CSP, LOTOS, TCCS	AFMC, CTL, GCTL	SML of New Jersey	Windows y Unix
GEAR [3]	Plano	?	AFMC, CTL, μ-calculus	Java	Windows y Unix
MCMAS [25]	Plano y epistémico	ISPL	CTL, CTLK	C++	Windows, Unix, MacOS
MRMC [22]	Tiempo real y probabilístico	Plain MC	CSL, CSRL, PCTL, PRCTL	C	Windows, Linux, MacOS
NuSMV [8]	Plano	SMV	CTL, LTL, PSL	C	Unix, Windows, MacOSX
PRISM [20]	Probabilístico	PEPA, PRISM language, Plain MC	CSL, PLTL, PCTL	C++, Java	Windows, Linux, MacOS
TAPAs [6]	Plano	CCSP	CTL, μ-calculus	Java	Windows, Unix, MacOS
CosyVerif [2]	Plano, simbólico y estocástico	Petri Nets, autómatas	?	Java	Windows, Unix, MacOS
COMBAS [15]	Semántico y paramétrico	U-RDF-PN	CTL	Java	Windows, Unix, MacOS

Tabla 3.1: Relación de herramientas de *Model Checking* con soporte CTL.

de adaptación complejo que considere las limitaciones y la expresividad del modelo de entrada del sistema destino.

Además, ninguna de las principales herramientas contempla la introducción de semántica, por lo que la expresividad dentro del modelo está limitada. Resulta evidente que estas alternativas no son válidas para analizar los modelos de U-RDF-PN, tan siquiera mediante la adaptación de éstos a los modelos de entrada de las herramientas por la carencia de soporte para la semántica en los mismos. Esto refuerza el objetivo inicial de proporcionar una herramienta que acepte modelos más expresivos reforzados con anotaciones semánticas, así como un lenguaje de consulta que aproveche dicha expresividad. Como se ha descrito anteriormente, en este trabajo se propone definirlos como redes de petri anotadas semánticamente (U-RDF-PN), con descripción del flujo de datos y dependencias, y el lenguaje CTL extendido con semántica.

3.3 Estándares y herramientas de SMT

Respecto al uso de *SMT solvers*, existe una amplia variedad de herramientas entre las que elegir. Entre las características más representativas están: la colección de teorías y lenguajes soportados, el lenguaje de programación en que están implementados y su portabilidad y reusabilidad. Sin embargo, hay otros aspectos a tener en cuenta, como la actividad de la comunidad de usuarios, la frecuencia con que se publican nuevas versiones, y la calidad de la documentación. De acuerdo con estos conceptos, la lista de opciones se reduce considerablemente. CVC, OpenSMT y STP representan los proyectos más activos. STP sólo soporta fórmulas en la teoría de *bit-vectors* y *arrays*, por lo tanto no supone una solución válida para nuestro problema, ya que la teoría básica que necesitamos es la de aritmética lineal, en concreto sobre los reales, enteros y booleanos. A pesar de que OpenSMT es una opción válida, elegimos CVC porque es una alternativa más estable y apta para entornos de producción. En concreto, elegimos CVC3, que es una versión estable, a diferencia de CVC4 que se encuentra en fase beta.

En cuanto a la descripción de predicados para realizar el proceso de *model checking*, cada año, desde el 2005, se celebra la competición Satisfiability Modulo Theories Competition (SMT-COMP), con el propósito de impulsar el desarrollo de los *SMT solvers* y promover la adopción de la librería estándar de SMT (SMT-LIB [3]). SMT-LIB es un formato diseñado por la comunidad que trata de unificar la descripción de teorías y las entradas/salidas de los *solvers*, así como proporcionar una colección de *benchmarks* para fomentar el desarrollo de este tipo de herramientas. Por tanto, comparado con otros formatos como CVC o DIMACS, SMT-LIB representa la opción más recomendable para mantener la compatibilidad y portabilidad de nuestros predicados.

[3]Disponible en http://www.smtlib.org/

Capítulo 4 | COMBAS: un Comprobador de Modelos Basado en Semántica

El objetivo de este trabajo es presentar el *framework* COMBAS, el cual permite cubrir algunas de las carencias en las herramientas y técnicas existentes en el campo de la verificación de *workflows* científicos, identificadas en la sección 3.

En las siguientes secciones se irán desgranando las características del *framework*, así como el modo de trabajo que su uso plantea. Se abordará el proceso de verificación desde el punto de vista del diseñador de sistemas, y más adelante nos adentraremos en el diseño de COMBAS. Una de las contribuciones de este trabajo es la incorporación al *framework* de soporte para tratar modelos y consultas paramétricas, y es explicada en mayor detalle en la sección 4.2. Concluiremos el capítulo mostrando los detalles de su implementación.

4.1 Proceso de verificación

Desde el punto de vista del ingeniero o diseñador de sistemas (e.g. un científico o un analista de procesos de negocio), existen una serie de pasos a seguir en el proceso de verificación que la herramienta propone. Estos pasos, que se corresponden con los componentes y estructuras de datos descritos en la Figura 4.1, se enumeran a continuación:

1. En primer lugar, el ingeniero debe diseñar el *wofkflow* del proceso usando una Red de Petri. Para dicho fin, puede usar herramientas gráficas como Renew [23]. Tanto el marcado inicial del modelo, como el propio modelo deben ser anotados semánticamente. Un asistente gráfico será de ayuda en este proceso.

2. A continuación, se llevan a cabo una serie de procesos transparentes para el usuario, quien puede revisar los resultados de cada fase. El generador de grafos de alcanzabilidad toma los ficheros del paso anterior y genera dos salidas: el grafo de alcanzabilidad (RG), y un conjunto de ficheros que contienen la relación entre los estados y sus marcados, así como los RDFGs (grafos RDF) correspondientes a cada marcado. También se obtiene la representación gráfica del grafo de alcanzabilidad. Todos estos resultados son visibles por medio de un visor Web, incluido en la aplicación *Web interface* de COMBAS.

3. Ahora, es necesario crear la fórmula CTL que se verificará con el *Model checker*. El diseñador puede construir dicha fórmula usando el interfaz de usuario proporcionado por el editor web.

4. Finalmente, el *model checker* usa la fórmula CTL, junto con el RG para computar y generar la salida. Dicha salida consiste en una colección de ficheros que representan y relacionan los

estados del RG validados con los fragmentos correspondientes de la fórmula CTL. Tanto los ficheros de entrada como los de salida pueden ser visualizados haciendo uso del interfaz Web de forma sencilla e intuitiva.

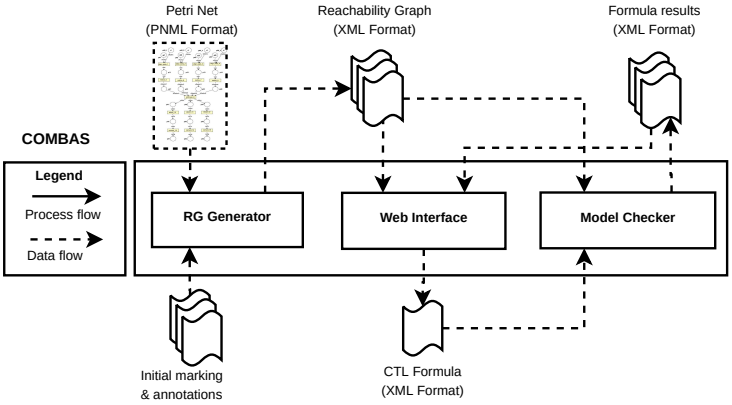

Figura 4.1: Arquitectura simplificada del *framework* COMBAS.

4.2 Diseño

La Figura 4.2 muestra la arquitectura del *framework*, el cual integra un conjunto de herramientas y técnicas que cubren el ciclo completo de verificación y análisis de *workflows* anotados semánticamente: desde la generación de modelos U-RDF-PN, el grafo de alcanzabilidad correspondiente y su estructura de Kripke, la creación y edición de consultas y formulas CTL, la ejecución del proceso de *model checking* y, por último, la visualización y revisión de resultados.

Todos los componentes en COMBAS ofrecen un interfaz sencillo y flexible, y se ocultan al usuario las tareas complejas como la generación del RG, el almacenamiento en la base de datos semántica y el proceso de verificación. Durante el proceso de *model checking*, se utiliza una base de datos RDF (almacén de tripletas). El *framework* COMBAS permite utilizar diversas bases de datos RDF, como *AllegroGraph RDFStore*, o *Virtuoso RDF Store* por ejemplo. El único requisito que la base de datos debe cumplir es que sea accesible por medio de un interfaz SPARQL. En este trabajo, se ha utilizado *Virtuoso RDF Store*. Tras el procesamiento, se obtiene el resultado de la verificación de la fórmula. Además, es posible visualizar los estados de RG utilizando un visor basado en un interfaz *Web*. De este modo, es posible encontrar situaciones específicas en las que un predicado viola alguna situación deseada, obteniendo una mejor comprensión del comportamiento del *workflow* y facilitando la mejora del mismo.

4.2.1 Generador del grafo de alcanzabilidad (*RG Generator*)

El proceso de generación del grafo de alcanzabilidad requiere de un modelo válido como entrada, en nuestro caso una red de Petri de tipo U-RDF-PN, con un marcado inicial que se corresponda con el estado inicial del sistema [21]. Las anotaciones semánticas pueden aparecer asociadas a tres elementos diferentes de la red:

- Arcos: son anotados con patrones RDF. Si el arco es de entrada (de lugar a transición), el patrón RDF representa la entrada (*input*) de la transición correspondiente, así que dicho patrón debe ser satisfecho por alguno de los grafos RDF que marcan en lugar de origen de dicho arco para que la transición se sensibilice. Si el arco va de la transición a un lugar, el patrón RDF representa la salida (*output*), y por tanto cómo será el grafo RDF que

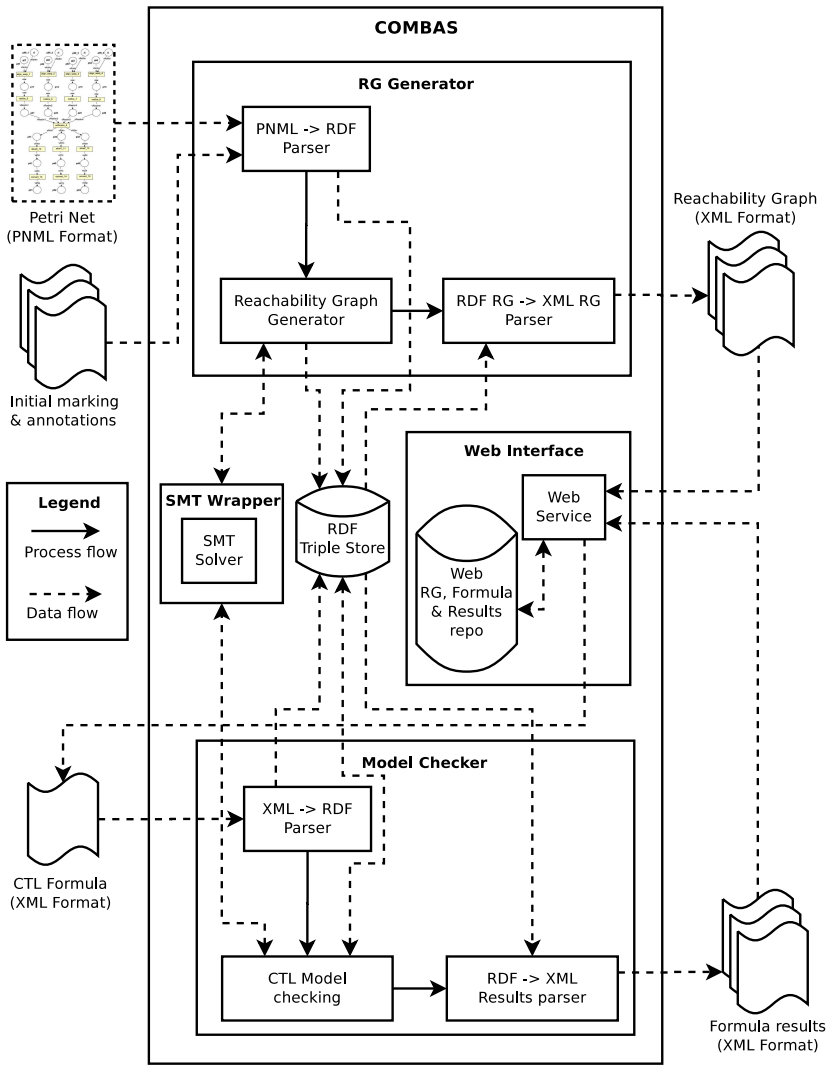

Figura 4.2: Arquitectura del *framework* de *model checking* COMBAS.

marcará el lugar de destino. En este último caso, las variables del patrón serán substituidas por valores concretos, obtenidos tanto del *binding* de las entradas como generados por la transición.

- Transiciones: contienen anotaciones, que llamaremos guardas, que representan la precondición. Si la transición está sensibilizada, se evalúa la guarda y en caso de que la guarda sea cierta se procede al disparo.

- Lugares: pueden contener grafos RDF, que representan el marcado. Un lugar puede tener una marca, representada por un grafo RDF, puede tener varias, que corresponderá con varios grafos RDF, o puede no estar marcado, con lo cual no habrá ningún grafo RDF asociado. Cada grafo contendrá tripletas RDF, en las que, además, pueden aparecer *blank nodes* (*bnodes*) que representan parámetros.

La generación del grafo de alcanzabilidad está basada en el algoritmo clásico usado en computación de grafos de alcanzabilidad en redes de Petri, tras hacer las modificaciones necesarias para adaptarlo a la naturaleza semántica de las anotaciones [21]. El grafo de alcanzabilidad generado se almacena como tripletas RDF, de acuerdo con la ontología descrita en la Figura 4.3.

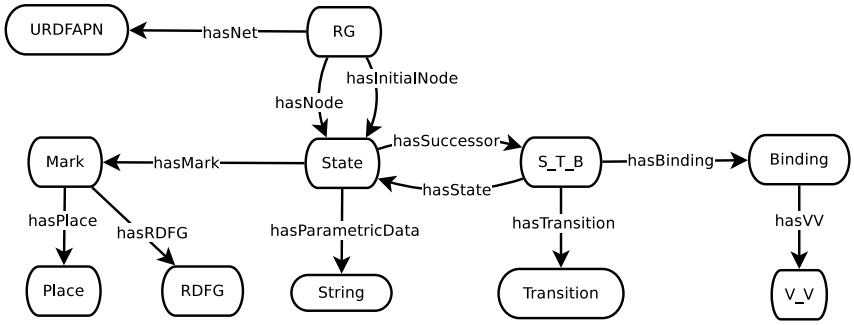

Figura 4.3: Ontología del grafo de alcanzabilidad paramétrico.

En este trabajo, se considera la extensión paramétrica sobre las U-RDF-PN definidas en [21]. El concepto es muy simple: añadir parámetros al modelo, en lugar de valores estáticos. De este modo, es posible representar casos más generales de un proceso o un *workflow* añadiendo proposiciones lógicas a las anotaciones, que luego etiquetarán cada estado en el grafo de alcanzabilidad. Las guardas de las transiciones también deben ser definidas haciendo uso de dichas anotaciones.

A continuación se describirán las principales diferencias entre el enfoque paramétrico y el planteamiento de *model checking* ordinario presentado en [21] respecto a la generación del grafo de alcanzabilidad. En primer lugar, se tomará como entrada una red U-RDF-PN paramétrica, que es una red U-RDF-PN ordinaria anotada con declaraciones paramétricas en algunas de sus transiciones (como guardas), lugares (como marcado inicial) y arcos (como parte de los patrones RDFG). Se hizo necesario desarrollar un *wrapper* que permitiera utilizar un *solver* SMT, herramienta crucial para trabajar con lógica y parámetros. De esta forma, el proceso de generación del grafo de alcanzabilidad es similar al usado para redes U-RDF-PN ordinarias, excepto por la siguientes consideraciones:

1. El estado inicial no está formado únicamente por el marcado inicial de cada lugar de la red, sino también por el marcado inicial paramétrico de cada lugar. En el caso de que no exista tal marcado, la declaración lógica *true* será considerada como marcado inicial paramétrico.

2. Para cada transición con una guarda paramétrica, se debe comprobar la validez de dicha guarda. Esto es posible utilizando un SMT *solver* que comprobará si la conjunción de las declaraciones lógicas en la guarda y las del estado actual es satisfacible.

3. Cuando se genera un nuevo estado en el RG, su marcado paramétrico está formado por la conjunción del marcado paramétrico del estado padre y la guarda de la transición disparada.

4. Cuando se inserta un nuevo estado en el almacén, es necesario comprobar que es único. Para ello, el generador debe comparar la parte semántica, así como la parte paramétrica. Cuando compara esta última, se usa el SMT *solver* para comprobar la equivalencia de ambas declaraciones lógicas (P y Q), observando que la siguiente fórmula es satisfacible $P \rightarrow Q \wedge Q \rightarrow P$.

4.2.2 Consideraciones de la extensión paramétrica

La incorporación del análisis paramétrico supone diversos cambios en la forma de especificar el modelo de entrada y en la expresión de las fórmulas a verificar. En cuanto al modelo, los parámetros son usados en ciertos elementos de la red U-RDF-PN:

- Lugares: los parámetros aparecen en los grafos RDF de marcado, expresados como *blank nodes* (*bnodes*). No representan un valor concreto, sólo un identificador de parámetro.

- Arcos: contienen patrones RDF. En dichos patrones no se hace referencia explícita a los parámetros, sin embargo, las variables que componen estos patrones pueden tomar un parámetro como valor en el emparejamiento (*binding*) antes de disparar una transición.

- Transiciones: las denominadas guardas son proposiciones lógicas en lenguaje *SMT-Lib*, que representan la precondición de la transición. Estas proposiciones contienen referencias a las variables que forman parte del *binding* o emparejamiento que se produce al asociar grafos RDF a patrones RDF presentes en los arcos de entrada de la transición. Si una guarda hace referencia a alguna variable de los patrones presentes en los arcos de entrada, se substituye por el valor que ha tomado en el *binding* y se procede a su validación usando el *wrapper* SMT. Si la proposición se evalúa a cierta, la transición es disparable.

Si atendemos a la expresión de fórmulas, la principal diferencia radica en la adición del elemento literal $SMT-EXP$ que contiene la proposición paramétrica a verificar. Más información sobre las fórmulas se detalla en la sección 4.2.4.

4.2.3 Interfaz *Web* (*Web Interface*)

Las funcionalidades básicas del interfaz *Web* son: 1) visualización del grafo de alcanzabilidad; 2) creación y edición de fórmulas CTL que usará el *model checker*; y 3) revisión de los resultados del proceso de *model checking*. El objetivo inicial fue integrar todas estas funcionalidades en una única interfaz. A continuación se mostrarán sus componentes con más detalle.

4.2.3.1 Visualización del RG

Como se ha mencionado anteriormente, la salida del generador de RG consiste en un fichero XML principal que describe la estructura del grafo, y una colección de ficheros XML que contienen los grafos RDF que marcan cada estado del RG. En el caso de que se trate de un sistema paramétrico, también se generarán varios ficheros *SMT-Lib*. Estos ficheros contienen los predicados paramétricos correspondientes a cada estado. El grafo de alcanzabilidad computado contendrá una gran cantidad de estados, alrededor de cientos si es un modelo simple, pero se incrementará exponencialmente conforme la complejidad crezca. Por lo tanto, la revisión del grafo puede convertirse en una tarea tediosa o prácticamente imposible. Esta fue una de las principales razones para desarrollar un visor, que represente el grafo como un diagrama, de forma que sea más sencillo revisar el marcado de cada estado.

La aplicación desarrollada hace posible seleccionar un RG determinado, y visualizar su estructura de forma gráfica, permitiendo consultar el marcado de forma sencilla e intuitiva.

4.2.3.2 Edición de fórmulas

Dado el propósito de este trabajo, es importante proporcionar herramientas efectivas para la edición de fórmulas CTL. Representa un beneficio evidente para el proceso de verificación, evitando errores de escritura, y mejorando la experiencia de usuario, ya que reduce la curva de aprendizaje desde el punto de vista del usuario, e incrementa la rapidez de creación y edición de fórmulas. Es por esto por lo que es crucial diseñar una herramienta tan intuitiva como sea posible. Parecía lógico desarrollar un constructor de fórmulas que proporcionara una lista de componentes y un mecanismo para incluirlas dentro de la fórmula de forma interactiva. También, se pensó que sería interesante incluir una fase de verificación para asegurar que la entrada está bien construida, evitando errores sintácticos.

Entre las funcionalidades de esta parte, destaca la habilidad de crear y editar fórmulas, visualizarlas de forma gráfica y la creación interactiva usando interfaces de usuario intuitivos.

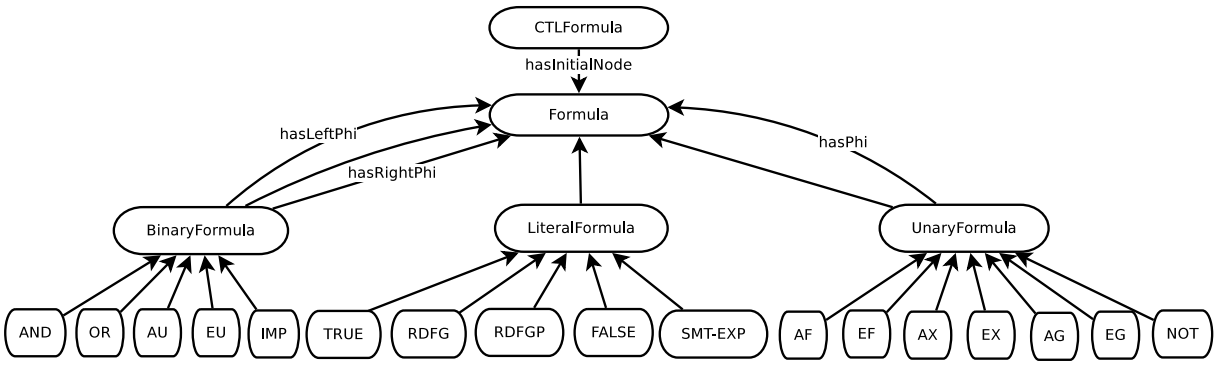

Figura 4.4: Ontología de la fórmula CTL con elementos paramétricos.

4.2.3.3 Revisión de resultados del *model checker*

Un paso muy importante dentro del proceso de *model checking* es la revisión de resultados. En esta fase, el usuario ha de verificar los resultados y analizarlos con el fin de identificar los errores, si existen, del modelo. Por lo tanto es crucial proveer herramientas para tratar la salida de nuestro *model checker*.

El interfaz desarrollado permite seleccionar un RG e inspeccionar el resultado de la ejecución de una determinada fórmula, usando la visualización del RG, y comprobar el marcado y la satisfacibilidad de cada componente de la fórmula en los estados del RG.

4.2.4 Comprobador de modelos (*Model Checker*)

La ontología representada en la Figura 4.4 se usa para especificar la formula CTL de entrada para el *model checker*. Puede contener cualquiera de los operadores CTL unarios AF, EF, AX, EX, AG, EG y NOT, o binarios AND, OR, AU, EU y IMP, o cualquiera de los nodos terminales $TRUE$, $FALSE$, $RDFG$, $RDFGP$ y $SMT-EXP$, siendo $RDFG$ un grafo RDF, $RDFGP$ a un patrón RDF, y $SMT-EXP$ una declaración lógica en formato *SMT-Lib*.

La otra entrada de nuestro *model checker* es el grafo de alcanzabilidad, cuyo proceso de generación ha sido explicado anteriormente. Debido a la posibilidad de que dicho grafo contenga información paramétrica, necesitamos disponer de un *SMT solver*. Por lo tanto, se requiere el mismo *wrapper* mencionado en 4.2.1.

El proceso de verificación es similar al usado con las U-RDF-PN ordinarias y fórmulas CTL tal y como se describe en [21]. La principal diferencia radica en el nodo terminal $SMT-EXP$, el cual representa una declaración lógica en formato *SMT-Lib*. Con el fin de saber si un estado satisface una declaración, es necesario usar el *wrapper*. Por lo tanto, el marcado paramétrico de dicho estado debe ser comparado a la declaración de la fórmula. Siendo P la parte paramétrica del estado, y Q el contenido del nodo $SMT-EXP$ de la fórmula, se usa la expresión lógica $P \vee Q$ para comprobar la validez del marcado paramétrico del estado (P) y la expresión de la fórmula (Q). De esta forma sabremos si ambas declaraciones lógicas son compatibles o contradictorias.

4.3 Descripción de algoritmos

Anteriormente se ha descrito el diseño de los componentes que conforman el *framework* COMBAS. Sin embargo, los detalles sobre el funcionamiento de ciertos algoritmos ha sido omitido por motivos de brevedad. A continuación se presentan, con mayor nivel de detalle, los principales algoritmos del sistema, como el de generación del grafo de alcanzabilidad, o el de *model checking*.

El interés de presentar dichos algoritmos es presentar las modificaciones introducidas en este trabajo para adaptar su funcionamiento al enfoque propuesto.

4.3.1 Algoritmo de generación del RG

El algoritmo utilizado para la generación del grafo de alcanzabilidad está basado en el algoritmo clásico de generación para redes de Petri. Sin embargo, presenta numerosas modificaciones dada la naturaleza semántica de la variante que usamos, así como la extensión paramétrica. A continuación se tratará de definir el algoritmo adaptado.

En primer lugar, como elementos y estructuras de datos utilizadas, encontramos:

- Pila de *No revisados*: pila de marcados del RG (m_i) de los que aún se deben obtener los estados sucesores.
- Lista de estados del RG: conjunto de estados de RG que ya han sido generados. Esta lista representa la base de datos.
- Red de Petri del modelo inicial: modelo, del que se obtendrá el RG, que contiene la estructura de la red además de las anotaciones semánticas y paramétricas.
- Tabla *hash* de estados del RG: tabla *hash* que, por medio de una clave asociada a cierta característica de cada estado, permite comprobar la existencia previa de forma rápida. La clave utilizada es un *hash* de una cadena que especifica el número de marcas que el estado posee para cada lugar de la red, siempre en un orden prefijado. Aunque la clave no es única por estado, sí que divide el conjunto de ellos en pequeñas listas, que quedan almacenadas en la citada tabla *hash* asociadas a su código. De este modo es posible saber si un estado no existe previamente (si su *hash* no existe en la tabla), o en caso de que exista, ha de ser comparado con un número de estados mucho menor que el total de los generados hasta el momento.
- Tabla *hash* de marcado paramétrico por estados: tabla que asocia a cada estado generado del RG, su marcado paramétrico.
- Tabla *hash* de guardas en transiciones: esta tabla utiliza como clave el nombre de la transición, y asocia a cada una de ellas la guarda paramétrica expresada en el modelo. Aunque esta información ya está definida de forma implícita en el modelo almacenado en la base de datos, mantener esta estructura en memoria mejora el rendimiento y evita acceder repetidamente a los mismos datos.

Una vez vistas las estructuras de datos, se enumeran los pasos, a grandes rasgos, que el algoritmo sigue en el proceso de generación:

1. Procesamiento del fichero PNML de la red de Petri anotada para introducirla en el Triple Store como RDF, así como la información asociada a las anotaciones.
2. Inicialización del SMT *solver* con los parámetros definidos en la red.
3. Introducción del marcado inicial m_0 en la pila de *No revisados*, en la base de datos y en las tablas hash de estados y de marcados paramétricos.
4. Bucle principal: mientras (*No revisados* != vacía) {
   (a) Desapilado de la cima de *No revisados* (m_i)
   (b) Obtención de las transiciones del modelo sensibilizadas por el marcado m_i

(c) Cálculo de posibles *bindings* para cada transición, y verificación de guardas (usando el SMT) para cada *binding*.

(d) Generación de los marcados resultantes de las transiciones disparadas

(e) Bucle: para cada m_j generado en el paso anterior {

i. Comprobación de la existencia de un estado equivalente a m_j {

A. Si existe, no se hace nada.

B. Sino, se añade a la base de datos, se apila en *No revisados*, se añade a la tabla *hash* de estados y a la de marcados paramétricos.

ii. Enlace del estado m_i con el estado m_j como sucesor.

(f) Si no se generó ningún marcado, enlazar el estado m_i con sí mismo, para convertirlo en un estado de *deadlock* o final.

5. Generación del XML de descripción del RG a partir de la representación RDF.

Tras estos pasos, el grafo de alcanzabilidad correspondiente se encontrará almacenado en la base de datos, así como en ficheros XML devueltos como resultado del proceso.

Como anotación, cabe destacar que, gracias a las optimizaciones introducidas añadiendo tablas *hash* y división del espacio de estados, es posible reducir el tiempo de generación de forma significativa. Esto queda demostrado aplicando el algoritmo al ejemplo presentado anteriormente en [15], en el que, para un caso complejo del *workflow First Provenance Challenge*, en el que se generan 11664 estados, el tiempo de cómputo original era de 44 horas y 23 minutos, y se vio reducido, ejecutándose sobre la misma plataforma y configuración, a 12 horas y 14 minutos.

4.3.2 Algoritmo de *model checking*

El *model checker* desarrollado basa su técnica en el algoritmo de etiquetado propuesto en [12], pero adaptado a las U-RDF-PN, tal y como se presenta en [21]. Sin embargo, ha sido necesario introducir modificaciones para soportar los aspectos paramétricos con los que se extienden las U-RDF-PN en este trabajo. Ya se ha explicado anteriormente cómo se trata este aspecto, sin embargo no se detalla cómo está estructurado el algoritmo.

Podemos establecer una serie de elementos y estructuras de datos utilizados por el algoritmo:

- Lista de estados del RG: representa la base de datos, en la que se almacena el RG, sus anotaciones semánticas y paramétricas.
- Fórmula RDF-CTL: representa la fórmula a analizar, almacenada en la base de datos subdividida en subfórmulas anidadas.

El algoritmo consta de los siguientes pasos que determinan el proceso de *model checking* planteado:

1. Carga de la fórmula, en formato XML, a la base de datos, transformándola en RDF.
2. Carga del modelo, en formato XML, a la base de datos en formato RDF. En el caso de que el modelo ya esté almacenado, se especifica el identificador del mismo dentro del almacén de tripletas.
3. Inicialización del SMT con los parámetros definidos en la red asociada al modelo.
4. Se selecciona el nodo raíz de la fórmula, f_0 y el nodo raíz del modelo (estado inicial), m_0.

5. Se llama a la función $SAT(f_0, m_0)$, la cual devolverá cierto o falso, dependiendo de si el estado inicial satisface dicha fórmula.

6. Generación de la colección de ficheros XML que indican qué partes de la fórmula se satisfacen en cada estado del RG. Esto conforma el contraejemplo, en caso de no validarse, o el ejemplo, en caso de que el modelo valide la fórmula.

A su vez, vemos que se hace una llamada al algoritmo SAT. Esta función toma dos parámetros: 1) f_i, que representa el nodo de la fórmula a verificar; 2) m_j, estado del RG sobre el que verificar la subfórmula f_i. Su funcionamiento se presenta a continuación:

1. Se verifica si f_i ya ha sido comprobado para m_j y, si es así, se devuelve el valor correspondiente.
2. Sino, identificamos el tipo de subfórmula {
   (a) f_i == OR:
      i. Si $SAT(f_i.left(), m_j) : setChecked(f_i, m_j, true)$ y devolver $true$.
      ii. Sino, si $SAT(f_i.right(), m_i) : setChecked(f_i, m_j, true)$ y devolver $true$.
      iii. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.
   (b) f_i == AND:
      i. Si $SAT(f_i.left(), m_j)$ {
         A. Si $SAT(f_i.right(), m_j) : setChecked(f_i, m_j, true)$ y devolver $true$.
         B. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.
      ii. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.
   (c) f_i == NEG:
      i. Si $SAT(f_i.phi(), m_j) : setChecked(f_i, m_j, false)$ y devolver $false$.
      ii. Sino: $setChecked(f_i, m_j, true)$ y devolver $true$.
   (d) f_i == TRUE:
      i. Devolver $true$.
   (e) f_i == FALSE:
      i. Devolver $false$.
   (f) f_i == AF:
      i. Si $SAT(f_i.phi(), m_j) : setChecked(f_i, m_j, true)$ y devolver $true$.
      ii. Sino, si $isChecking(f_i, m_j) : setChecked(f_i, m_j, false)$ y devolver $false$.
      iii. Sino {
         A. $setChecking(f_i, m_j)$
         B. Si $f_i.hasSuccessors()$: Para cada sucesor m_{jk} {
            a. Si $SAT(f_i, m_{jk})$: siguiente.
            b. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.
         C. Si para todo m_{jk} fue cierto: $setChecked(f_i, m_j, true)$ y devolver $true$.
         D. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.
   (g) f_i == AG:
      i. Si $SAT(f_i.phi(), m_j)$ {
         A. Si $isChecking(f_i, m_j) : setChecked(f_i, m_j, true)$ y devolver $true$.
         B. Sino {
            a. $setChecking(f_i, m_j)$
            b. Para cada m_{jk} sucesor de m_j {
               i. Si $SAT(f_i, m_{jk})$: siguiente.
               ii. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.

c. Si para todo m_{jk} fue cierto: $setChecked(f_i, m_j, true)$ y devolver $true$.

ii. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.

(h) $f_i ==$ EF:

i. Si $SAT(f_i.phi(), m_j)$: $setChecked(f_i, m_j, true)$ y devolver $true$.

ii. Sino, si $isChecking(f_i, m_j)$: $setChecked(f_i, m_j, false)$ y devolver $false$.

iii. Sino {

A. $setChecking(f_i, m_j)$

B. Para cada m_{jk} sucesor de m_j {

a. Si $SAT(f_i, m_{jk})$: $setChecked(f_i, m_j, true)$ y devolver $true$.

b. Sino: siguiente.

C. Si para todo m_{jk} fue falso: $setChecked(f_i, m_j, false)$ y devolver $false$.

(i) $f_i ==$ EG:

i. Si $SAT(f_i.phi(), m_j)$ {

A. Si $isChecking(f_i, m_j)$: $setChecked(f_i, m_j, true)$ y devolver $true$.

B. Sino {

a. $setChecking(f_i, m_j)$

b. Para cada m_{jk} sucesor de m_j {

i. Si $SAT(f_i, m_{jk})$: $setChecked(f_i, m_j, true)$ y devolver $true$.

ii. Sino: siguiente.

c. Si para todo m_{jk} fue falso: $setChecked(f_i, m_j, false)$ y devolver $false$.

ii. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.

(j) $f_i ==$ AX:

i. Si $m_j.hasSuccessors()$ {

A. Para cada m_{jk} sucesor de m_j {

a. Si $SAT(f_i.phi(), m_{jk})$: siguiente.

b. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.

B. Si para todo m_{jk} fue cierto: $setChecked(f_i, m_j, true)$ y devolver $true$.

ii. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.

(k) $f_i ==$ EX:

i. Si $m_j.hasSuccessors()$ {

A. Para cada m_{jk} sucesor de m_j {

a. Si $SAT(f_i.phi(), m_{jk})$: $setChecked(f_i, m_j, true)$ y devolver $true$.

b. Sino: siguiente.

B. Si para todo m_{jk} fue falso: $setChecked(f_i, m_j, false)$ y devolver $false$.

ii. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.

(l) $f_i ==$ AU:

i. Si $isChecking(f_i, m_j)$: $setChecked(f_i, m_j, false)$ y devolver $false$.

ii. Sino {

A. $setChecking(f_i, m_j)$

B. Si $SAT(f_i.left(), m_{jk})$ {

a. Para todo m_{jk} sucesor de m_j {

i. Si $SAT(f_i.right(), m_{jk})$: siguiente.

ii. Sino, si $SAT(f_i, m_{jk})$: siguiente.

iii. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.

b. Si para todo m_{jk} fue cierto: $setChecked(f_i, m_j, true)$ y devolver $true$.

C. Sino: $setCheched(f_i, m_j, false)$ y devolver $false$.

(m) $f_i ==$ EU:

i. Si $isChecking(f_i, m_j)$: $setChecked(f_i, m_j, false)$ y devolver $false$.
ii. Sino {
A. $setChecking(f_i, m_j)$
B. Si $SAT(f_i.left(), m_{jk})$ {
a. Para todo m_{jk} sucesor de m_j {
i. Si $SAT(f_i.right(), m_{jk})$: $setChecked(f_i, m_j, true)$ y devolver $true$.
ii. Sino, si $SAT(f_i, m_{jk})$: $setChecked(f_i, m_j, true)$ y devolver $true$.
iii. Sino: siguiente.
b. Si para todo m_{jk} fue falso: $setChecked(f_i, m_j, false)$ y devolver $false$.
C. Sino: $setCheched(f_i, m_j, false)$ y devolver $false$.

(n) $f_i ==$ IMP:
i. Si $SAT(f_i.left(), m_j)$ {
A. Si $SAT(f_i.right(), m_j)$: $setChecked(f_i, m_j, true)$ y devolver $true$.
B. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.
ii. Sino: $setChecked(f_i, m_j, true)$ y devolver $true$.

(o) $f_i ==$ RDFG:
i. Si $SAT\_RDFG(f_i, m_j)$: $setChecked(f_i, m_j, true)$ y devolver $true$.
ii. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.

(p) $f_i ==$ RDFGP:
i. Si $SAT\_RDFGP(f_i, m_j)$: $setChecked(f_i, m_j, true)$ y devolver $true$.
ii. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.

(q) $f_i ==$ PARAMETRIC:
i. Si $SAT\_PARAMETRIC(f_i, m_j)$: $setChecked(f_i, m_j, true)$ y devolver $true$.
ii. Sino: $setChecked(f_i, m_j, false)$ y devolver $false$.

En el pseudocódigo presentado, aparecen llamadas a varias funciones que no habían sido explicadas anteriormente. A continuación se comenta el significado de cada una de ellas:

- $setCheched(f_i, m_j, boolean)$: establece que el estado m_j ha sido comprobado usando la fórmula f_i, y el resultado es $true$ o $false$, dependiendo de la constante booleana que se pase como argumento.
- $setChecking(f_i, m_j)$: establece que el estado m_j está actualmente siendo comprobado con la fórmula f_i.
- $isChecking(f_i, m_j)$: devuelve el valor establecido por la función anterior.
- $SAT\_RDFG(f_i, m_j)$: verifica si el estado m_j verifica el grafo RDF declarado en f_i.
- $SAT\_RDFGP(f_i, m_j)$: verifica si el estado m_j verifica el patrón RDF declarado en f_i.
- $SAT\_PARAMETRIC(f_i, m_j)$: verifica si el estado m_j verifica la declaración paramétrica especificada en f_i, usando el SMT *solver* para comprobar si la fórmula $P \wedge Q$ es cierta, siendo P el marcado paramétrico del estado m_j, y Q la declaración paramétrica definida en f_i.

4.4 Implementación

La implementación del *framework* supone la integración de tecnologías muy diversas, tanto para el procesado de la información semántica, su almacenamiento, el interfaz *Web*, el procesado de declaraciones lógicas, y otros aspectos de la herramienta. A continuación se detallan, de forma separada para cada uno de los componentes del *framework*, qué tecnologías han sido utilizadas en la implementación.

4.4.1 Generador de RGs

Este componente del *framework* ha sido desarrollado en el lenguaje Java [1], concretamente usando la versión 1.6 de la JVM de Oracle. Además, como entorno de programación se ha recurrido a Eclipse [2], el cual tiene soporte multilenguaje y es muy extensible por medio de *plugins*.

Una de las librerías más utilizadas es JDOM [3]. Ésta, permite tratar y generar, con un API accesible desde Java, documentos XML [4], lenguaje con el que se codifica la información de intercambio generada en cada paso de la herramienta.

En el generador de RGs se hace necesario utilizar una base de datos semántica, o *Triple Store*. Como requisito, debe ofrecer un interfaz de consulta utilizando el lenguaje SPARQL, al menos la versión 1.1. Esto se debe a que, para hacer consultas complejas, se requiere de anidación de consultas, entre otras características, lo cual está disponible sólo a partir de la versión 1.1 del estándar SPARQL. En este aspecto, la base de datos híbrida Virtuoso [5] representa una alternativa adecuada, dado su buen rendimiento, soporte para SPARQL 1.1 y API basada en Sesame [6]. Éste último es un framework open-source para consultar y analizar datos RDF [7]. Ofrece una interfaz (apilable) a través de la cual se pueden añadir funcionalidades, y se abstrae el motor de almacenamiento del motor de consultas. Existen multitud de *Triple-stores* que pueden usarse por medio de la API de Sesame.

El formato PNML [8], que también es utilizado en el generador de RGs, es una sintaxis basada en XML para describir *high-level Petri nets*. Se requiere para importar las redes anotadas, desde las que se generará el grafo de alcanzabilidad.

Cuando se genera el grafo de alcanzabilidad, el lenguaje DOT [9] es utilizado para describir la estructura de dicho grafo. A partir de esta descripción, se generará un diagrama del mismo haciendo uso del paquete de herramientas Graphviz [10]. La librería *Apache log4j* [11] también es utilizada en la herramienta para funciones de *logging*.

Una parte importante en la extensión paramétrica de COMBAS, es el procesado de los predicados lógicos. Para esta tarea, se decidió utilizar alguno de los SMT *solvers* existentes. La herramienta seleccionada, por motivos de especificaciones, rendimiento y soporte, fue CVC3 [12]. Aunque la aplicación dispone de un API, ésta presenta limitaciones como la falta de soporte del lenguaje SMT-Lib. Es por esto por lo que se decidió implementar un *wrapper* para utilizar el binario de *cvc3*. Este *wrapper* carga en memoria la utilidad, la inicializa y la mantiene en ejecución para evitar la sobrecarga de instanciar el ejecutable varias veces en cada iteración. De esta forma se mejora el rendimiento.

4.4.2 *Model checker*

Este componente del *framework* se parece bastante, en cuanto a tecnologías utilizadas, al generador de RGs. Ha sido desarrollado en Java y, al igual que el generador, hace uso de la base de datos Virtuoso para almacenar las tripletas. Utiliza las librerías JDOM, para la interpretación y generación de XML, y *Apache log4j*, para el registro de *logs*. Adicionalmente, requiere del *wrapper* desarrollado para la herramienta SMT, ya que las fórmulas CTL pueden contener expresiones paramétricas en formato SMT-Lib.

[1] http://www.oracle.com/technetwork/java/index.html
[2] http://www.eclipse.org
[3] http://www.jdom.org
[4] http://www.w3.org/TR/REC-xml
[5] http://virtuoso.openlinksw.com
[6] http://www.openrdf.org
[7] http://www.w3.org/RDF
[8] http://www.pnml.org
[9] http://www.graphviz.org/Documentation.php
[10] http://www.graphviz.org
[11] http://logging.apache.org/log4j
[12] http://www.cs.nyu.edu/acsys/cvc3

4.4.3 Visor *Web*

Para la implementación del visor *Web*, se ha utilizado Google-Web-Toolkit (GWT) [13], un entorno de desarrollo Java para construir aplicaciones AJAX manteniendo la compatibilidad entre navegadores. Y para incluir nuevos elementos en el interfaz (*widgets*) se ha recurrido a la librería SmartGWT [14]. Otra librería usada en este componente es GWT-dnd [15], que proporciona funcionalidades de *Drag-and-Drop* para el entorno GWT, a la cual se ha recurrido para la implementación del editor de fórmulas. En la parte servidor del visor *Web*, también se usó el paquete Commons-FileUpload [16], que proporciona funciones para procesar ficheros subidos desde el navegador *Web* de forma robusta. Además, se hizo uso del formato JSON [17], que permite intercambiar datos de forma ligera. Es un subconjunto de la notación literal de objetos *JavaScript* y que no requiere el uso de XML. Para procesar este formato, la librería JSON.simple [18] fue de gran utilidad.

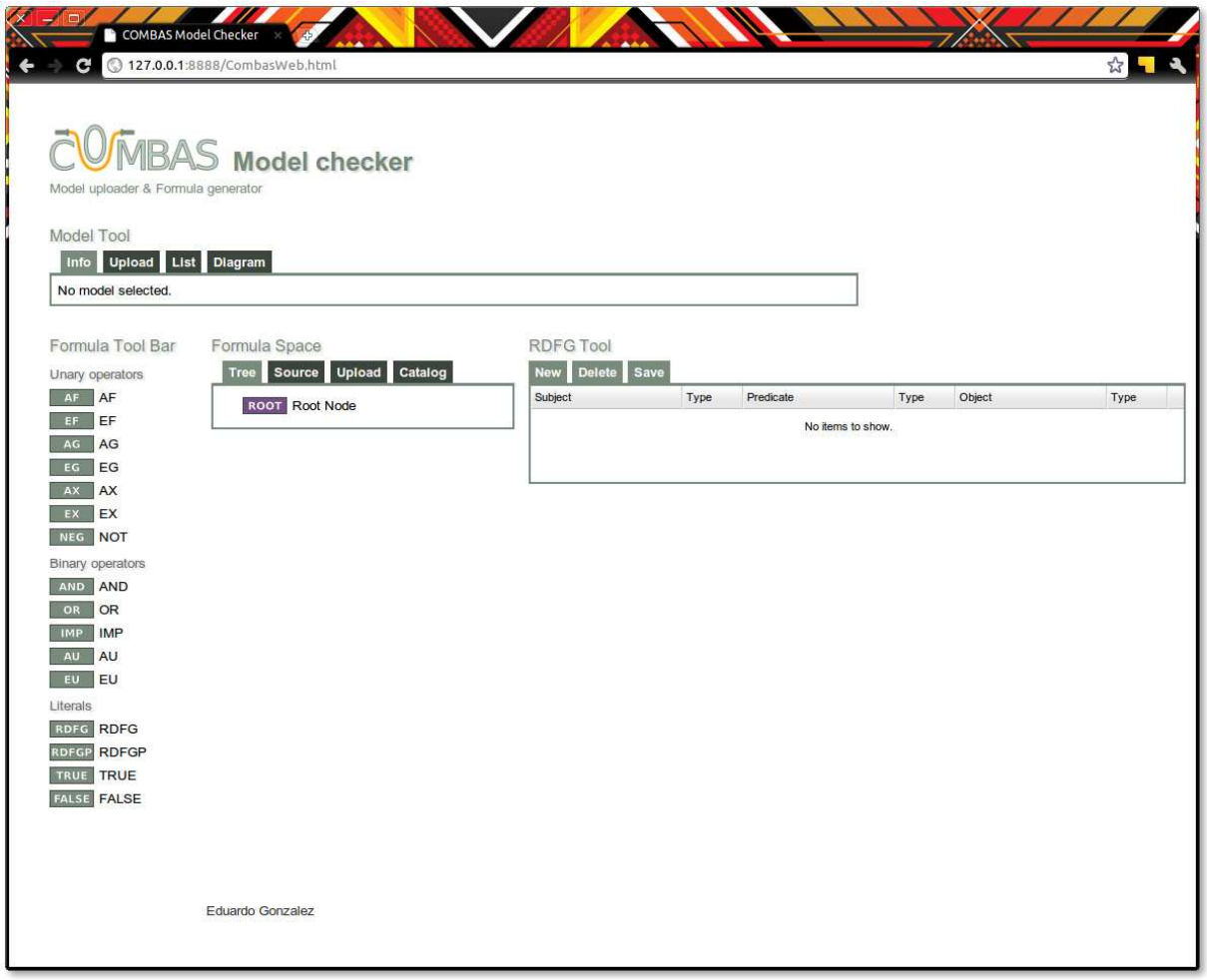

Figura 4.5: Vista general de la aplicación *Web* de COMBAS.

En la Figura 4.5 se muestra el aspecto general del interfaz *Web*. Vemos que se integran paneles para trabajar con cada uno de los componentes: fórmulas, grafos RDF, y grafos de alcanzabilidad. La Figura 4.6 muestra una captura del proceso de revisión de resultados. En ella se puede ver un grafo de

[13] http://code.google.com/webtoolkit
[14] http://code.google.com/p/smartgwt
[15] http://code.google.com/p/gwt-dnd
[16] http://commons.apache.org/fileupload
[17] http://www.json.org
[18] http://code.google.com/p/json-simple

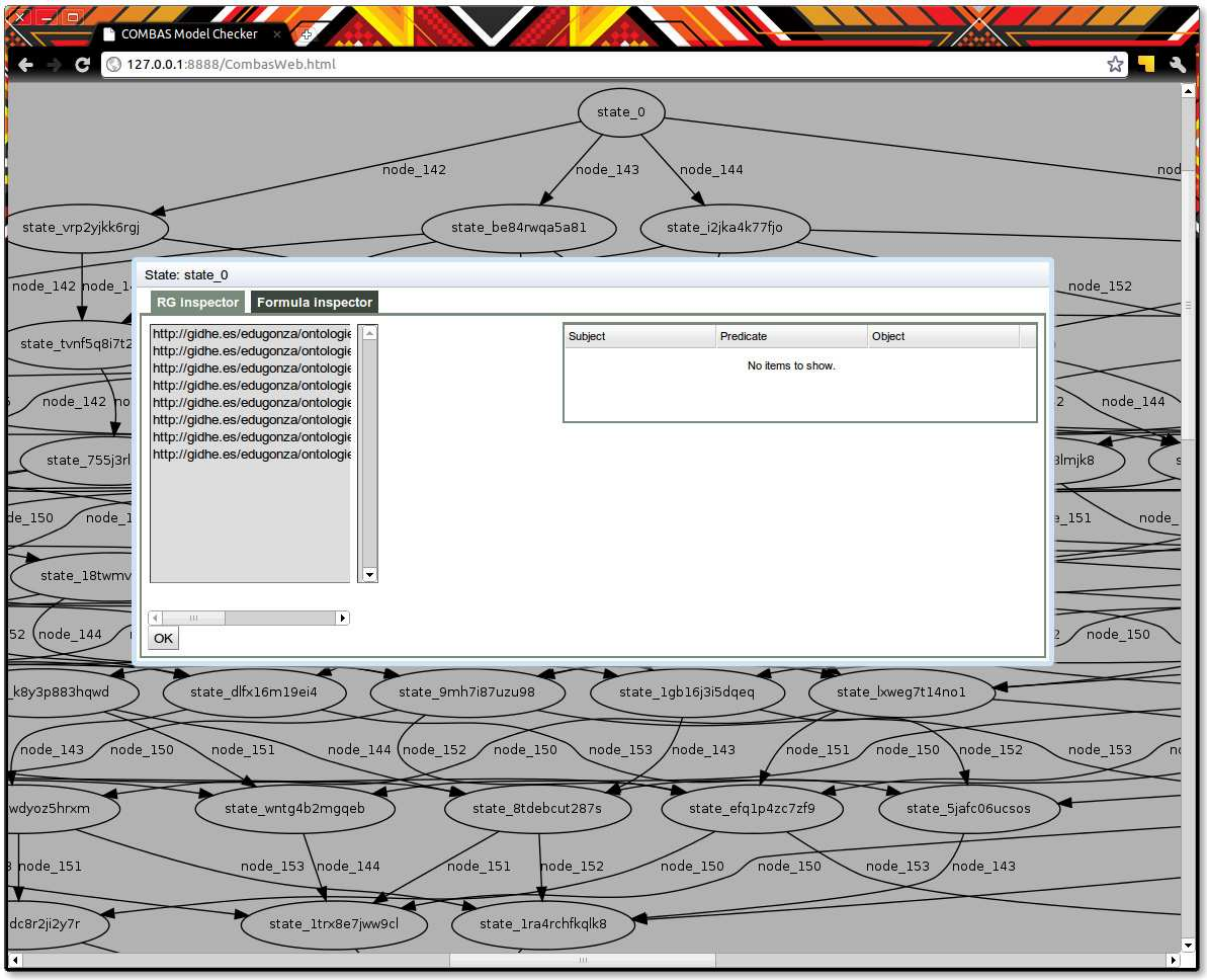

Figura 4.6: Revisión del marcado de un grafo de alcanzabilidad en la aplicación *Web* de COMBAS.

alcanzabilidad, y una ventana que muestra el marcado correspondiente a uno de los estados de dicho grafo.

Capítulo 5 | Análisis del *workflow* EBI InterProScan

Para mostrar la utilidad del *framework* COMBAS, en esta sección se analiza un caso de gran interés en la comunidad científica. Se trata del *workflow* InterProScan. Este *workflow*, que utiliza el servicio *EBI's WSInterProScan* [1], puede ser consultado en la comunidad *Myexperiment.org* [2].

5.1 Planteamiento original

El *workflow*, cuya estructura puede ser visualizada en la Figura 5.1 recibe como entrada una secuencia de proteínas, una dirección de correo electrónico y algunos parámetros adicionales para el análisis. Partiendo de dichos datos, se busca la caracterización de una secuencia proteica en una base de datos de familias de proteínas integradas dentro de *InterPro* [3]. Como resultado, se obtiene un conjunto de coincidencias debidamente formateadas. Dichos resultados son anotados con las asignaciones de términos *InterPro* y *GO* correspondientes (para más detalles sobre la asignación de dichos términos, consultar la *Web* del experimento). Para llevar a cabo la ejecución de la tarea *runInterProScan*, existen dos servicios *Web* en el repositorio: *runInterProScan1* y *runInterProScan2*. Ambos servicios son capaces de realizar el análisis de proteínas, que representa la parte más cara del experimento, en términos computacionales.

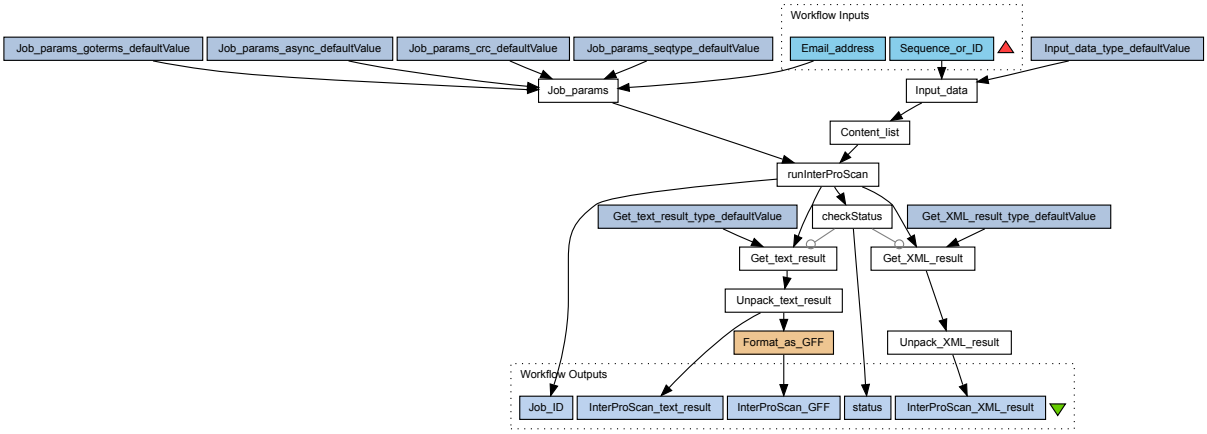

Figura 5.1: Planteamiento en taberna del *workflow* InterProScan para el análisis de proteínas, extraído de la comunidad *Myexperiment.org*.

[1] http://www.ebi.ac.uk/Tools/webservices/services/interproscan
[2] http://www.myexperiment.org/workflows/814.html
[3] http://www.ebi.ac.uk/interpro/

5.2 Modelado del *workflow*

La Figura 5.2 muestra el *workflow* modelado como una red de Petri de algo nivel usando la herramienta Renew [23] y anotada semánticamente de acuerdo con el formalismo U-RDF-PN paramétrico. Los nodos que corresponden al *workflow* original aparecen representados en colores azul, verde y naranja, mientras que algunas estructuras adicionales han sido añadidas con el fin de proporcionar más generalidad en el grafo de alcanzabilidad generado (representando una mayor variedad de casos y estados a analizar) que aparecen en color púrpura.

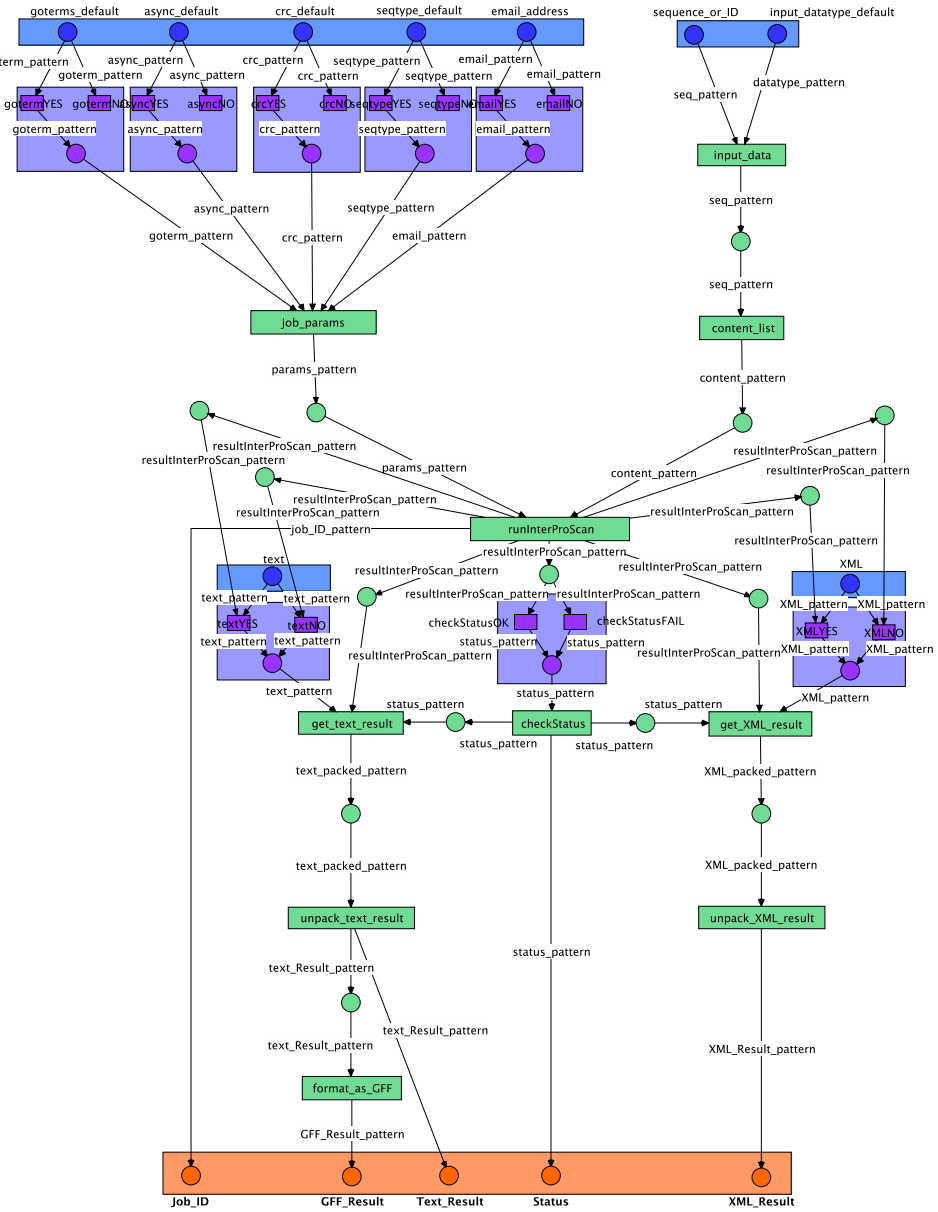

Figura 5.2: U-RDF-PN paramétrica que modela el *workflow* InterProScan para el análisis de una secuencia de proteínas.

A continuación se describirá brevemente la red. Podemos decir que se trata de una red paramétrica, ya que incluye referencias a parámetros en su marcado, en el sentido de que no tienen, a priori, asociado un valor concreto. Los siete lugares anotados en la parte superior de la red (en color azul) son las entradas principales del *workflow*. Cinco de estas entradas, que representan los parámetros de la tarea y son necesarias para configurar correctamente el servicio *InterProScan*, están agrupadas en el lado izquierdo:

goterms_default, async_default, crc_default, seqtype_default y email_address. Estas entradas son necesarias para que la transición *job_params* se sensibilice y pueda ser disparada. En el lado derecho, dos lugares representan los datos de la secuencia de proteínas: sequence_or_ID y input_datatype_default. Dichos datos son también necesarios para disparar la transición *Input_data*.

Observando la estructura de la red, es visible que los parámetros mencionados son necesarios para la ejecución del servicio *InterProScan*. Cuando este servicio es ejecutado, el proceso entra en la mitad inferior de la red, en la cual la salida es transformada al formato adecuado. Hay dos lugares encargados de seleccionar dicho formato, y también representan una entrada (en color azul) del *workflow*: 1) text y 2) XML. Estos lugares están anotados semánticamente conteniendo, cada uno de ellos, la referencia a un parámetro. Bajo ambos lugares, hay una sección añadida en color púrpura. El fin de dicha estructura es asignar dos valores diferentes a estos parámetros (cierto o falso), de forma que sea posible analizar el comportamiento del *workflow* en cualquier situación, usando un solo grafo de alcanzabilidad. La misma técnica se ha utilizado en la parte superior de la red, donde se sitúan los cinco parámetros descritos anteriormente.

En la parte inferior (en color naranja), hay cinco lugares de salida: Job_ID, GFF_Result, Text_Result, XML_Result y Status. Estos son los lugares en los que se almacenará el resultado del *workflow*. Representan los cinco tipos de datos que podemos obtener: el identificador del trabajo, el resultado en formato GFF, texto y XML, y el valor de estado, respectivamente.

5.3 Generación del RG

El modelo presentado es, entonces, procesado por el generador de RG con el fin de computar el grafo. El grafo de alcanzabilidad resultante fue construido (en un sistema Debian GNU/Linux con kernel optimizado 2.6.38.4, procesador Intel Core 2 Duo E7400 de 64bits de dos *cores* a 2.80 GHz, 4GB de RAM, E/S SATA y JVM JRE6u25) en 1343 segundos, y está compuesto de 798 estados. Debido al tamaño de la estructura resultante, se presenta una versión resumida en la Figura 5.3, que esquematiza su morfología.

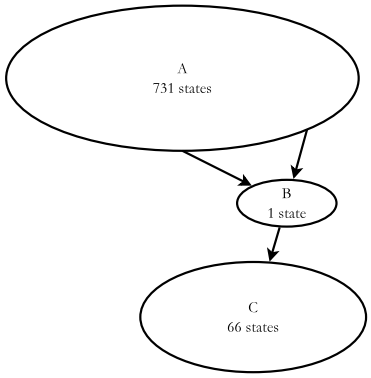

Figura 5.3: Vista parcial del grafo de alcanzabilidad correspondiente al modelo de la Figura 5.2.

La parte superior del grafo de alcanzabilidad generado (parte A) está compuesta por una colección de 731 estados. De acuerdo con el modelo descrito en la Figura 5.2, es sencillo calcular aproximadamente el número de estados esperados. En la parte superior, encontramos cinco entradas (arriba a la izquierda), y cada una puede estar en tres estados diferentes. Esto se traduce en 3^5 combinaciones. Entonces, otro estado se añade como resultado del disparo de la transición *job_Params*. Finalmente, la parte derecha muestra otros tres estados posibles: 1) los lugares de entrada están marcados; 2) *input_data* es disparado; 3) *content_list* es disparado. Estos tres estados, combinados con los del lado izquierdo suponen un total de $(3^5 + 1) * 3 = 732$ estados. En el diagrama se ha descompuesto la suma en dos grupos (A y B), siendo el estado en B el correspondiente al momento justamente anterior a la ejecución de la transición *runInterProScan*. Por lo tanto, la parte inferior del grafo muestra una complejidad menor, estando formada por 66 estados (C), los cuales, sumados a los grupos A y B, hacen un total de 798 estados.

5.4 Comprobación de propiedades

Debido al tamaño del grafo de alcanzabilidad generado, realizar una verificación manual no es factible, y supondría una tarea extremadamente compleja y tediosa. Esto presenta la situación perfecta para aplicar la solución propuesta para analizar el problema. A continuación pasamos a diseñar y formular algunas consultas sobre el modelo, con el fin de verificar su corrección. Dichas consultas son expresadas como fórmulas en lenguaje CTL y han sido implementadas usando el formato XML correspondiente, el cual es utilizado para su procesamiento por COMBAS. El *model checker* es el encargado de verificar la satisfacibilidad de cada fórmula.

La primera consulta a considerar cuando se verifica un *workflow* podría ser la que comprueba si el proceso es capaz, en algún caso, de finalizar de forma adecuada. Esto ocurrirá si los lugares de salida de la red son marcados con un grafo resultado. Existen cinco lugares de salida diferentes: Job_ID place; Text_Result place; GFF_Result place; Status place y, finalmente, XML_Result place. Estos lugares se corresponden con los localizados en la parte inferior de la red presentada en la Figura 5.2. Por lo tanto, para verificar si dichos lugares son alcanzados, es posible comprobar si existe algún estado en el que dichos lugares se encuentren marcados con cualquier grafo (el grafo vacío representa el concepto de 'cualquier grafo'). Esta consulta aparece expresada como un predicado CTL de la siguiente forma: $EF(RDFG_{Text\_Result} \vee RDFG_{GFF\_Result} \vee RDFG_{XML\_Result} \vee RDFG_{Status} \vee RDFG_{Job\_ID})$

Cuando esta fórmula es ejecutada en el *model checker*, se obtiene la siguiente salida:

```
INFO [main] (CombasApp.java:240) - Model satisfies the formula!
INFO [main] (CombasApp.java:250) - Output files generated.
INFO [main] (CombasApp.java:254) - Checked in 17159 millis
INFO [main] (CombasApp.java:255) - Formula: formula_1h9wbb3xwwj7c
INFO [main] (CombasApp.java:256) - Model: netId1337458105565_RG
```

Esto quiere decir que el modelo satisface la fórmula, y por tanto es posible alcanzar un estado donde alguno de los lugares especificados esté marcado. Es posible comprobar el resultado haciendo uso del interfaz *Web*. En la Figura 5.4 podemos consultar el resultado de la fórmula en el estado inicial del grafo de alcanzabilidad (*state_0*), de forma que vemos qué nodos de la fórmula se satisfacen.

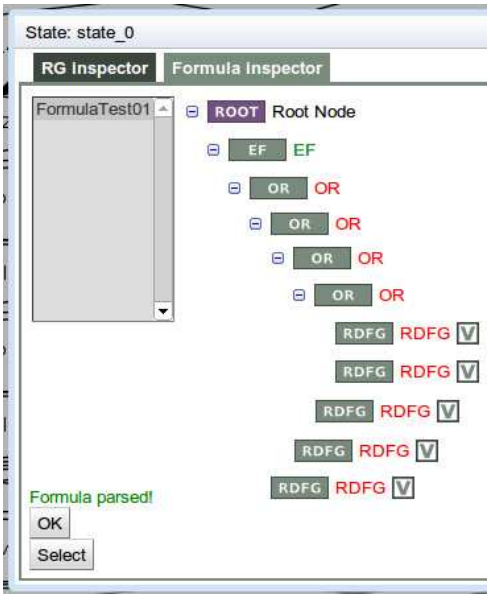

Figura 5.4: Vista del resultado de la fórmula en el interfaz *Web*.

Vemos que algunos nodos aparecen en rojo (no se satisfacen) y otros en verde (sí se satisfacen). Esto quiere decir si se satisfacen o no en el estado actual. El la Figura 5.4 se trata del estado inicial, en el cual no se cumple ninguna de las condiciones *OR* ni *RDFG*, ya que este estado no contiene dichas marcas en

los lugares especificados. Sin embargo, sí que existe algún camino futuro en el cual, en algún estado, se cumple alguna de esas propiedades. Por eso el nodo EF aparece en color verde.

El siguiente paso será verificar si el *workflow* es siempre capaz de finalizar. La fórmula correspondiente es: $AF(RDFG_{Text\_Result} \vee RDFG_{GFF\_Result} \vee RDFG_{XML\_Result} \vee RDFG_{Status} \vee RDFG_{Job\_ID})$

Como se puede observar, la fórmula es muy similar a la anterior, a excepción del nodo raíz que en este caso, en lugar de EF, es AF. Esto quiere decir que, en lugar de buscar un camino que satisfaga la condición, buscamos que todos los caminos la satisfagan. El resultado del *model checker* establece que dicha fórmula no es satisfecha por el modelo. Esto quiere decir que dicho modelo no siempre es capaz de alcanzar alguno de esos lugares en la red. Esto puede ocurrir, por ejemplo, cuando la red alcanza un estado de bloqueo en el que ninguna transición está lista para ser disparada.

Debido a la morfología de la red, para que la transición *runInterProScan* sea disparable, sus lugares fuente deben estar marcados. Estos lugares, a los que nos referiremos como *params* y *content_ list*, estarán marcados en alguna situación. Si no fuera así, la primera consulta no habría sido satisfecha por el modelo (la que pregunta si es posible alcanzar en algún caso un lugar de salida). Entonces, sería interesante saber si siempre que los dos lugares mencionados estén marcados, sería posible alcanzar algún estado de salida. La fórmula CTL que expresa esta consulta es la siguiente:$AG((RDFG_{params} \wedge RDFG_{content}) \rightarrow AF(RDFG_{GFF\_Result} \vee RDFG_{Text\_Result} \vee RDFG_{Status}))$.

Vemos que está formada por un operador $P \rightarrow Q$, que quiere decir que será cierto siempre que P y Q sean ciertos, o P sea falso. Esto es, que la fórmula será evaluada a cierto si, siempre que existan marcas en ambos lugares, se cumple que para todos los caminos desde dicho lugar existe algún estado futuro en el que los lugares de salida estén marcados. Cuando es procesada, el *model checker* establece que el modelo satisface la propiedad.

Otra consulta útil es la que verifica si siempre que se obtiene un resultado (al menos uno de los lugares de salida está marcado), el valor del resultado de estado es *true* (es decir, que el *token* en el lugar *status* tiene el valor *true* para la variable *status*). Dicha propiedad en CTL es: $AG((RDFG_{Text\_Result} \vee RDFG_{GFF\_Result} \vee RDFG_{XML\_Result}) \rightarrow RDFGP_{Status(true)})$. Se ha usado el operador AG para indicar que la propiedad ha de cumplirse siempre, se comprueba que si alguno de los resultados ha sido obtenido (alguno de los lugares de salida está marcado), entonces el lugar de *status* debe estar marcado y con valor *true*. Si no existe resultado, entonces la fórmula también será cierta, por el significado del operador $\rightarrow$. El predicado es satisfecho por el modelo, lo que quiere decir que siempre que obtenemos un resultado, la variable *status* tendrá valor *true*.

Tal y como se a demostrado, el modelo es susceptible de ser analizado por medio de consultas CTL correspondientes a las propiedades que queremos verificar. Una vez que el modelo inicial y sus anotaciones correspondientes han sido definidos, COMBAS permite realizar de forma sencilla una comprobación del modelo, visualizar en el grafo de alcanzabilidad qué estados no verifican las propiedades, revisar los resultados, etc. No se requiere conocimiento previo del algoritmo de *model checking* ni detalles internos y tecnológicos por parte del científico. El interfaz incluido en el *framework*, ofrece un conjunto de funcionalidades y herramientas avanzadas para el análisis de sistemas complejos.

Información adicional sobre las anotaciones semánticas y paramétricas del modelo objeto de estudio puede ser consultada en el Anexo A.

Capítulo 6 | Conclusiones y trabajo futuro

Para finalizar, en este capítulo se presentan las conclusiones extraídas de este Trabajo Fin de Máster, y se presentan las líneas de investigación que se han abierto a partir de su desarrollo. Estas líneas engloban áreas como la optimización en la ejecución de los algoritmos relacionados con el proceso de *model checking* por medio de técnicas de ejecución *Grid* y *Cloud*, extensión de COMBAS para dar soporte a los principales estándares de modelado en computación científica y, finalmente, la utilización de la herramienta para el análisis de problemas complejos en entornos reales.

6.1 Conclusiones

A lo largo de este trabajo se ha resaltado la importancia que han cobrado los *workflows* científicos en el mundo de la computación. Los *workflows* científicos abarcan tareas de cómputo del mundo científicoque generalmente representan un gran coste computacional, tanto en tiempo como en recursos. Por tanto, la verificación de estos sistemas adquiere una gran importancia, especialmente la comprobación previa a la ejecución.

El análisis de las alternativas actuales en el campo de la verificación de modelos basados en semántica dentro del área de la computación científica ha dejado en evidencia las carencias existentes, especialmente en cuanto al alcance y la expresividad de los mismos, y ha asentado las bases para el desarrollo de este Trabajo Fin de Máster. En concreto, refiriéndonos a la utilización de semántica en el mundo de los *workflows* científicos, vemos que se está aplicando, aunque de forma más orientada a la recolección y análisis de datos de *provenance* y descubrimiento de servicios y recursos, entre otros. Sin embargo, no se utiliza para la verificación previa.

La importancia de integrar semántica en los modelos radica en el aumento de la expresividad que ofrece. Gracias a su uso, es posible especificar *workflows* con mayor detalle, modelando propiedades de los datos y su flujo a través del mismo y permitiendo describir problemas de mayor complejidad. Por esto, vemos la necesidad de contar con herramientas de verificación que permitan comprobar el correcto funcionamiento de *workflows*, haciendo uso de aspectos semánticos.

Así, se ha desarrollado el *framework* COMBAS con el fin de suplir dichas carencias y, además, se ha extendido la herramienta para incorporar el análisis paramétrico de *workflows* científicos. Como resultado se ha desarrollado un *framework* que abarca el ciclo completo de verificación, desde el modelado hasta el análisis de resultados y retroalimentación, pasando por la verificación, creación y edición de consultas y visualización de resultados. COMBAS representa una herramienta extensible, escalable y de fácil utilización por parte del usuario que facilita la tarea de definición y verificación de *workflows* anotados semánticamente. Es posible modelar sistemas complejos usando redes de Petri con anotaciones semánticas, incluir condiciones en lógica proposicional, y analizar casos generales usando parámetros cuyo valor inicial no está especificado.

Uno de los puntos clave de este trabajo ha sido la incorporación de soporte paramétrico en la herramienta. Los parámetros permiten modelar datos en el sistema a los que no se asigna un valor concreto de entrada. De esta forma, podemos verificar los sistemas desde un punto de vista general, sin tener en cuenta datos de entrada concretos que limitan el comportamiento del *workflow* a un caso determinado. Esto ha implicado la integración en el *framework* de técnicas y herramientas que permitan tratar con dichos parámetros. Para ello se ha recurrido a las Satisfiability Modulo Theories (SMT) que nos permiten

resolver el problema de decisión para las fórmulas lógicas que especifican los parámetros, y se han valorado las herramientas SMT existentes, así como estándares y lenguajes de descripción de proposiciones.

Finalmente se ha demostrado la utilidad de COMBAS aplicándolo a un caso de uso concreto, el análisis del *workflow* de análisis de proteínas EBI InterProScan. Se ha mostrado el proceso de modelado, que incluye la adición de anotaciones en base al estándar RDF, la creación de consultas mediante lógica temporal y la verificación de propiedades sobre el modelo, mostrando los resultados obtenidos para consultas relevantes sobre el funcionamiento del *workflow* y ciertas características que resultan de interés para el diseñador del sistema o el científico. Ha sido posible detectar casos de bloqueo, y analizar condiciones de finalización del *workflow* que, de otra forma, habría sido imposible detectar sin una implementación previa del experimento y su ejecución, hechos que implican unos costes elevados en tiempo y recursos.

En definitiva, se ha mostrado la viabilidad del uso de COMBAS y su gran utilidad como verificador y analizador de *workflows* en el ámbito científico. Se han cumplido los objetivos propuestos y se han resuelto las necesidades más inmediatas detectadas en el campo. COMBAS representa una solución viable y útil, basada en un formalismo bien estudiado, y que resuelve una necesidad relevante y creciente dentro del mundo de los *workflows* científicos.

6.2 Trabajo futuro

El desarrollo y el análisis previo de este Trabajo Fin de Máster han abierto una serie de líneas de investigación que se abordarán como parte de la carrera investigadora inmediata del autor. A continuación se detallan las principales líneas.

- Búsqueda de un algoritmo de generación de RGs que aproveche la potencia de infraestructuras distribuidas de *Cloud* y *Grid*. Los tiempos de generación de las estructuras internas para el proceso de model checking resultan excesivos cuando la complejidad del modelo de entrada aumenta. Esto resulta especialmente evidente en el caso de la generación del grafo de alcanzabilidad. La posibilidad de utilizar infraestructuras de computación Grid y Cloud en el Instituto de Investigación en Ingeniería de Aragón (I3A) y en el Instituto Universitario de Investigación, Biocomputación y Física de Sistemas Complejos (BIFI) abre una puerta para la investigación de cómo estos paradigmas pueden resultar beneficiosos para la mejora y optimización de los algoritmos relacionados. Esto permitirá tiempos de generación menores y mayor capacidad de almacenamiento, extendiendo la aplicación de COMBAS a modelos de escala mucho mayor, cuyo tratamiento está muy limitado con el enfoque actual.
- Extensión del *framework*, ofreciendo soporte para otros formalismos de modelado y estándares de diseño, como por ejemplo Business Process Model and Notation [1] (BPMN), o herramientas específicas para el modelado de *workflows* científicos muy extendidas y populares (como Taverna o Kepler, por ejemplo).
- Aplicación de la técnica de verificación a más ejemplos de problemas complejos. Evidentemente, la mejora y extensión de COMBAS permitirá resolver problemas con una complejidad mayor, lo que permitirá demostrar su utilidad, a la par que impulsar la búsqueda de nuevas necesidades en el *framework*.
- Aplicación del *framework* para la verificación de nuevas fuentes de datos. Un caso de especial interés es el análisis de datos de *provenance*, los cuales pueden tener un origen variado, por ejemplo de resultados de ejecución de *workflows*. Esto permitirá estudiar el comportamiento de acciones ocurridas en el pasado y ayudar en la reproducción de experimentos. También es de interés el análisis de datos de *provenance* en la *Web* de *Linked Data*, tema tratado en [17], de forma que sea posible estudiar la procedencia de la información compartida en ámbitos científicos y comprobar su fiabilidad y origen.
- Extensión de los mecanismos de deducción. Actualmente es posible hacer uso de la inferencia que nos ofrece la semántica. Sin embargo, sería interesante ampliar el *framework* dotándolo de mayor capacidad de deducción, de forma que sea capaz de analizar o predecir comportamientos de forma más eficaz, a la vez que extraer conclusiones de mayor utilidad.

[1] Para más información sobre el estándar BPMN, consultar http://www.bpmn.org/

Bibliografía

[1] Arc - altarica checker. http://altarica.labri.fr/forge/projects/arc.

[2] Cosyverif environment. http://www.cosyverif.org/.

[3] Game-based, easy and reverse model-checking. http://jabc.cs.tu-dortmund.de/modelchecking/.

[4] Dean Allemang and James Hendler. *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[5] Chad Berkley, Shawn Bowers, Matthew Jones, Bertram Ludäscher, Mark Schildhauer, and Jing Tao. Incorporating Semantics in Scientific Workflow Authoring. In *SSDBM 2005*, pages 75–78, 2005.

[6] F. Calzolai, R. De Nicola, M. Loreti, and F. Tiezzi. Tapas: a tool for the analysis of process algebras. *Transactions on Petri Nets and Other Models of Concurrency I*, pages 54–70, 2008.

[7] Maria Cláudia Cavalcanti, Rafael Targino, Fernanda Bai ao, Shaila C. Rössle, Paulo M. Bisch, Paulo F. Pires, Maria Luiza M. Campos, and Marta Mattoso. Managing Structural Genomic Workflows using Web Services. *Data and Knowledge Engineering*, 53(1):45–74, 2005.

[8] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 241–268. Springer, 2002.

[9] R. Cleaveland, T. Li, and S. Sims. The concurrency workbench of the new century, version 1.2-user's manual. 2000.

[10] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, H. Zheng, et al. Bandera: Extracting finite-state models from java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448. IEEE, 2000.

[11] Tommy Ellqvist, David Koop, Juliana Freire, Cláudio Silva, and Lena Stromback. Using mediation to achieve provenance interoperability. In *Proceedings of the 2009 Congress on Services (SERVICES 2009), Los Angeles, California, USA, July 6-10, 2009*, pages 291–298, Washington, DC, USA, 2009. IEEE Computer Society.

[12] E. Allen Emerson. Temporal and Modal Logic. *Handbook of Theoretical Computer Science*, Formal Models and Semantics:995–1072, 1990.

[13] J. Frey. Condor dagman: Handling inter-job dependencies. *University of Wisconsin, Dept. of Computer Science, Tech. Rep*, 2002.

[14] Carole Goble, Jiten Bhagat, Sergejs Aleksejevs, Don Cruickshank, Danius Michaelides, David Newman, Mark Borkum, Sean Bechhofer, Marco Roos, Peter Li, and David De Roure. myExperiment: a repository and social network for the sharing of bioinformatics workflows. *Nucleic Acids Research*, 2010.

[15] E. González-López de Murillas, J. Fabra, P. Álvarez, and J. Ezpeleta. Combas: una herramienta para el análisis de procesos con información semántica. In *VII Jornadas de Ciencia e Ingeniería de Servicios (JCIS)*, 2011.

[16] Tomasz Gubala, Daniel Herezlak, Marian Bubak, and Maciej Malawski. Semantic composition of scientific workflows based on the petri nets formalism. In *E-SCIENCE'06*.

[17] Olaf Hartig and Jun Zhao. Publishing and consuming provenance metadata on the web of linked data. In Deborah McGuinness, James Michaelis, and Luc Moreau, editors, *Provenance and Annotation of Data and Processes*, volume 6378 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010.

[18] Patrick Hayes. RDF Semantics. Technical report, W3C, February 2004. http://www.w3.org/TR/rdf-mt/.

[19] T. Herault, R. Lassaigne, and S. Peyronnet. Apmc 3.0: Approximate verification of discrete and continuous time markov chains. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 129–130. IEEE, 2006.

[20] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 441–444, 2006.

[21] María José Ibáñez, Javier Fabra, Pedro Álvarez, and Joaquín Ezpeleta. Model checking analysis of semantically annotated business processes. *Systems, Man, and Cybernetics – Part A: Systems and Humans*, 2012.

[22] J.P. Katoen, M. Khattri, and IS Zapreevt. A markov reward model checker. In *Quantitative Evaluation of Systems, 2005. Second International Conference on the*, pages 243–244. IEEE, 2005.

[23] O. Kummer and F. Wienberg. Renew - the Reference Net Workshop. In *Tool Demonstrations, 21st International Conference on Application and Theory of Petri Nets*, pages 87–89, 2000.

[24] Zoé Lacroix, Christophe Legendre, and S. Tuzmen. Reasoning on Scientific Workflows. In *Proceedings of the 2009 IEEE Congress on Services (SERVICES 2009), Los Angeles, California, USA, July 6-10, 2009*, pages 306–313, 2009.

[25] A. Lomuscio and F. Raimondi. Mcmas: A model checker for multi-agent systems. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 450–454, 2006.

[26] K. McMillan. Applying sat methods in unbounded symbolic model checking. In *Computer Aided Verification*, pages 303–323. Springer, 2002.

[27] David Newman, Sean Bechhofer, and David De Roure. myexperiment: An ontology for e-research. In *Semantic Web Applications in Scientific Discourse*, August 2009.

[28] Alun D. Preece, Binling Jin, Edoardo Pignotti, Paolo Missier, Suzanne M. Embury, David Stead, and Al Brown. Managing Information Quality in e-Science Using Semantic Web Technology. In *Procedings of the 3rd European Semantic Web Conference (ESWC 2006), Budva, Montenegro, June 11-14, 2006, Proceedings*, pages 472–486, 2006.

[29] Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[30] Yong Zhao, Michael Wilde, and Ian T. Foster. Applying the Virtual Data Provenance Model. In *Provenance and Annotation of Data, International Provenance and Annotation Workshop (IPAW 2006), Chicago, IL, USA, May 3-5, 2006*, pages 148–161, 2006.