Universidad Complutense de Madrid

Facultad de Informática

PROYECTO DE SISTEMAS INFORMÁTICOS

CURSO 2008/2009

FlexiMC Framework

FrameWork Flexible para Model Checking

Autores:
Iván Mikovski
José Antonio González
Nicolás Mon

Directores:
José Luis Sierra Rodríguez
Rubén Fuentes Fernández

Se autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria como el código, la documentación y/o el prototipo desarrollado.

Nicolás Mon Trortti José Antonio González M. Iván Mikovski

AGRADECIMIENTOS:

Este es el fruto de bastante esfuerzo y perseverancia,
pero que a pesar de los efectos colaterales ha merecido la pena.
Por ello queremos agradecer la paciencia
a todos los que han contribuido a facilitarnos dicha labor.
En especial a nuestros familiares, seres queridos, amigos y colegas,
entre los que destacan los que no hace falta nombrar.

Resumen

Los algoritmos de verificación de modelos (*Model Checking*) verifican una fórmula lógica sobre un modelo. Esta técnica permite tanto comprobar un correcto funcionamiento como descubrir errores de diseño y es aplicable a una gran variedad de campos. Una desventaja común al utilizar esta técnica es que dichos modelos deben ser traducidos al lenguaje concreto del sistema de *Model Checking*. El *framework* que aquí se presenta permite adaptar la técnica a los lenguajes, soportando la definición y uso de lenguajes específicos del dominio de forma directa, además de otras facilidades como permitir una visualización de contraejemplos personalizada o una gran flexibilidad en los algoritmos a usar.

Palabras clave: Model Checking, Framework, lenguajes específicos de dominio, contraejemplos, CTL

Abstract

Model Checking algorithms verify whether a logical formula holds in a model. Such technique allows checking the correct functionality of the modeled system as well as finding design mistakes, and can be applied in a wide variety of fields. A typical disadvantage of using this technique is the need to translate such models to the concrete Model Checking system language. The framework presented here allows adapting the technique to different languages, supporting the definition and use of domain specific languages directly. It also incorporates some other facilities like the creation of customized counter-example viewers or allowing the replacement of the specific algoritms used in the checking.

Keywords: Model Checking, Framework, domain-specific languages, counterexamples, CTL

Tabla de contenidos

Figuras

I. Introducción

I.1 Motivación

En el diseño de protocolos, juegos, simuladores, circuitos y, en general, cualquier sistema abstraído como máquina de estados, puede ser importante garantizar que el modelo que se ha diseñado cumple las especificaciones deseadas. Una verificación formal puede encontrar errores en el diseño, pero puede llegar a ser muy costosa y no se puede garantizar que esté exenta de errores humanos. Por lo tanto, se requieren nuevas herramientas y métodos de verificación automáticos que sean útiles cuando la verificación manual sea inviable.

De estas ideas nace el *Model Checking* [1], un método viable y automático para verificar propiedades sobre un modelo[I]. Consiste básicamente en traducir las propiedades a fórmulas de lógica temporal y verificar que las propiedades se cumplen estado a estado, usando algoritmos guiados por la definición de las fórmulas temporales. Dicho de otra manera, permite verificar las propiedades sobre una simulación del sistema explorando todas sus posibles líneas temporales.

Para ello es necesario abstraer el sistema como máquina de estados, traducir las propiedades a sus correspondientes fórmulas lógicas temporales, y explorar el espacio de estados verificando que, sea cual sea el comportamiento del sistema, siempre se cumplan dichas fórmulas. La máquina de estados debe proveer entonces, para cada estado, los estados destino de todas las transiciones que se le puedan aplicar. El conjunto de transiciones aplicables a un estado representan las distintas evoluciones temporales que puede sufrir el sistema a partir de ese estado frente a posibles cambios (eventos, entradas, etc.). Dichas transiciones dependen totalmente del aspecto del modelo que se quiere estudiar. Considerándolo de una forma práctica, en cada estado se debe estudiar qué eventos (externos o internos) pueden provocar un cambio de estado, cuáles de ellos son aplicables o son interesantes para el aspecto concreto a verificar del sistema objetivo, y determinar para cada evento (o combinación de eventos) el estado final al que se llega. El lector interesado en más detalles sobre las máquinas de estado puede encontrar abundante información al respecto en cualquier libro de máquinas de estados o de inteligencia artificial; p.e. [2, 3].)

El conjunto de transiciones desde un estado fija un conjunto de estados posteriores temporalmente. No obstante,estos estados pueden ser estados repetidos en esta línea temporal o en otra (sólo si en la definición de estado no se tiene en cuenta explícitamente el tiempo). La granularidad del paso de tiempo queda oculta tras la abstracción que se haya hecho del modelo y por lo tanto es transparente al algoritmo de *Model Checking*. Para el algoritmo, cada transición representa un avance temporal, de manera discreta, y no se sabe cuánto tiempo representa cada transición.

El espacio de estados que el algoritmo espera como entrada es entonces un grafo dirigido sin pesos. Más adelante se explicará que para la visualización de contraejemplos se pide que las aristas estén etiquetadas con un nombre.

[I] Un modelo en Model Checking se puede considerar como cualquier sistema, protocolo,… (u otro artefacto) abstraído como una máquina de estados

El algoritmo comienza en un estado inicial del sistema y explora el grafo verificando una sola propiedad a la vez. Esta exploración no recorre todo el espacio de estados necesariamente. Por un lado la verificación se detiene en cuanto se pueda determinar un resultado para la fórmula (ya se puede decir que la fórmula es cierta o falsa). Por otro lado, los estados explorados dependen de la fórmula que se quiera verificar ya que el algoritmo está guiado por la semántica de la fórmula. Por ejemplo, el caso más básico y claro es la verificación de una fórmula formada únicamente por una proposición, en la que solamente se explora el estado inicial.

Dado que se pueden expresar las propiedades sobre los modelos utilizando diferentes lógicas temporales, se han desarrollado varios algoritmos de *Model Checking*. Estos algoritmos se diferencian en la lógica temporal utilizada y en las optimizaciones que aplican según el tipo particular de sistema verificado.

Las herramientas y métodos de *Model Checking* han sido ampliamente usadas en ámbitos reales [4] en los cuales se requiere una cierta garantía de seguridad en el funcionamiento de los sistemas modelados. Algunos de estos ámbitos son la electrónica [5] o el diseño de protocolos para usos como el comercio electrónico [6].

En la actualidad existen muchas herramientas de *Model Checking*, de efectividad comprobada y gran eficiencia [7-9]. La mayor parte de ellas requieren que el modelo se adapte a un lenguaje de especificación propio [10-12]. Aunque estos lenguajes incorporan distintas construcciones de alto nivel para facilitar su uso, sigue siendo difícil para el usuario adaptar su especificación del modelo al lenguaje específico que toma como entrada la herramienta de verificación. También hay que indicar que muchas herramientas de chequeo están orientadas a una única lógica temporal[I], ya sea CTL, LTL, CTL*,… Estos aspectos restan flexibilidad a muchas de las herramientas actuales, lo que provoca que no se usen en ámbitos en los que la aplicación del *Model Checking* podría ser muy beneficiosa.

[I] En el contexto del *model cheking* es usual que el uso de lógicas temporales sea para especificar la propiedad a comprobar

I.2 Objetivos

Como antes se ha mencionado, la técnica de *Model Checking* es una forma de verificación muy útil y viable, ya que está automatizada. No obstante, presenta ciertos problemas de flexibilidad en su aplicación a ciertos ámbitos. El principal es la necesidad que tiene el usuario de adaptarse al lenguaje y algoritmo concreto de la herramienta que va a utilizar, sin considerarse el dominio de su problema. Este proyecto nace como una opción para conseguir una herramienta de *Model Checking* flexible y fácil de utilizar, incluso para usuarios con escasos conocimientos de esta técnica. Para ello pretende flexibilizar varios aspectos de la aplicación de la técnica que se discuten brevemente a continuación.

Se pretende que el *Framework* que se va a desarrollar deje libertad al usuario para poder usar su propio lenguaje específico de dominio. En esta línea, el *Framework* toma como entrada un intérprete para un lenguaje y una especificación en dicho lenguaje, que es el modelo. El intérprete codifica la semántica operacional del lenguaje y se encarga de traducir el modelo a una máquina de estados. El algoritmo de *Model Checking* pide al intérprete únicamente los estados iniciales del sistema y las transiciones para cada estado. Es decir, el intérprete, desde el punto de vista del algoritmo de *Model Checking*, es una caja negra que le proporciona un grafo dirigido.

Con tal grado de libertad, el usuario puede implementar intérpretes para distintos lenguajes. Normalmente usará el lenguaje específico de dominio que le sea más cómodo para especificar el modelo, pudiendo ser el mismo lenguaje reutilizado de otra herramienta. También podría implementar el intérprete sin necesidad de utilizar un lenguaje específico de dominio (por ejemplo si el intérprete fuera un módulo de una herramienta que permita crear el modelo mediante una interfaz gráfica). Lo importante es que el usuario proporcione el modelo como máquina de estados, sin estar restringido por el algoritmo de verificación o por la lógica usada para definir las propiedades. Con esto se pretende permitir flexibilidad en el uso de la herramienta y acercarla a cualquier ámbito de diseño, aunque teniendo en cuenta que esta flexibilidad puede tener un precio en rendimiento y eficiencia. Por ello, desde el primer momento del desarrollo del proyecto se ha tendido a vigilar el consumo de recursos.

En otra línea, el proyecto pretende dar la posibilidad de que se puedan usar diferentes técnicas de chequeo basadas en diferentes lógicas temporales. El *Framework* cuenta con una estructura diseñada para poder elegir fácilmente el algoritmo de *Model Checking* a utilizar y poder añadir en cualquier momento un nuevo algoritmo. Para ello se da una interfaz de algoritmo de *Model Checking* genérica, que proporciona gran libertad para computar la verificación. A un nivel más bajo se proporciona un conector de algoritmos que da flexibilidad para la incorporación de algoritmos a nivel de fórmula, ya que la mayoría de algoritmos de *Model Checking* utilizan un algoritmo específico para verificar cada tipo de fórmula. Este conector se comporta como un único algoritmo de *Model Checking* que permite elegir qué estrategia utilizar para cada tipo de fórmula y así crear un algoritmo mixto. El precio a pagar es que cada algoritmo utilizado (que únicamente se dedica a un tipo particular de fórmula) ha de coordinarse con el conector para intercambiar datos básicos para que la verificación modular funcione. Como ejemplo de esta flexibilidad se proporciona un algoritmo de *Model Checking* centrado en poder verificar máquinas de estados infinitas con dos implementaciones diferentes.

En este *Framework* por tanto se puede ampliar fácilmente el abanico de algoritmos en cualquiera de los dos niveles: a nivel de algoritmo (lo más común) y a nivel de fórmula.

En cuanto a las fórmulas a verificar sobre los modelos, el *Framework* ofrece interfaces para implementar las construcciones básicas de cualquier lógica y las proposiciones a utilizar en un problema concreto. En la actualidad se proporcionan las construcciones básicas de CTL, aunque, si se necesita, el usuario puede ampliar el *Framework* a otras lógicas temporales. En cuanto a las proposiciones, hay que señalar que estas cambian entre diferentes problemas, ya que las propiedades de un sistema se escriben en base a proposiciones que son particulares de dicho sistema. El usuario es el encargado de determinar cuáles son las proposiciones del sistema sobre las que quiere construir las fórmulas a verificar. Por tanto, la libertad de elección de las proposiciones en el *Framework* es obligada, y el desarrollo de las mismas responsabilidad del usuario. La implementación de una proposición debe indicar si un estado dado cumple la proposición. Tanto en el caso de las primitivas de una lógica como de las proposiciones para un problema, su implementación únicamente debe seguir una interfaz proporcionada por el *Framework* para poder usarse en el proceso de *Model Checking*.

Otro de los objetivos del proyecto ha sido acercar la forma de proporcionar contraejemplos a las necesidades del usuario. A fin de estudiar el sistema, no sólo hay que saber si el sistema cumple una propiedad, sino que en caso de fallo hay que conocer la secuencia de eventos y estados que lo ha producido para poder entonces descubrir la causa. Más precisamente, un contraejemplo es un subgrafo que permite comprobar que el sistema no cumple la propiedad verificada. La respuesta devuelta por el algoritmo debe avisar si la fórmula se cumple o no y, en caso de que no se cumpla, devolver un subgrafo del espacio de estados asociado al modelo del sistema donde se pueda evaluar que la fórmula es falsa. En realidad, por extensión, se puede devolver siempre un subgrafo donde se pueda evaluar un valor concreto de la fórmula. El algoritmo de *Model Checking*, mientras explora el espacio de estados, va almacenando los estados explorados y así, en cuanto puede determinar si la fórmula es cierta o falsa, devuelve parte del subgrafo explorado desde el estado inicial hasta el último estado computado. Por supuesto, el subgrafo devuelto depende de la semántica de la fórmula y se devolverá el contraejemplo más pequeño del que se disponga. Todo este almacenamiento de grafos penaliza la eficiencia del framework, pero se ha estimado que puede ayudar bastante al usuario final en su labor de diseño de modelos y descubrimiento de errores. Además, ilustrar la verificación de las propiedades con grafos extiende la práctica habitual en *Model Checking*, donde sólo se le muestran al usuario una secuencia única de estados y eventos.

No obstante, la mera descripción de un subgrafo del espacio de estados no basta para que el usuario comprenda la situación mostrada. Por tanto, el objetivo previo abre un nuevo sub-objetivo bastante relacionado. El sistema proporciona una respuesta y un subgrafo ejemplo/contraejemplo y se necesita poder visualizar dicho subgrafo. Para ello el *Framework* proporciona una estructura de clases e interfaces que permite, desarrollando únicamente un visor de estados, explorar el contraejemplo. De igual manera que con las proposiciones, el visor de estados depende totalmente del estado particular que se quiere visualizar y por ello no se puede proveer uno por defecto. El usuario entonces puede desarrollar un visor que permita visualizar un solo estado y utilizar el navegador de contraejemplos por defecto que permite explorar el subgrafo

avanzando a través de las transiciones. De esta manera puede encontrar dónde falla su sistema.

Opcionalmente, se puede querer desarrollar un navegador de contraejemplos propio si se quiere cambiar la presentación de los controles o se quiere unir a una herramienta propia. La estructura del *Framework* está preparada para facilitar dicha labor y también soporta la visualización en varios navegadores simultáneos.

Con todo esto el *Framework* ofrece generalidad en la interacción del *Model Checker* con el sistema a verificar, a la vez que flexibilidad y personalización en varios aspectos, tanto de cara al usuario final como de cara al desarrollador de algoritmos de *Model Checking.* Con ello se espera reducir parte de las limitaciones actuales de uso de esta técnica para usuarios no expertos en ella.

I.3 Estructura de la memoria.

En la primera sección trata sobre el estado del arte. En los primeros apartados (del II.1 al 0) se explicarán los fundamentos teóricos en los que se basan los algoritmos de *Model Checking*. Le sigue una presentación histórica de los *Model Checkers* más conocidos y una explicación detallada de las ideas en la que se basan sus optimizaciones (apartado II.6). Un usuario que esté iniciándose en este ámbito le es recomendable leer las secciones antes enumeradas para obtener una perspectiva inicial sobre *Model Checking*.

La sección siguiente (III) comienza con una comparativa entre las ventajas e inconvenientes de los sistemas de verificación automática actuales y los objetivos perseguidos en este proyecto (III.1). De esta manera, el lector obtiene una visión general de los compromisos que se encuentran al desarrollar esta clase de herramientas.

A continuación sección III.3, se da una explicación de los conceptos básicos que debe conocer un usuario final antes de usar el *Framework*. Seguidamente se explica con detalle a la estructura del *Framework* proporcionando una guía de uso a nivel medio (seccion III.5 en adelante). Estas secciones incluyen la información esencial para poder aplicar el *Framework* en nuevos dominios y problemas.

A dicho bloque le siguen dos casos de estudio (sección IV): El primero es un laberinto simple, que puede servir para aclarar la correspondencia entre los conceptos teóricos y un caso real. El otro ejemplo (sección V) es más complejo y está tomado del ámbito de la psicología. Considera el modelado de sistemas sociales usando Teoría de la Actividad. En él se proporciona un ejemplo de intérprete (sección V.2) que implementa la máquina de estados directamente a partir de una estructura de objetos, que se puede haber creado tanto a partir de un modelo expresado en un lenguaje específico de dominio como con una herramienta gráfica. Este segundo caso muestra el tipo de ramificación temporal mencionada anteriormente.

Los últimos bloques se dedican a explicar los detalles del desarrollo del proyecto (sección VI). Se comenta la organización llevada a cabo durante el curso (VI.1), seguida de un resumen de los requisitos del proyecto (VI.2). El bloque concluye con una documentación pormenorizada de la estructura del *Framework* (VI.3). En ella se incluyen los diagramas UML que documentan el sistema y la explicación del cometido de cada clase.

Finalmente se discuten algunas conclusiones y líneas de trabajo abiertas en este proyecto (VII.1 y VII.2). En esta última sección también se comenta la experiencia obtenida por los miembros del grupo en el proyecto (VII.3).

Un desarrollador de algoritmos de *Model Checking*, al igual que cualquier usuario, tendrá que elegir por que sección comenzar. Para decidir, debemos señalar que los primeros bloques son más básicos y se van concretando más detalles de implementación a medida que avanza en la lectura.

II. Estado del Arte.

II.1 Introducción

Los conceptos del *Model Checking* han sido desarrollados desde mediados de los ochenta, aunque las bases teóricas e ideas sobre las que se asienta el concepto aparecen en la década de los 60. Clarke [13] y Emerson [14] dan en trabajos recientes una versión personal de los inicios del *Model Checking*, en los cuales se remontan a más de 25 años desde la publicación de los documentos.

A lo largo de las últimas dos décadas han aparecido una cierta variedad de *frameworks* y herramientas para realizar *Model Checking* sobre dominios generales o específicos. En los últimos puntos de este apartado hablamos de algunas de las herramientas más relevantes y las estrategias que usan para enfrentarse a los problemas del *Model Checking.*

A continuación hablaremos del concepto de *Model Checking* y de sus implementaciones.

II.2 Model Checking

Los sistemas de *Model Checking* nacen como un método de verificación que intenta proporcionar una solución automática frente a los métodos manuales clásicos de verificación formal. Con ellos se buscaba dar otra aproximación a los problemas de verificación formal en sistemas concurrentes, ya que, como afirma Clarke [13], el proceso de verificación formal manual (como la comprobación de teoremas) puede ser realmente tedioso. Además el uso de otros métodos de verificación como el *debugging* y el testeo puede consumir más de la mitad del tiempo total invertido en cualquier tipo de desarrollo.

Los sistemas de verificación de modelos son métodos para verificar formalmente un sistema basado en una máquina de estados. La definicion clasica de *Model Checking* es*:*

> *Sea* ***M*** *una estructura de Kripke (por ejemplo: un grafo que guarda una serie de estados y sus transiciones)*
>
> *Sea* ***f*** *una formula en lógica temporal (que es la especificación de la propiedad a verificar)*
>
> *El sistema se encarga de encontrar todos los estados s pertenecientes a* ***M*** *que cumplan:* $M, s \models f$

Algunos de los pioneros en el ámbito del *Model Checking* fueron los ya citados Clarke y Emerson [15], cuyos trabajos han sido el punto de partida de los algoritmos de que aquí se incluyen. En la década de los ochenta presentaron un algoritmo de verificación de máquinas de estados finitas de complejidad lineal respecto al número de estados del grafo y respecto al tamaño de la fórmula.

En los primeros años de esta aproximación, (como ya mencionamos) nació como un método para verificar sistemas concurrentes. Se fue demostrando que para este nuevo concepto de verificación y para sistemas concurrentes, las lógicas temporales eran más

útiles, frente a la lógica de Hoare. Pnueli en el articulo [16] y Owicki y Lamper en el articulo [17]demostraron que las lógicas temporales son ideales para expresar conceptos tales como la exclusión mutua, expresar la ausencia de interbloqueos y expresar los problemas de inanición. Por ello, se empezó a desarrollar otras lógicas, siendo las más aceptadas hoy en día la CTL (*Computacional Tree Logic*) basada en árboles de computación, LTL (*Linear Time Logic*) y CTL*. De las dos primeras logicas se hablará en profundidad más adelante (apartados II.3 y II.4). Además la metodología Owick-Gries [18], bastante extendida en el mundo de la verificación formal, es un método de verificación manual y puede ser complicado de usar para verificar sistemas complejos. Por lo que se puede encontrar útil un método automático como es el *model checking*.

Por último, sólo cabe mencionar que en la actualidad, los sistemas de *Model Checking* han demostrado su fiabilidad y utilidad en áreas como el diseño de protocolos, circuitos y controladores hardware. Ejemplos bien conocidos fueron las comprobaciones de protocolos propuestos por el IEEE realizadas por los equipos de Dill y Clarke. En el caso de Dill y sus estudiantes de Standford, verificaron en 1992 el protocolo de coherencia de cache de la *IEEE Scalable Coherent Interface*. Para ello usaron Murphi [19], un *Model Checker* para grafos finitos, con búsqueda en anchura y profundidad desarrollado por ellos. El protocolo había sido ampliamente discutido, simulado e incluso implementado. A pesar de todo ello, el equipo encontró una gran cantidad de errores en el diseño y sutiles errores en la lógica que habían pasado desapercibido a los ojos de los ingenieros implicados en el proyecto.

II.3 Lógica CTL

Normalmente, los *Model Checkers* implementan optimizaciones para una clase de fórmulas en particular. En el proyecto descrito se han implementado algoritmos de *Model Checking* compatibles con la lógica CTL (*Computational-Tree Logic*). Ésta era adecuada para los primeros pasos que daba nuestro grupo de trabajo en el área del *Model Checking* al ser la más extendida para esta técnica.

Las fórmulas proposicionales CTL son una superclase de las fórmulas proposicionales y se pueden definir recursivamente de la siguiente manera:

- Una proposición p es una fórmula.
- Si f_0 y f_1 son fórmulas, entonces Not(f_0) , And(f_0,f_1), AX(f_0), EX(f_0),A(f_0 U f_1), E(f_0 U f_1) también son fórmulas.

Hay más fórmulas, pero el conjunto que aquí se expone forma una base, por lo que las demás fórmulas se pueden escribir en términos de éstas.

La semántica de estas fórmulas cobra sentido sobre una estructura M formada por un conjunto de estados (S), un conjunto de transiciones entre estados de S (R), y una función (P) que, para cada estado de S, devuelva las proposiciones atómicas que son ciertas en dicho estado. La relación binaria R ha de ser total, es decir, que para todo estado s_0 de S, tiene que existir una transición (s_0, s_1) en R, con s_1 un estado de S. Para evaluar una fórmula se necesita además un estado s perteneciente al conjunto de estados S de M. Para indicar que una fórmula f es cierta en un estado s de M usaremos la notación: $\mathbf{s \models f}$.

En las siguientes definiciones se usará el concepto *camino*, que se define como una secuencia de estados s_i, $0 \leq i \leq N$, tales que para todo i con $0 \leq i < N$, la transición $(s_i, s_{i+1}) \in R$ de M. Dado un estado s de M, y siendo f, f_0 y f_1 fórmulas CTL, la semántica de cada fórmula es la siguiente:

- $s \vDash p$, siendo p una proposición, es cierto si y sólo si la proposición p se cumple en el estado s, es decir, si p pertenece al conjunto de proposiciones que devuelve la función P de M.
- $s \vDash Not(f)$ es cierto ssi $s \vDash f$ es falso.
- $s \vDash And(f_0, f_1)$ es cierto ssi son ciertos $s \vDash f_0$ y $s \vDash f_1$.
- $s \vDash AX(f)$ (Always neXt) es cierto ssi todos los estados a los que transita s cumplen f, es decir, si para toda transición (s, s_i) perteneciente a R de M, $s_i \vDash f$.
- $s \vDash EX(f)$ (Eventually neXt) es cierto ssi algún estado al que transita s cumple f, es decir, si para alguna transición (s, s_i) perteneciente a R de M, $s_i \vDash f$.
- $s \vDash A(f_0 \ U \ f_1)$ (Always Until) es cierto ssi para todos los caminos que comienzan en s existe algún estado s_f que cumple f_1 y todos los estados s_i tales que $i < f$, s_i cumple f_0. Dicho de otra forma, todos los caminos que comienzan en s cumplen f_0 hasta que un estado cumple f_1.
- $s \vDash E(f_0 \ U \ f_1)$ (Eventually Until) es cierto ssi existe algún camino que comienza en s en el que existe un estado s_f que cumple f_1 y para todo estado s_i tal que $i < f$, s_i cumple f_0.

Para comprenderlo mejor, si un estado s cumple $E(f_0 \ U \ f_1)$, podemos decir que existe un camino que comienza en s que contiene un prefijo en el que el último estado cumple f_1 y todos los estados anteriores cumplen f_0. En el caso de $A(f_0 \ U \ f_1)$, todos los caminos que comienzan en s tienen esta propiedad.

Con estas definiciones, los algoritmos de *Model Checking* toman como entrada una estructura M, un estado s de M, y una fórmula f. Básicamente exploran todo el grafo necesario para determinar el valor de f en s. Como se explicará más adelante, hay algoritmos que además devuelven un subgrafo contraejemplo en el cual se pueda verificar que la fórmula f es falsa en s.

II.4 Lógica LTL

LTL (*Linear Time Logic*) es otra lógica temporal, usada para la especificación de propiedades en *Model Checking*. En el proyecto descrito en este documento no hemos incluido un algoritmo basado en fórmulas LTL, aunque como la arquitectura del *Framework* es flexible siempre puede ser añadido en futuras revisiones.

LTL permite expresar propiedades de *fairness* (“imparcialidad”) que no son expresables sobre CTL. LTL se centra en los caminos que hay en un modelo. Cuando se interpreta una formula LTL sobre un conjunto de caminos la fórmula tiene que ser cierta sobre todos los caminos que componen el modelo. Por lo tanto la diferencia sobre CTL es que el comportamiento del modelo no se interpreta como árboles sino como caminos

individuales. Por lo tanto, el modelo en LTL se define como un conjunto de estados y un conjunto de caminos.

Al igual que en CTL, las formulas proposicionales LTL son una superclase de las fórmulas proposicionales y se pueden definir recursivamente de la siguiente manera:

- Una proposición p es una fórmula.
- Si f_0 y f_1 son fórmulas, entonces Not(f_0) , And(f_0, f_1), X(f_0), G(f_0), F(f_0), (f_0 W f_1), (f_0 R f_1) e (f_0 U f_1) también son fórmulas.

En la siguientes definiciones supongamos que un conjunto de caminos $\mathcal{K}$ donde π es un camino, definimos las relación $\mathcal{K}, \pi \vDash \phi$ .Que significa que π perteneciente a $\mathcal{K}$ satisface la fórmula ϕ. Siendo además π^i un subcamino de π que empieza en el estado s_i. Siendo π una sucesión de estados $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n$. Supongamos también que Σ es un conjuntos de estados s_i y que una interpretación *I* es una secuencia de estados *s*0, *s*1, *s*2,… que se relacionan con proposiciones atómicas pertenecientes a Π (que es un conjunto de proposiciones).

- $\mathcal{K}, \pi \vDash p$, siendo p una proposición, es cierto si y sólo si la proposición p pertenece a una proposición de *I* a partir del estado inicial.
- $\mathcal{K}, \pi \vDash \neg f_0$, es cierto ssi $\mathcal{K}, \pi \vDash f_0$ es falso.
- $\mathcal{K}, \pi \vDash \mathrm{And}(f_0, f_1)$, es cierto sii son ciertos $\mathcal{K}, \pi \vDash f_0$ y $\mathcal{K}, \pi \vDash f_0$.
- $\mathcal{K}, \pi \vDash G(f_0)$, es cierto sii para todos los caminos π^i siendo $i \geq 0$ se tiene que cumplir $\mathcal{K}, \pi^i \vDash f_0$.
- $\mathcal{K}, \pi \vDash F(f_0)$, es cierto sii para algún camino π^i siendo $i \geq 0$ se cumple $\mathcal{K}, \pi^i \vDash f_0$
- $\mathcal{K}, \pi \vDash X(f_0)$, es cierto sii para un camino π, en π^1 se cumple f_0.
- $\mathcal{K}, \pi \vDash (f_0\ U\ f_1)$, es cierto sii existe $i \geq 0$ tal que se cumpla $\mathcal{K}, \pi^i \vDash f_1$ y para todo $j < i$ se cumple $\mathcal{K}, \pi^j \vDash f_0$.
- $\mathcal{K}, \pi \vDash (f_0\ R\ f_1)$, es cierto sii existe $i \geq 0$ tal que se cumpla $\mathcal{K}, \pi^i \vDash f_0$ y para todo $j < i$ se cumple $\mathcal{K}, \pi^j \vDash f_1$, o que se cumpla $\mathcal{K}, \pi^i \vDash f_1$ para todo $i \geq 0$.
- $\mathcal{K}, \pi \vDash (f_0\ W\ f_1)$, es cierto sii existe $i \geq 0$ tal que se cumpla $\mathcal{K}, \pi^i \vDash f_1$ y para todo $j < i$ se cumple $\mathcal{K}, \pi^j \vDash f_0$, o que se cumpla $\mathcal{K}, \pi^i \vDash f_0$, para todo $i \geq 0$.

Con esto podemos decir que un algoritmo de *Model Checking* basado en lógica LTL toma como entrada una representación del modelo formada por un conjunto de caminos y una fórmula LTL que representa la propiedad a probar.

II.5 Problemas a la hora de diseñar un algoritmo de *Model Checking*

Se puede decir que los principales problemas para diseñar un *Model Checker* son dos:

1. Especificación de los formalismos. Uno de los primeros retos a los que se enfrentaron los desarrolladores de la idea del *Model Checking*, fue la de encontrar formalismos para construir modelos con propiedades verificables. Es decir, formas de representar los problemas como máquinas de estados finitos. Con el tiempo, este primer obstáculo se ha solventado.

2. Algoritmos eficientes: éste es uno de los mayores retos a la hora de realizar una herramienta o algoritmo de *Model Checking*. Su principal causa en la explosión de estados. La verificación de una gran cantidad de modelos está limitada por la incapacidad espacial o temporal de computar sus estados en unos límites razonables. También hay que tener en cuenta que, ante un espacio de estados mayor, aumenta el número de estados que el algoritmo tiene que visitar potencialmente. Para solucionar el problema de la representación del modelo (en una máquina de estados) y el de la representación de las fórmulas, nace el *Model Checking* Simbólico. El *Model Checking* Simbólico [20] busca representar con fórmulas booleanas las estructuras de Kripke (presentes en las representaciones de los modelos como maquinas de estados) y representar dichas fórmulas como formas canónicas eficientes. Uno de estos acercamientos es el del OBDD (*Ordered Binary Decision Diagram*: Diagrama de Decisión Binario Ordenado, desde ahora simplemente BDD), el cual se usa en las implementaciones del *Model Checker* SMV.

II.6 Implementaciones actuales de Model Checkers.

En la actualidad existe un importante número de herramientas de *Model Checking*, algunas de ellas con una gran acogida en ciertos campos como la electrónica. Antes de repasar los *framework* actuales, merecen una mención especial dos de los primeros *frameworks* para *Model Checking*: EMC y Caesar.

EMC fue diseñado por Clarke, Emerson y Prasad. En el caso de Caesar el diseño corrió a cargo Queille y Sifakis. Siguiendo lo que sería el concepto actual de *model checking*, estas implementaciones se basaban en la idea de analizar sistemas que se podían representar como un grafo dirigido finito sobre el cual se probaban propiedades expresadas en la lógica temporal. En el caso de estos dos *frameworks* se usaba lógica CTL.

EMC tiene una optimización importante aprovechando que la máquina de estados es finita, consiguiendo un coste de NxM, siendo N el número de estados y M el número de subfórmulas que componen la propiedad a verificar. Para ello verifica la fórmula haciendo un recorrido en postorden y almacena el valor de cada subfórmula en cada estado. Para determinar el valor de cada subfórmula, hace un recorrido especial del grafo. Por ejemplo, para las subfórmulas proposicionales ha de recorrer todo el grafo, pero para una subfórmula del tipo $E(f_1 \ U \ f_2)$, empieza por los estados donde se cumple f_2 y recorre las transiciones a la inversa, marcando que la fórmula es cierta en todos los

estados que cumplen f_1. Como se ve, este algoritmo es muy interesante, pero no es aplicable a grafos infinitos y tiene como inconveniente que no está dirigido a la obtención de contraejemplos.

Actualmente los acercamientos para resolver el problema del *Model Checking* son variados:

- Se puede usar algoritmos simbólicos: que buscan no tener que construir el grafo de estados explícitamente y clasificar más eficientemente los estados.
- Algoritmos guiados por contraejemplos, destinados principalmente a obtener un contraejemplo sólido que pruebe lo erróneo del modelo para la propiedad.
- Algoritmos con propósitos de realizar reducciones de orden parcial.

II.6.1.1SMV FrameWork

SMV Fue la primera herramienta en representar las fórmulas como Diagramas de Decisión Binarios (BDD). La potencia de SMV se basa en el uso de representaciones simbólicas [20] del espacio de estados con BDD. Los BDD Ordenados (OBDD) son representaciones canónicas de las funciones. Los BDD Ordenados se obtiene al aplicar las siguientes reglas a una formula booleana:

- Fijar el orden de las variables que mejor convenga para reducir el número de nodos en el diagrama.
- Eliminar las redundancias, presentes en el diagrama. Para eliminar estas redundancias se iteran los pasos siguientes, hasta que no se pueda eliminar ninguna redundancia:
    - Quitar los nodos terminales repetidos, dejando en el proceso un solo nodo 0 y un solo nodo 1.
    - Quitar los nodos no terminales repetidos, coinciden la variable y los hijos.
    - Quitar los “test” redundantes (es decir coinciden los dos hijos del nodo).

Supongamos el ejemplo de la Ilustración 1, figura 1 es un diagrama de decisión binario para la función $(a1 \leftrightarrow b1) \wedge (a2 \leftrightarrow b2)$. Las flechas discontinuas indican la transición cuando la variable del nodo vale *false* al evaluarse, y la flecha continua cuando vale *true*. Los nodos finales (hojas) valen 1 si es *true* y 0 si es *false*. En la figura 2 es el mismo diagrama que en la figura 1 pero ordenado. Además se le ha eliminado las redundancias y repeticiones. Por lo tanto esta representación posee menos nodos y es más eficiente.

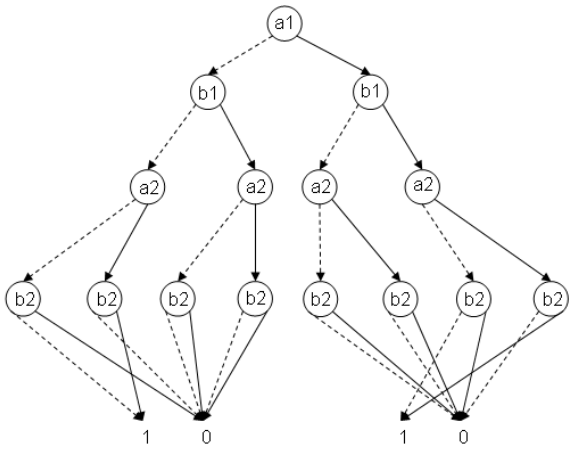


Figura 1

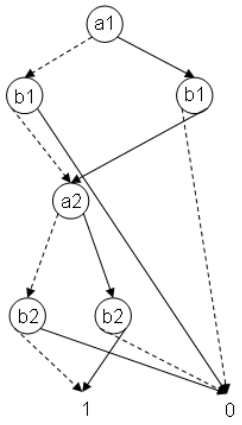


Figura 2

Ilustración 1 Diagrama de Decisión Binario Ordenado

Los OBDD producen una mejora en la eficiencia de los algoritmos de *model checking*. Permiten el tratamiento de modelos con espacios de estados mucho mayores que otras representaciones alternativas presentes en otras herramientas de la época de su desarrollo. La razón [21] es que una vez fijado el orden de las variables para que favorezca un menor número de estados en la reducción y una vez eliminadas las redundancias obtenemos una representación canónica de una fórmula.

El uso de OBDD en SMV se centra en 2 aspectos: representar las fórmulas para que el algoritmo puede manejarlas de forma eficiente; representar las estructuras de Kripke para definirlas como una formula booleana. Por lo tanto, se tiene una representación simbólica (o abstracta) del espacio de estados, y una representación canónica y eficiente de las fórmulas. Con estas mejoras se consigue reducir la explosión de estados, al representar el espacio de estados como un conjunto de OBDD. Esto supone una mejora de rendimiento importante con respecto a otros métodos que expresan el espacio de estados de forma explícita con un grafo, ya que al reducir el coste en espacio permite almacenar en la memoria un mayor número de estados. Además también se mejora el rendimiento en tiempo de ejecución al ser una representación con menos nodos explícitos que tratar, y que posee propiedades que permiten realizar ciertas operaciones en un tiempo de ejecución menor.

Hay varias variantes e implementaciones de SMV aparte de la perteneciente a la Universidad de Carnegie Mellon (CMU). Una de las variantes más robusta y escalables es NuSMV, que incorpora mejoras en la implementación y las herramientas destinadas a la construcción del modelo, visualización de contra-ejemplo,

II.6.1.2 SPIN Framework

SPIN: Uno de los puntos fuertes de SPIN es el hecho de que las fórmulas de las propiedades a comprobar son definidas en la lógica temporal LTL y que es un *framework* orientado a la verificación de programas concurrentes. Se basa fundamentalmente en una serie de resultados matemáticos que a partir de una fórmula LTL, permiten construir autómatas reconocedores de cómputos que cumplan dicha fórmula [22]. Estos autómatas reconocen palabras infinitas, tomando como letras del alfabeto las subfórmulas que se hacen ciertas tras la ejecución de cada instrucción del

programa concurrente modelo. La secuencia de instrucciones del programa se traduce a una secuencia de subfórmulas que son ciertas en cada uno de los estados. Al autómata se le introduce esta secuencia de subfórmulas como palabra de entrada. Si se pasa por un estado de aceptación significa que el cómputo cumple la fórmula. Este recurso se puede utilizar para verificar una fórmula deseable para el sistema, o bien para detectar que el sistema cumple una fórmula no deseada. Para este último caso, se busca que la intersección entre los cómputos posibles del sistema y los cómputos aceptados por el autómata sea el conjunto vacío, y dicho tamaño se puede verificar como caso mejor en tiempo constante [7].

SPIN usa este resultado y traduce "al vuelo" las fórmulas en autómatas de Büchi, con lo que se permiten dos casos interesantes para la reducción de la cantidad de estados en memoria y la eficiencia.

- Al ser un autómata, permite que se traduzca a código C, generando así el código de un programa que compruebe una fórmula concreta para un caso en concreto. Con esto se pueden aplicar optimizaciones propias del sistema sobre el que correrá el programa y añadir porciones de código C propio sobre el ya generado.
- Permite realizar reducciones de orden parcial, así como otras técnicas que ayudan a controlar la explosión de estados durante la comprobación del modelo.

II.6.1.3 Java PathFinder

Java PathFinder es un sistema reciente cuya intención es proporcionar un chequeo para aplicaciones ya construidas en Java [23]. Principalmente está pensado para sistemas concurrentes y orientado a encontrar interbloqueos. Además, también avisa de otros errores de la programación concurrente en Java que son complicados de detectar con las herramientas de *debug* y *testing* tradicionales. El sistema funciona como un *model checker* en lugar de cómo las tradicionales herramientas de depuración en el sentido de que no muestra el error en sí, sino el camino por el que se llega a producir (secuencia de estados del programa).

Aunque los *Model Checkers* han recibido ideas frescas desde sus inicios y han evolucionado notablemente, actualmente las herramientas existentes son poco populares. Estudiantes o desarrolladores que quisieran hacer uso de estas herramientas se pueden encontrar con problemas importantes a la hora de encontrar entornos que se adecuen a sus necesidades.

III. Descripción del proyecto

III.1 Introducción. Motivación y Objetivos

En la actualidad existe una cierta variedad de sistemas capaces de realizar *Model Checking*, algunos muy difundidos. A pesar de ello se presentan algunos inconvenientes en su utilización.

Uno de estos inconvenientes reside en la representación del sistema que se desea verificar. La mayoría de las herramientas de *Model Checking* usan un lenguaje propio para definir los modelos, como es el caso de SPIN y su lenguaje PROMELA, que otorga la posibilidad de realizar optimizaciones sobre el algoritmo de verificación. Esto obliga al usuario a aprender el lenguaje del *Model Checker*, traducir su modelo de un lenguaje de modelado propio a dicho lenguaje y, en el caso de haber desarrollado un prototipo, imposibilita la reutilización de dicho prototipo en la verificación del mismo. Esto puede producir incompatibilidades: es posible que el lenguaje no se adapte a las necesidades del modelo del sistema, se tenga que simplificar el modelo a verificar o incluso que al final el modelo representado en el lenguaje propio del *Model Checker* tenga un peso computacional mayor que el original. Para estos casos interesa tener un acercamiento más flexible, que posibilite la utilización de un lenguaje propio o el uso directo del desarrollo como motor del modelo.

Otro tema importante es poder estudiar cómodamente dónde falla el modelo cuando no cumple una propiedad. Muchos sistemas de verificación incorporan herramientas para visualizar el contraejemplo devuelto por el *Model Checker*. Sin embargo, éstas son genéricas y poco ilustrativas. En ciertos dominios de trabajo es interesante tener la posibilidad de implementar fácilmente una representación del contraejemplo, más apropiada para el dominio en cuestión.

Por último, cabe la posibilidad de que un algoritmo de *Model Checking* sea poco eficiente en tiempo o espacio al verificar un modelo en particular y sea conveniente realizar la verificación con un algoritmo diferente, o incluso aprovechar lo mejor de cada estrategia utilizando un algoritmo mixto. En este sentido, los *Model Checkers* actuales suelen estar especializados cada uno en una estrategia de optimización, y en caso de que se haya utilizado una herramienta determinada y se decida probar otra herramienta, el usuario estaría obligado a traducir el modelo al nuevo lenguaje con todo el esfuerzo y posibles inconvenientes que ello acarrea. Lo ideal es usar un único lenguaje fuente que sirva de entrada a una colección de algoritmos entre los que poder elegir y que dicha colección sea fácilmente ampliable.

Por ello, el *Framework* aquí presentado está pensado para ser un acercamiento muy generalizado y flexible. El hecho de que sea un *framework* provee más flexibilidad que la de una aplicación ya construida con sus interfaces graficas (o de consola) que tenga un método propio para definir el modelo o sistema a verificar. El *framework* no proporciona un lenguaje de modelado, sino la posibilidad de crear lenguajes propios para cada dominio, y programar intérpretes[I] óptimos que sirvan para el dominio

[I] Consideramos intérprete como el elemento que traduce un modelo a una máquina de estados, de tal forma que proporciona los posibles estados alcanzables a partir de un estado proporcionado. En términos lingüísticos, codifica la semántica operacional del lenguaje específico en el que se expresan los modelos.

deseado, lo que abre la posibilidad de reutilizar un desarrollo ya implementado y adaptarlo para su uso como intérprete.

El *framework* incluye un algoritmo de *Model Checking* por defecto. Cambiar el algoritmo a utilizar es sencillo con las interfaces aquí definidas. También se puede usar un algoritmo mixto a partir de varios verificadores, usando un objeto conector proporcionado por el *framework* y eligiendo, para cada tipo de fórmula, el verificador que la comprobará. Esto es posible porque los algoritmos de verificación tratan a cada tipo de fórmula en particular. Además, si se considera oportuno, se pueden implementar nuevos algoritmos y añadirlos a esta colección. Como caso particular, en algunos campos donde se aplica la verificación de modelos, no conviene utilizar un algoritmo eficiente que explore todo el espacio de estados, sino que comience a expandir a partir de los estados iníciales, ya que el espacio de estados es demasiado grande (o infinito) y las fórmulas a verificar, pequeñas. Bastante parte del trabajo se ha centrado en optimizar este algoritmo preparado para recibir máquinas de estados infinitas, ya que el gasto en memoria es importante.

La actual implementación que premia la flexibilidad, no aporta algunas características eficientes, como es el uso de BDD (*Binary Decision Tree*). Los BDD proporcionan una forma de ahorro de tiempo de cómputo para encontrar el resultado a una propiedad, pero pueden sesgar el contraejemplo obtenido. Aunque el uso de BDD en el *framework* siempre es posible, si se realiza una implementación del motor de chequeo e intérprete acorde.

En éste *framework* se apuesta también por la facilidad de uso, lo que permite una de las características más interesantes: que el usuario no necesite saber de *Model Checking* para usarlo. Basta con que represente su problema particular usando las interfaces que se proveen y, opcionalmente, que elija una representación gráfica o textual de sus estados para estudiar los contraejemplos. Un lanzador ejecuta el algoritmo de *Model Checking* que viene por defecto y visualiza el contraejemplo en un navegador. El usuario puede entonces navegar a través del sub-grafo que delata el lugar exacto del fallo en el modelo que creó. Si se quiere usar un navegador más personalizado, la arquitectura está preparada para ser extensible y escalable, ya que se pueden crear nuevos navegadores y tener varios visualizadores a la vez. Las herramientas de *Model Checking* actuales no se centran en estos aspectos, ofreciendo visualizaciones genéricas pero tediosas de estudiar, ya que no tienen información de qué es lo que se está representando.

Con todas estas mejoras se presenta un *framework* generalizado, flexible, escalable y fácil de usar que pretende facilitar el trabajo tanto a usuarios finales como a futuros desarrolladores de algoritmos de *Model Checking*.

III.2 Implementación de los objetivos.

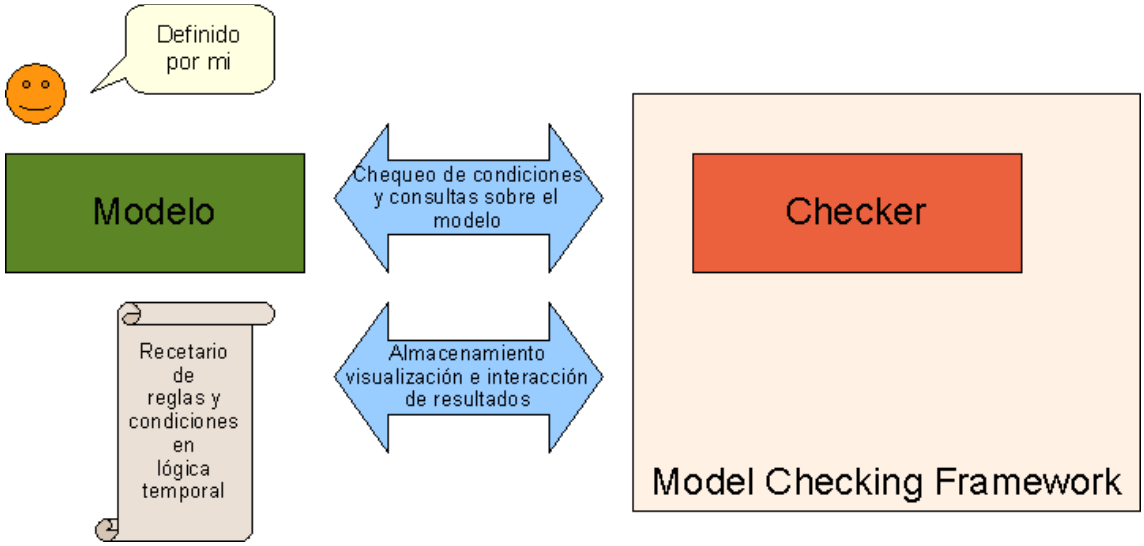


Ilustración 2 En Abstracto

Este *Framework* consiste en una API que proporciona herramientas para verificar modelos basados en máquinas de estados infinitos. Un ejemplo fácil de visualizar sería tomar un juego de rol en el cual hay un *storyboard*, reglas, limitaciones... ¿No existe ningún momento en el cuál se produzca un estado de *Dead-lock*[I]? Si el juego tiene una extensión elevada, con muchas tramas, alternativas, no es un tema que se resuelva trivialmente sin el uso de alguna herramienta automatizada. Estas situaciones no deseadas siempre pueden darse, incluso después de diseñar las misiones y línea argumental con especial cuidado. Su aparición reduce la calidad del producto y puede afectar a sus ventas. Por lo tanto se presenta la necesidad de usar una herramienta de verificación cuyos costes de adquisición y tiempos de desarrollo y uso sean asequibles, y a ser posible que permita reutilizar la experiencia previa. Ahora usemos esta idea para ilustrar un modo de uso de nuestro *Framework* de *Model Checking*:

Una importante compañía desarrolladora de videojuegos está realizando el maquetado de un mundo de rol. Se desea realizar operaciones de chequeo sobre la actividad permitida al jugador a lo largo de este escenario, para lo cual se han de verificar ciertas propiedades del diseño de misiones y línea argumental. La cantidad de información y de detalle es enorme, ya que es un juego de cierta complejidad. Se decide entonces que el mejor modo es usar una herramienta de *Model Checking*.

Si queremos aplicar *Model Checking* hay que tener en cuenta que en ciertos ámbitos o problemas, el computo puede ser excesivo incluso para máquinas bien preparadas. Tal vez se desee evitar estas situaciones, por no ser necesarias para la representación del modelo. Se presenta la necesidad de simplificar y abstraer el sistema de reglas y regiones. Con esto se consigue empezar a realizar un modelo de lo que representa el

[I] En el ámbito del juego consideramos *dead-lock* a un estado en el cual se necesitan acciones u objetos (requisitos) para avanzar en el argumento, pero desde dicho estado no podemos realizar o conseguir los dichos requisitos, es decir, alcanzar el estado en el que se satisfacen. Por ejemplo, podemos: llegar a una puerta en la que se requiere una llave, pero no podemos conseguir la llave porque está en una cueva cuya entrada se ha derrumbado y ya nunca se puede pasar dentro.

mundo del juego.

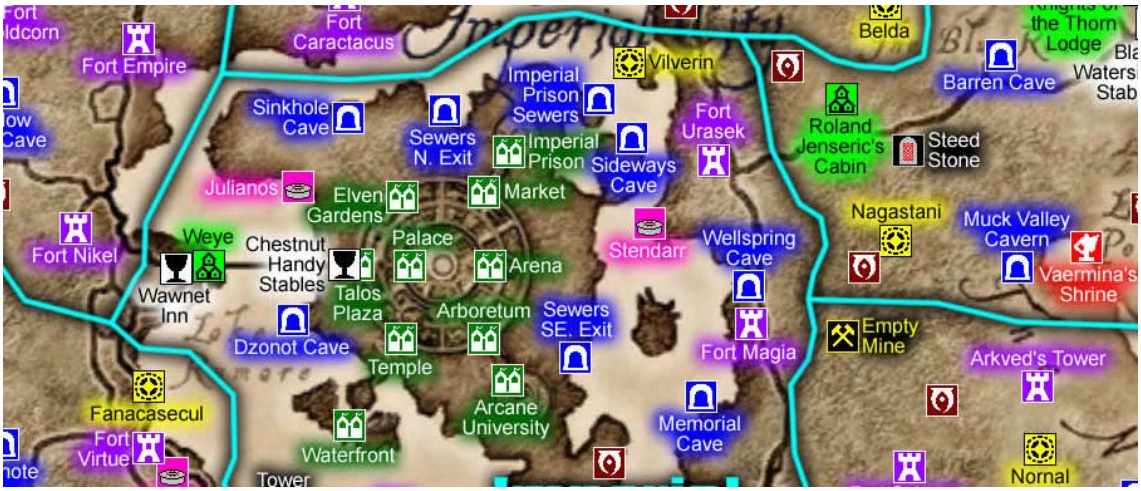


Ilustración 3 Mapa de juego de rol

Centrándose en el aspecto de recorrido por el juego, éste se puede dividir en zonas las cuales calificaremos de visitables o no alcanzables por el jugador. La accesibilidad a una zona se define en el modelo, pues puede requerir de un objeto clave del juego, o un nivel específico del jugador.

Las situaciones a comprobar se especifican como ecuaciones lógicas basándose en proposiciones[I]. Las proposiciones se corresponden a diferentes aspectos del juego que interesan al programador para un momento dado del juego. Así, una zona se vuelve visitable si se presentan ciertas características, como por ejemplo, que el jugador disponga de la llave de acceso o forme parte del grupo de esa zona.

Otro factor a tener en cuenta es las acciones de que dispone el jugador. Así se debe tener en cuenta que aunque una zona se encuentre en situación de visitable, si las acciones del jugador no le permiten moverse hasta esa zona, no se llegará nunca a probar que puede ser visitada. Por ejemplo, se puede tener en cuenta que se desea que el personaje no pueda cruzar a una zona delimitada por un rio sin tener una barca. Si aplicamos *Model Checking* a esta situación podríamos obtener varios resultados. La proposición de ejemplo para esta situación es “Solo se puede acceder a la zona B visitando la zona A (el rio)?”. Suponiendo que las acciones del jugador son navegar y caminar, que la zona de destino es visitable, y sólo se puede cruzar el río en barca, el *Model Checker* deduciría si existe un resultado positivo o negativo para esta proposición.

La obtención del resultado y de la información del mismo depende del programador. En el caso anterior, si la respuesta fuera negativa aún siendo visitable, sería conveniente llevar una traza para diagnosticar el motivo. El presente *Framework* incorpora un sistema para visualizar el contraejemplo o ejemplo de un resultado, donde el programador puede definir la trazabilidad de los datos que le interesan.

[I] En este caso diferenciamos proposiciones de fórmulas. Las proposiciones se refieren a propiedades sobre los atributos de un estado, sin tener en cuenta su evolución en el tiempo.

III.3 Estructura de alto nivel

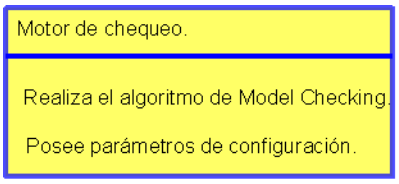
El Framework desarrollado en este proyecto está construido a partir de 4 conceptos, que resumimos a continuación:

- ***Modelo***: es una abstracción de lo que se desea verificar.
- ***Intérprete:*** interpreta el modelo y tiene la capacidad de indicar cuáles son las transiciones posibles para un estado.
- ***Estado:*** está ligado al intérprete principalmente. Guarda la información necesaria para diferenciar un estado de otro y poder deducir los posibles estados a los que puede transitar.
- ***Formula y Proposición:*** juntos definen una propiedad a comprobar sobre el modelo. Una fórmula constituye la definición de una propiedad a comprobar en lógica temporal. Actualmente en el *Framework* los motores de chequeo implementados usan la lógica CTL. Una proposición se define (e implementa) junto con el intérprete, e indica una propiedad concreta a comprobar sobre un estado.
- ***Motor de chequeo:*** es el encargado de realizar la operación de verificación. Puede funcionar con diferentes lógicas temporales y usar diferentes algoritmos y estrategias de *Model Checking*.

A continuación se detallan en profundidad los conceptos señalados anteriormente.

III.3.1 Motor de chequeo:

Para llevar a cabo la acción de *Model Checking* se precisa de una unidad que realice esta actividad. La vamos a denominar *Motor de chequeo*. Esta unidad opera con uno o varios algoritmos de *Model Checking* y proporciona opciones de configuración del mismo. Estos aspectos configurables determinan propiedades de la búsqueda que se aplican a lo largo del proceso de *Model Checking*.

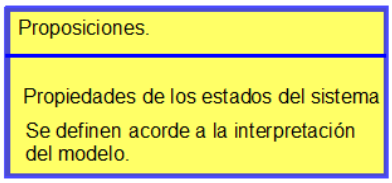


Debemos destacar que el motor de chequeo siempre finaliza con uno de los tres posibles resultados: (i) La propiedad a comprobar es cierta, proporcionando un ejemplo; (ii) La propiedad a comprobar es falsa, proporcionando un contra-ejemplo; y (iii) Indeterminación por alguna fuente de error, normalmente porque un modelo es formalizado como una máquina de estados infinita en la que no se ha conseguido un resultado concluyente.

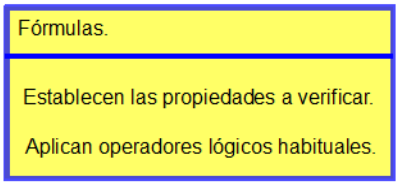
III.3.2 Formula y Proposición

La aplicación de *Model Cheking* está implícitamente ligada a la verificación de una propiedad que debe cumplirse en uno o varios estados del sistema puesto a prueba. Este tipo propiedades se suelen definir con lógicas temporales. Para especificar estas propiedades se pensó en dos módulos clave: las proposiciones y las fórmulas.

Las proposiciones son definiciones de propiedades de los estados del modelo, que se consideran importantes para el sistema que se va a someter a verificación y que es representado por el modelo. Desde una perspectiva lógica, las proposiciones serán predicados que evalúan propiedades de un estado en la verificación.



El concepto de fórmula se deduce por su finalidad: componer expresiones lógicas complejas a partir de las proposiciones. Una proposición es el concepto más básico de una fórmula. Las fórmulas se componen de operadores de lógica temporal y de otras fórmulas. Las fórmulas junto con las proposiciones son la traducción de las propiedades que se quieren verificar en lógica temporal. Cabe indicar que el resultado obtenido puede variar según la forma de expresar las propiedades que se desean verificar, ya que pueden existir dos formulas diferentes y no equivalentes, pero con significados homólogos.



Las fórmulas son elementos que definen una propiedad del sistema, como explicábamos antes. Estas son proporcionadas al motor de chequeo, para que éste obtenga una contestación. Como los motores de chequeo implementados actualmente usan una lógica temporal CTL, para definir las propiedades hay que proporcionar una fórmula CTL y un estado (o conjunto de estados) del que se parte para comprobar la propiedad.

A continuación se presenta un esquema de relación entre los módulos presentados hasta el momento.

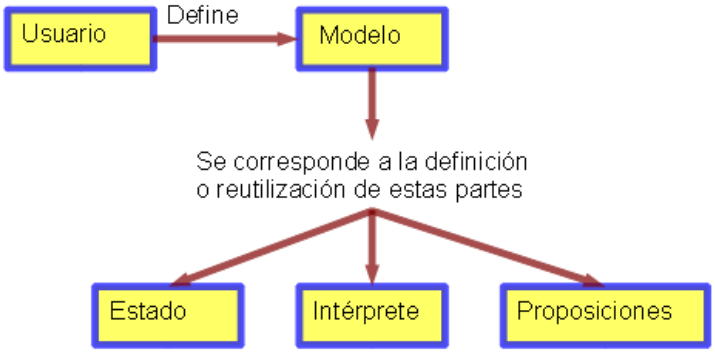
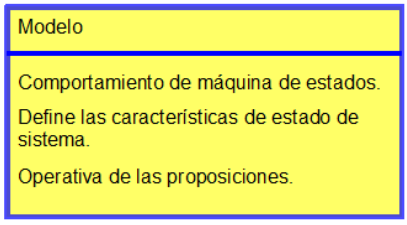


Ilustración 4 Correspondencia del Modelo

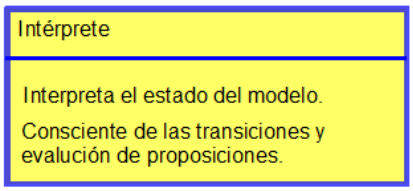
La verificación de propiedades se realiza sobre el modelo a verificar. Cuando se hace referencia a un modelo a verificar hay que tener presentes las dificultades que conlleva realizar *Model Checking* sobre él. Los problemas de complejidad que se encaran en este proceso a veces suponen un verdadero reto al desarrollador que usa estas herramientas. No sólo se tiene que tener en cuenta la explosión de estados, para conseguir eficiencia en la obtención de resultados, sino también la complejidad de formalizar en fórmulas los conceptos de las propiedades.

III.3.3 Modelo e Intérprete

El modelo debe ser una máquina de estados, a poder ser con un número finito de estados. El modelo es la entidad abstracta que proporciona las reglas de transición, o lo que es lo mismo, la semántica de las transiciones entre estados. El intérprete usa el modelo para definir las transiciones entre los estados representativos del sistema que se va a verificar. Los estados están ligados al intérprete, ya que es éste el que realiza las transiciones. El motor de chequeo realiza una exploración y verificación de las propiedades en los diferentes estados posibles alcanzables. Una propiedad está definida por una fórmula temporal que contiene un número indeterminado de proposiciones.



Para poder asimilar este modelo con el motor de chequeo, se va a necesitar un módulo que adapte la información del modelo. Este módulo se denomina *Intérprete*. El intérprete, se encarga de traducir el modelo a las necesidades del motor de chequeo. En este caso, el motor de chequeo necesita estados, y conocer las posibles transiciones desde un estado en concreto. El intérprete se encarga de reconocer un estado y aportar las posibles transiciones desde dicho estado. Con esto se consigue un elemento intermedio entre el modelo a verificar y el motor que da la posibilidad de definir la interpretación de un modelo, sin estar ligado a un lenguaje en concreto de la herramienta. Por lo tanto, el intérprete usa el modelo para definir las transiciones entre los estados representativos del sistema que se va a verificar. Los estados están ligados al intérprete, ya que es éste el que realiza las transiciones.



III.3.4 Interacción intérprete y motor.

La verificación de una propiedad sobre un modelo a lo largo del proceso de chequeo se consigue gracias a la colaboración del motor de chequeo y el intérprete. El intérprete proporciona información sobres las transiciones que se pueden hacer desde un estado. Esta es información necesaria para el motor de *Model Checking*, ya que con ello se puede expandir el grafo[I] sobre el que se realiza la búsqueda.

El motor de chequeo (en este apartado nos referiremos a él simplemente como el motor) para poder operar recibe como información: la fórmula a verificar, el intérprete del modelo y un estado inicial. Desde el motor de chequeo se recorre la fórmula desde las sub-fórmulas más complejas a las más sencillas. Con este recorrido se dirige la búsqueda entre los estados del modelo, y también se procede a la extracción del ejemplo o contraejemplo necesario.

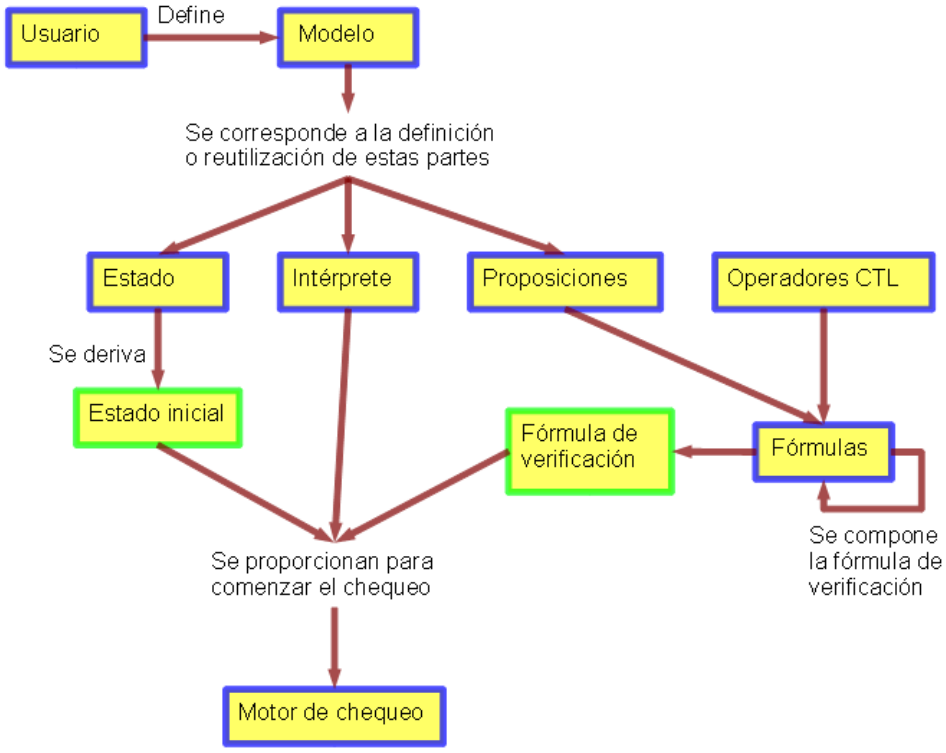


Ilustración 5 Interacción Modelo - Motor de chequeo

El motor realiza consultas constantemente al intérprete mientras recorre los estados guiado por la formula CTL introducida. Según la operación lógica que se evalúa de la

[I] Representamos el modelo como una máquina de estados, y la máquina de estados como un grafo

fórmula, se realiza una operación de búsqueda u otra. El ejemplo o contraejemplo obtenido dependerá de los operadores lógicos usados para definir la propiedad.

El resultado de la evaluación de la proposición no consiste únicamente en una respuesta positiva o negativa con respecto a la propiedad a evaluar. También se obtiene del seguimiento de las transformaciones (cambios de estado) un conjunto de estados y transiciones que se usan para poder demostrar el resultado. Este conjunto consiste en una estructura arborescente con las transiciones y estados por los que se ha pasado al verificar una propiedad. En el caso de que la propiedad se cumpla, se considera que el conjunto es un ejemplo. Para el caso en que no se cumple la propiedad, se considera que el conjunto es un contra-ejemplo.

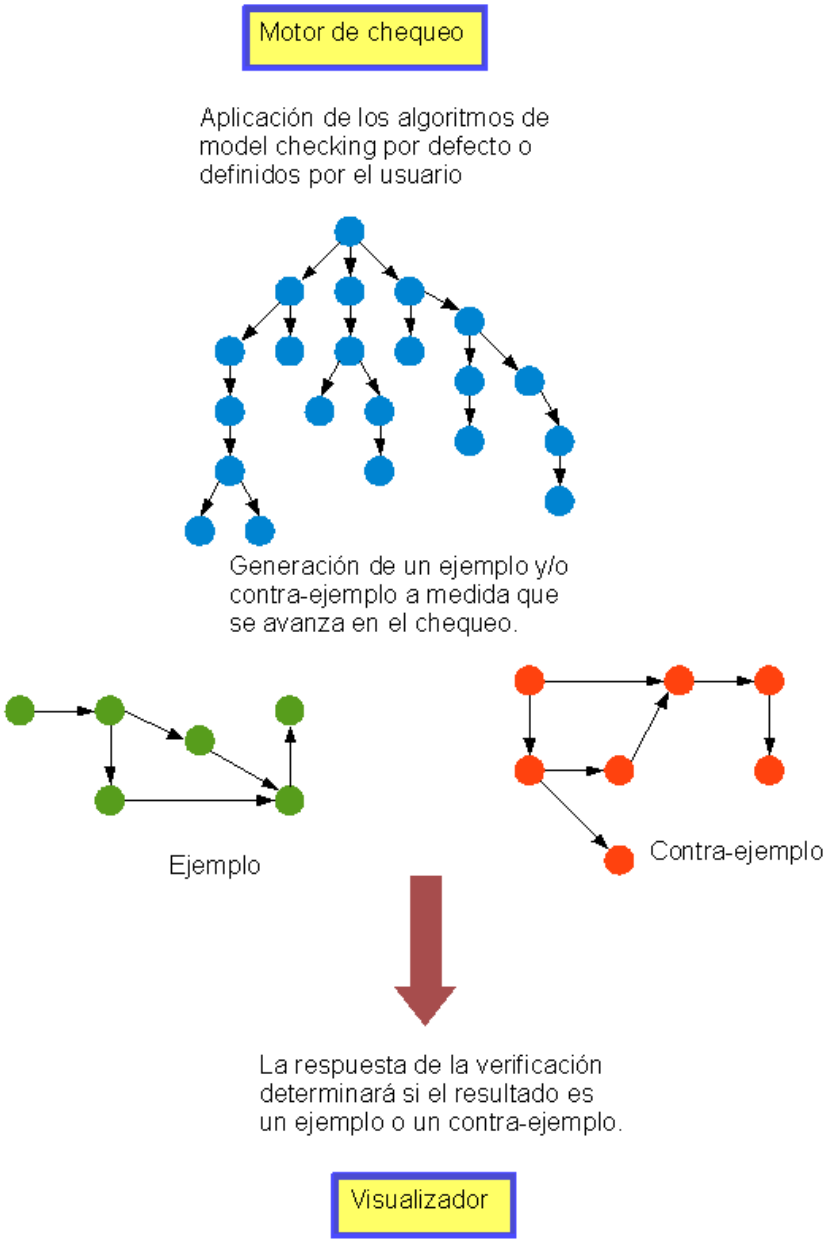


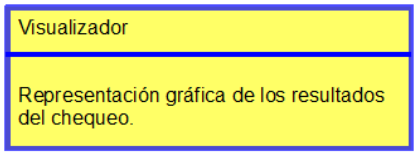
Ilustración 6 Esquema de procesamiento del Motor de chequeo

III.3.5 Visualización

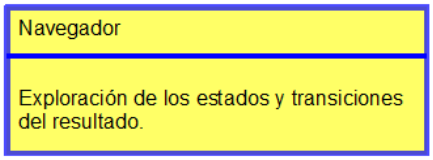
Anteriormente se hablaba de la forma en la que se obtienen los resultados en el Framework. Pero este resultado en bruto no sirve de mucho al desarrollador. Por lo tanto es una opción interesante el que se le proporcione un sistema sencillo para visualizar y explorar este resultado.

En el *Framework* está incluido un sistema para la visualización de estos resultados. Consiste en dos módulos el Visualizador y el Navegador.

Visualizador: Es un módulo que se encarga exclusivamente de la interacción con el usuario a través de una pantalla grafica. Muestra y recoge datos para el *Navegador*. Este componente se puede adaptar a cualquier modelo: sólo hay que definir un objeto *Drawer* específico al ámbito de estudio. El objeto *Drawer* sólo se encarga de componer una visualización representativa del estado proporcionado. De los demás aspectos se encarga el modulo *Visualizador*.



Navegador: Este módulo nace por la necesidad de poder procesar el resultado de forma sencilla, flexible y compatible con diversos métodos. El navegador se encarga de navegar a través de un ejemplo o contraejemplo recuperado después de la verificación. Dada la estructura arborescente del resultado, el navegador permite trabajar con cada uno sus elementos, consiguiendo una navegación paso a paso sobre la verificación a la que se ha sometido el modelo.



III.4 Pasos recomendables para el uso del Framework.

La metodología de trabajo propuesta se realiza a través de los pasos mostrados en el diagrama de la Ilustración 6:

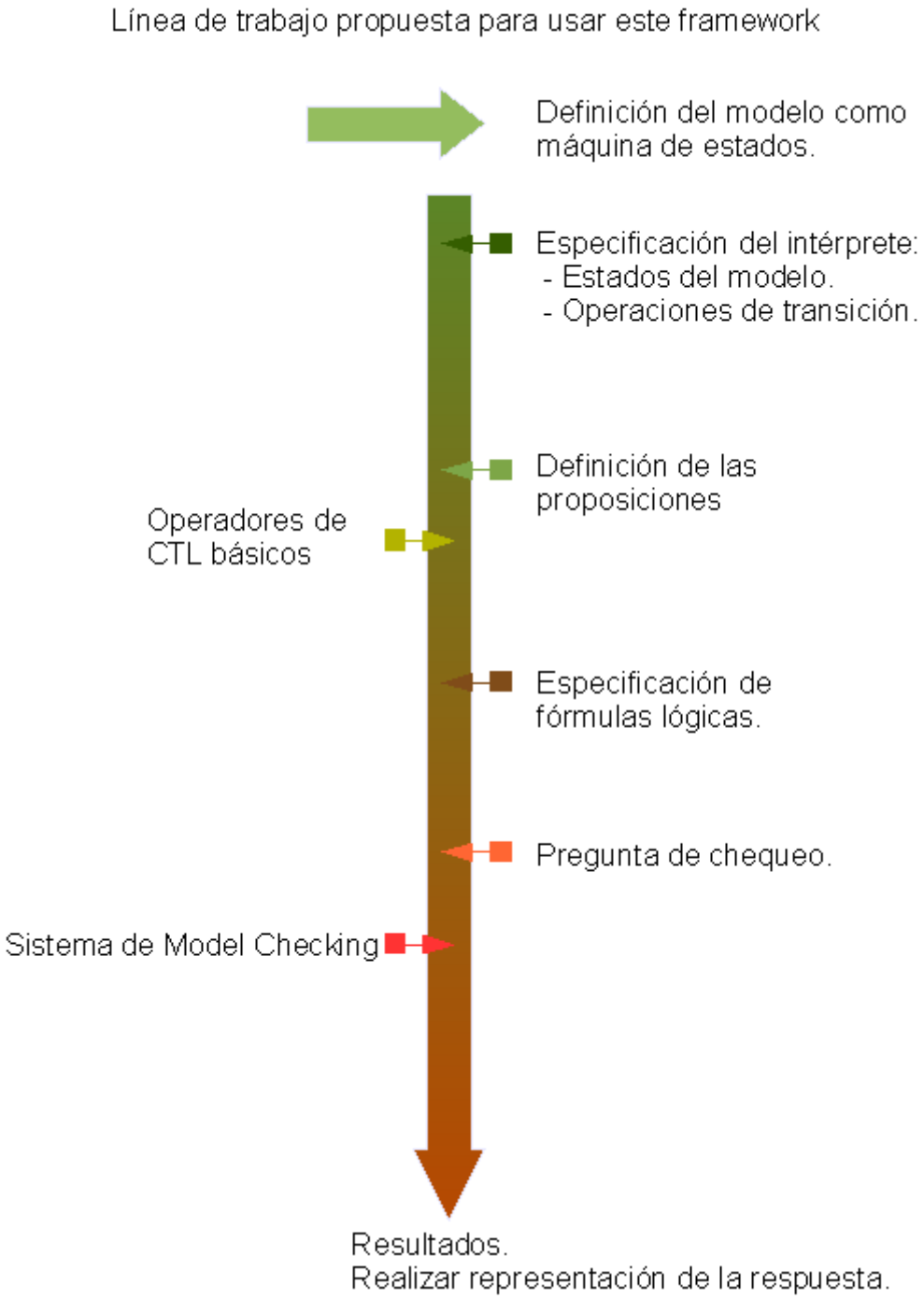


Ilustración 7 Diagrama de desarrollo

El primer paso a seguir es concebir un intérprete para el modelo que se desea verificar. Cuando se piensa en el intérprete también se piensa en la representación de los estados, ya que están relacionados (estados-intérprete). También se puede reutilizar un desarrollo ya creado o disponible.

Después de esta fase, se procede a definir las fórmulas en lógica temporal y las proposiciones que se necesitan para definir las propiedades a verificar.

Con estos dos pasos, se obtienen los elementos básicos y necesarios para hacer funcionar el *Model Checker*.

Por último se escoge una estrategia, o se usa una propia, para configurar el motor de chequeo. Luego se pueden usar las facilidades internas del *Framework* para lanzar la verificación.

Al final de la verificación se puede implementar una visualización usando las facilidades propuestas o conformarse con la respuesta básica del *Model Checker*, que puede ser cierto, falso, o indeterminado.

III.5 Análisis a nivel medio

III.5.1 Propósito:

El *Framework* presentado tiene las siguientes características relevantes:

- Es una herramienta destinada a la verificación. Persigue su amplia utilización no solamente por programadores y expertos en lógica (en concreto en la lógica usada para definir las propiedades), sino también que la definición del modelo y/o intérprete pueda ser realizada por personas con conocimientos básicos de *Model Checking*.
- Desarrollada en lenguaje Java, ampliamente extendido.
- Ampliable en todos los puntos de la arquitectura. Es un desarrollo abierto y posee una documentación detallada de las clases usadas. El usuario que quiera adaptar o mejorar el *Framework* puede centrarse (como recomendación) en los siguientes aspectos:
    - El modelo e intérprete. El usuario puede definir un intérprete que use un lenguaje específico para definir un modelo. El usuario también puede crear un intérprete orientado a un modelo en concreto.
    - La representación. El usuario puede mejorar el modulo de visualización, o crear uno nuevo adaptado a sus necesidades.
    - Motor de Chequeo. Para ampliar o mejorar los algoritmos y usar nuevas estrategias.

III.5.2 ¿Por qué esta arquitectura?

El *Framework* esta desarrollado íntegramente en Java. Las razones de usar este lenguaje son claras. Java es un lenguaje Orientado a Objetos, lo que permite una gran flexibilidad. Gracias a esto presenta ventajas la hora de reutilizar el código, así como para definir la arquitectura. Es un lenguaje ampliamente extendido y soportado. Estas razones hacen que sea una buena elección para nosotros (los desarrolladores de la aplicación), ya que en un principio éramos ajenos a las técnicas de *Model Checking*, lo que implica que el desarrollo estará plagado de pruebas y cambios. Todas estas pruebas y cambios en la estructura, diseño e implementación del desarrollo del *Framework*, son facilitadas por este lenguaje y su máquina virtual. También ofrece la posibilidad de usar código nativo: esto es binarios y librerías escritos en otros lenguajes tales como C/C++, Python o pertenecientes a plataformas tales como .NET. Java cuenta con un gran número de intérpretes de otros lenguajes tales como CLIPS, Prolog,... accesibles desde aplicaciones en Java. Todo esto, unido al hecho de que existe un gran repertorio de herramientas, librerías y *Frameworks* de Ingeniería, Inteligencia Artificial,... escritos para Java, hacen del lenguaje una buena elección para este desarrollo.

El otro punto importante de elección en el Framework es su arquitectura. A continuación se discute brevemente dicha arquitectura y las razones por las que se ha escogido.

La arquitectura está organizada de tal manera que el modelo que defina el usuario esté lo más desacoplado posible del módulo de chequeo, a diferencia de otros *Model Checkers* que imponen un lenguaje propio para definir el modelo. Desde el punto de vista de los lenguajes de modelado propios de las herramientas de *Model Checking*, esta idea da libertad al desarrollador para poder definir su modelo en un lenguaje de programación real o en cualquier otro lenguaje de su elección. Es decir se puede usar un lenguaje para el que ya hay soporte, crear un lenguaje nuevo a conveniencia, o simplemente programar en Java (u otro lenguaje de programación compatible) el modelo y su intérprete. Con esta última opción se intenta liberar al desarrollador de adaptar el modelo que quiere chequear a un lenguaje de modelado concreto. Estas alternativas pretenden dotar de flexibilidad al *Framework*. Además se pueden conseguir ciertas mejoras en el comportamiento del verificador a través del uso de lenguajes más apropiados al ámbito del modelo a comprobar.

Aprovechando esta flexibilidad y al haber separado el funcionamiento en 5 conceptos (modelo e intérprete, motor de chequeo, fórmulas y proposiciones), se quiere que el *Framework* sea reutilizable y útil para usuarios no expertos en *Model Checking*. También se desea facilitar las operaciones para mostrar, representar y navegar a través del ejemplo o contraejemplo devuelto después del proceso de verificación. Usando esta idea en la respuesta del *Checker*, se deja libertad en cómo visualizar los resultados para facilitar su estudio. Para ello, el desarrollador no necesita conocer ningún detalle propio del sistema de chequeo, únicamente debe conocer la estructura de los estados definidos para su propio modelo.

Por último, una característica deseable para el *Framework* es la posibilidad de optimizar el algoritmo de chequeo de forma independiente. Para poder llevar a cabo esta funcionalidad se ha decidido modularizar el algoritmo de chequeo. Existe una implementación del motor de verificación cuyo algoritmo está modularizado a nivel de fórmula, es decir, se puede escoger una u otra estrategia de comprobación según sea el

tipo de operación lógica (por ejemplo And, Or, AU, EU,..). Esto permite usar diferentes estrategias o implementaciones de las mismas estrategias para el análisis de los diferentes operadores de las fórmulas. Así se permite generar distinstas estrategias o acercamientos que se adapten al modelo a chequear o incluso a una propiedad en concreto.

III.5.3 El Modelo

El modelo debe ser representado a poder ser por una máquina de estados finita. Aunque, la restricción de que sea finita o no depende del algoritmo de *Model Checking*. El algoritmo que se proporciona por defecto está preparado para máquinas de estados infinitas. La limitación entonces se encuentra en el almacenamiento.

Para una gran mayoría de sistemas, existe una representación a un determinado nivel de abstracción donde el sistema se reduce a una máquina de estados finita. Conocer a qué nivel es necesario estudiar el sistema ayuda a encontrar dicha representación o a encontrar representaciones más eficientes.

III.5.4 El Intérprete

Es una interfaz que define las operaciones mínimas necesarias para que el motor de chequeo pueda trabajar con el modelo:

- Indicar un posible conjunto inicial de estados.
- Generar las transiciones entre estados.
- Las transiciones deben de tener un identificador (nombre) para facilitar la visualización de resultados, con el fin de poder indicar las transiciones seguidas.

El desarrollador debe proporcionar una clase que implemente esta interfaz y que será la conexión entre el modelo a estudiar y el algoritmo de *Model Checking*. Cada vez que el motor de chequeo pregunte cuáles son los posibles estados siguientes a un estado dado, esta clase debe computar todas las posibles transiciones desde ese estado. Dicho de otra forma, cada estado de ese conjunto debe representar una línea temporal distinta. Esta interfaz puede estar implememtada por un intérprete de un lenguaje específico para un dominio de modelos, o por una clase que tenga programadas las reglas de transición del modelo a evaluar.

III.5.5 Fórmulas y Proposiciones para definir Propiedades

III.5.6 Fórmulas

La lógica empleada en la implementación actual del motor del *Model Checker* es la lógica temporal CTL (*Computation Tree Logic*). Por tanto, las fórmulas están definidas en esta lógica temporal.

Las fórmulas en CTL se definen a partir de las conectivas básicas, que están representadas por sus clases homólogas. Estas fórmulas son:

Not: Negación lógica

And: Conjunción

Or: Disyunción

AX: Para todo estado siguiente que cumple un propiedad.

EX: Existe un estado siguiente que cumple un propiedad.

AU: Para-todo hasta (*always until*) A(f1 U f2). Quiere decir que todo camino a partir del estado actual cumple f1 hasta que llega a un estado que cumple f2.

EU: Existe hasta (*eventually until*) E(f1 U f2). Quiere decir que existe un camino en el que siempre se cumple f1 antes de alcanzar un estado que cumple f2.

Este grupo reducido de fórmulas permite cubrir todo el abanico de propiedades definibles en CTL, ya que cualquier otra operación lógica en CTL se dpuede expresar en términos de las fórmulas enumeradas.

En el *Framework* las fórmulas se representan como un árbol, que posee solamente nodos con uno o dos hijos, y las hojas son proposiciones. Supongamos la formula: E(Not f1 U f2), siendo f1 y f2 dos proposiciones cualesquiera. Esta fórmula se representa como muestra la Ilustración 8.

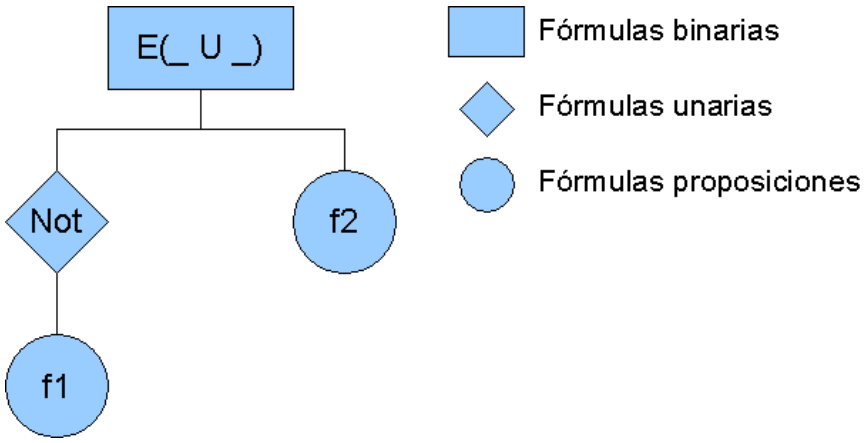


Ilustración 8 Ejemplo de fórmula

III.5.7 Proposiciones

En nuestro *Model Checker*, al no poseer éste un lenguaje de modelado propio, no podemos aplicar proposiciones sobre un estado, ya que éste y el intérprete son transparentes en su funcionamiento. Es decir, a priori, el *Checker* no necesita conocer la semántica del modelo ni el contenido del estado, aunque ambas son necesarias para una de las características relevantes de la implementación de nuestro *Model Checker*. Ésta consiste en no restringir al desarrollador en un lenguaje propio, pero para conseguir este objetivo se necesita que se definan proposiciones que son simples preguntas a un estado. Por lo tanto, esta decisión implica que el desarrollador defina una serie de proposiciones propias a cada intérprete y su modelo a representar. De esta forma, a partir de las proposiciones, el *Checker* puede realizar una pregunta cualquiera definida

en las proposiciones a cualquier estado proporcionado por el intérprete.

En resumen, las proposiciones están personalizadas a un ámbito y siempre deben responder a la pregunta "¿se cumple?" para cualquier estado del sistema. Dicho de otra forma, son funciones de estados a valores de verdad. Por lo tanto son un concepto lógico que complementa a las fórmulas.

III.5.8 El motor de chequeo

El motor es el módulo que implementa toda la lógica computacional de los algoritmos de *Model Checking*. Como actualmente el motor sólo acepta formulas en lógica CTL, su entrada está formada por un estado inicial, una fórmula en lógica CTL y el intérprete del modelo. Este motor verifica la fórmula en el estado inicial y devuelve un objeto *Resultado* que contiene la respuesta y un ejemplo o contraejemplo que respalda dicha respuesta.

Este motor actualmente se encuentra dividido en módulos, con el objetivo de que sea posible la elección del algoritmo de verificación a utilizar. Por defecto se proporciona una clase que implementa los métodos de los algoritmos de tal manera que permite elegir qué algoritmo utilizar a nivel de fórmula, ya que puede ser interesante en los algoritmos de *Model Checking* especificar una estrategia distinta para cada tipo de fórmula. El patrón *Visitor* es la estrategia más natural para implementar cualquier algoritmo de este estilo, ya que las fórmulas se representan como un árbol. Por lo tanto, todos los algoritmos que se quieran añadir al sistema deben seguir el patrón *Visitor* y si lo que se quiere añadir es un módulo, tiene que seguir un protocolo impuesto por el conector inter-modular (o algoritmo "envoltorio").

El algoritmo de *Model Checking* que hemos desarrollado comienza en el estado inicial, computa el valor lógico de la fórmula en ese estado, y detiene la verificación en cuanto determina el resultado y dispone de un sub-grafo ejemplo(o contraejemplo[I]) mínimo donde la fórmula es cierta (respectivamente falsa). En cualquier algoritmo, en general:

- El cómputo de una fórmula atómica (proposiciones) verificará la fórmula en ese estado.
- Una fórmula no atómica computará su resultado a partir de los cómputos de sus sub-fórmulas en el estado dado.
- Una fórmula temporal, adicionalmente, computará su resultado a partir de los resultados de sus sub-fórmulas en el estado dado y sus estados siguientes.

En cuanto a los contraejemplos, cada vez que una fórmula no atómica evalúa una sub-fórmula en un estado determinado, debe añadir el sub-grafo obtenido en dicha evaluación al sub-grafo resultado de evaluar la fórmula no atómica. Pongamos por ejemplo una fórmula E(f1 U f2), donde f1 y f2 son atómicas, y supongamos que el resultado es cierto. El ejemplo sería un camino donde f1 se cumple en todos los estados (quizás excepto en el último) y en el último se cumple f2. Sin embargo, si f1 fuera del tipo E(f3 U f4), con f3 y f4 atómicas, el ejemplo que obtenemos[1] sería un camino como

[I] Consideramos ejemplo o contra-ejemplo, según la respuesta devuelta sea cierta o falsa respectivamente

el del ejemplo anterior en el cuál desde cada estado que cumple f1 sale adicionalmente otro camino, que es el ejemplo de f1 evaluado en dicho estado (ver Ilustración 9).

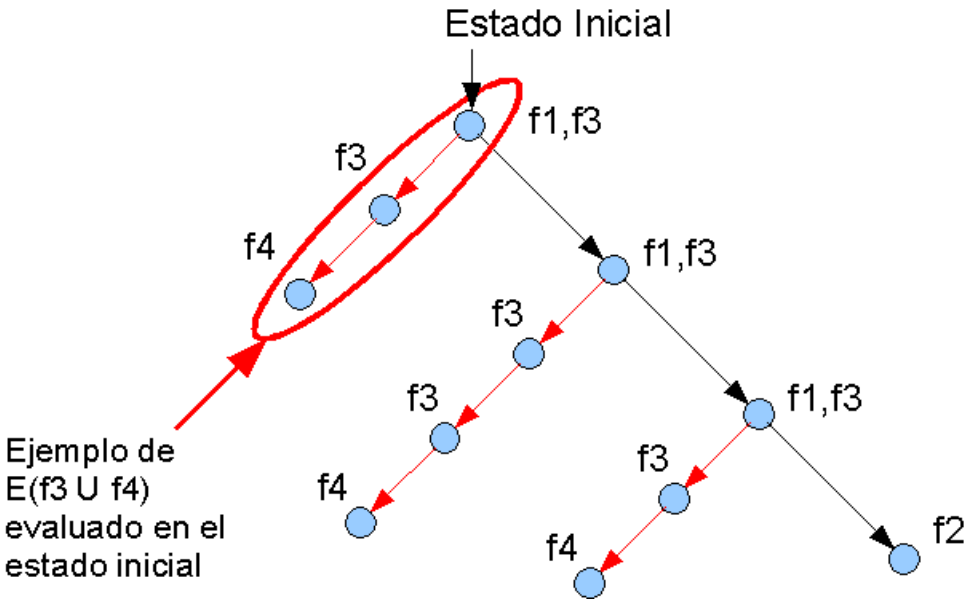


Ilustración 9 Desarrollo a lo largo de la verificación

Nota 1: el ejemplo que aquí se describe es un ejemplo elegido para facilitar la comprensión de lo que se quiere transmitir. Para ello se supone que todos los caminos tienen por lo menos dos estados y que no se repite ningún estado entre dos caminos diferentes.

A priori no se conoce el resultado de la fórmula no atómica, por lo que durante el cómputo será necesario mantener a la vez dos sub-grafos: el potencial ejemplo y el potencial contraejemplo. Normalmente uno de ellos se mantiene implícito en alguna estructura y se construye sólo al final, cuando ya se conoce el resultado. Es el caso de la implementación recursiva del algoritmo que verifica una fórmula $E(f_1 \, U \, f_2)$, en el que se mantiene una estructura con el subgrafo contraejemplo durante todo el cómputo y el subgrafo ejemplo se mantiene implícito en la pila de llamadas, y únicamente al encontrar un camino ejemplo se construye el subgrafo ejemplo correspondiente.

A continuación vamos a explicar el chequeo de algunas fórmulas temporales:

III.5.8.1 E(f1 U f2)

Llamaremos estado inicial al estado donde se quiere evaluar la fórmula. Si en este estado inicial se cumple f2, devolvemos cierto y el ejemplo será este estado inicial unido al ejemplo de f2. En caso de que no se cumpla f2 y tampoco se cumpla f1, se devuelve falso y este estado inicial será el contraejemplo, unido al contraejemplo de f1 y unido al contraejemplo de f2. De esta manera el sub-grafo resultado contiene toda la información necesaria para verificar la fórmula E(f1 U f2) (ver Ilustración 10).

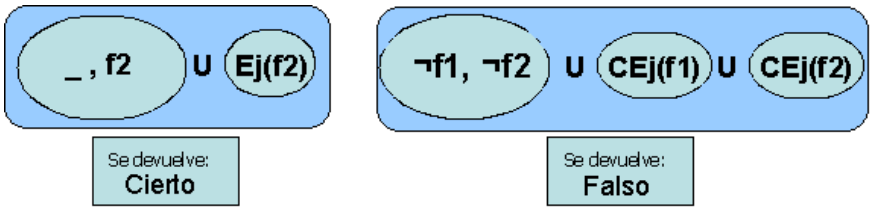


Ilustración 10 Ejemplo y Contra-Ejemplo para "E(f1 U f2)"

Si en el estado inicial no se cumple f2 pero se cumple f1, entonces hay que buscar algún hijo que cumpla E(f1 U f2). Si no tiene hijos, se devuelve falso, ya que no existe ningún camino que al final cumpla la condición f2, y como contraejemplo se devuelve el estado inicial unido al ejemplo de f1 y al contraejemplo de f2. Si tiene hijos, llamemos a este estado inicial P, que será el estado inicial tanto en el ejemplo como en el contraejemplo. Como contraejemplo potencial ya tenemos el estado P unido al ejemplo de f1 y al contraejemplo de f2. Cada estado hijo H que no cumpla E(f1 U f2) formará parte del contraejemplo, al cual se añade la arista (P,H) y se une con el contraejemplo de E(f1 U f2) evaluado en H. La Ilustración 11 muestra un caso en que ningún hijo cumple la fórmula E(f1 U f2).

En caso de que un hijo H cumpla E(f1 U f2), se devuelve cierto y como ejemplo se devuelve P, unido al ejemplo de f1, al contraejemplo de f2, al ejemplo de evaluar E(f1 U f2) en H y añadiéndole la arista (P, H). En la 11 se muestra este caso.

En el proceso descrito se mantiene un conjunto de estados visitados. Si en cualquier nivel de anidamiento se está evaluando E(f1 U f2) en un estado P', y un hijo H' pertenece a este conjunto, significa que anteriormente se ha evaluado H' y no cumplía f2. En tal caso la arista (P', H') ha de añadirse al contraejemplo potencial, pero no hay que evaluar recursivamente en este estado puesto que estamos dentro de dicha evaluación y, aunque sea posible que a partir de H' se encuentre un camino que cumpla E(f1 U f2), el camino más corto es el camino que primero visitó H' (ver Ilustración 12).

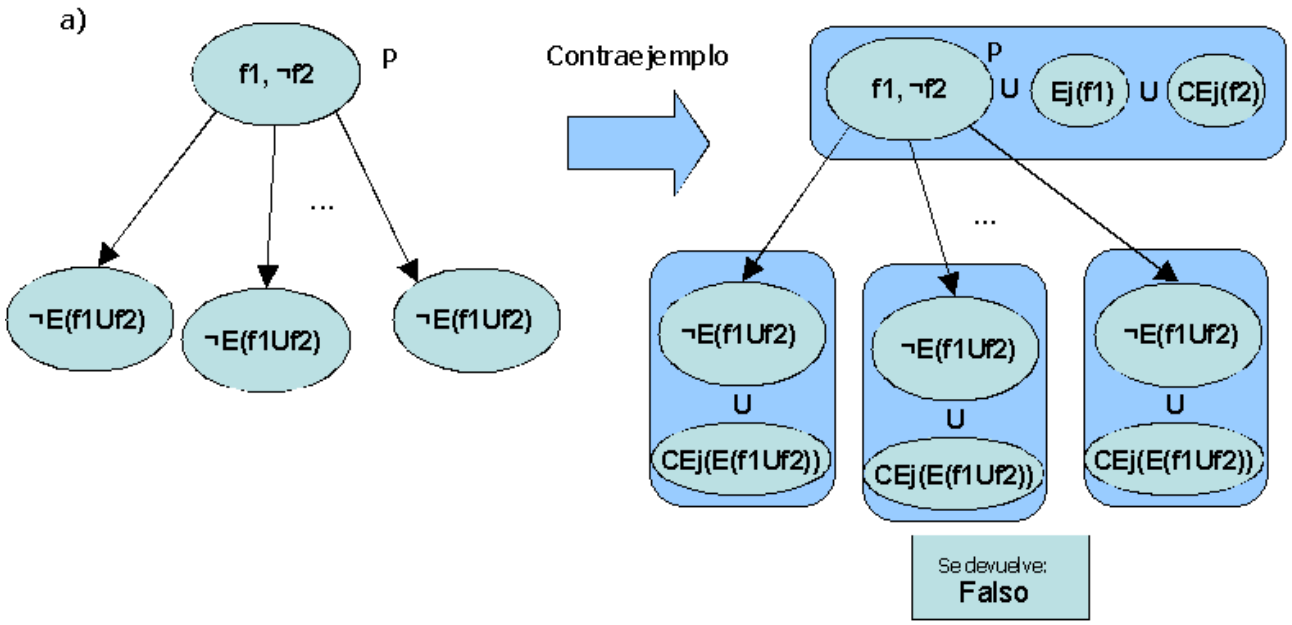


Ilustración 11 Generación de Contra-Ejemplo para "E(a U b)"

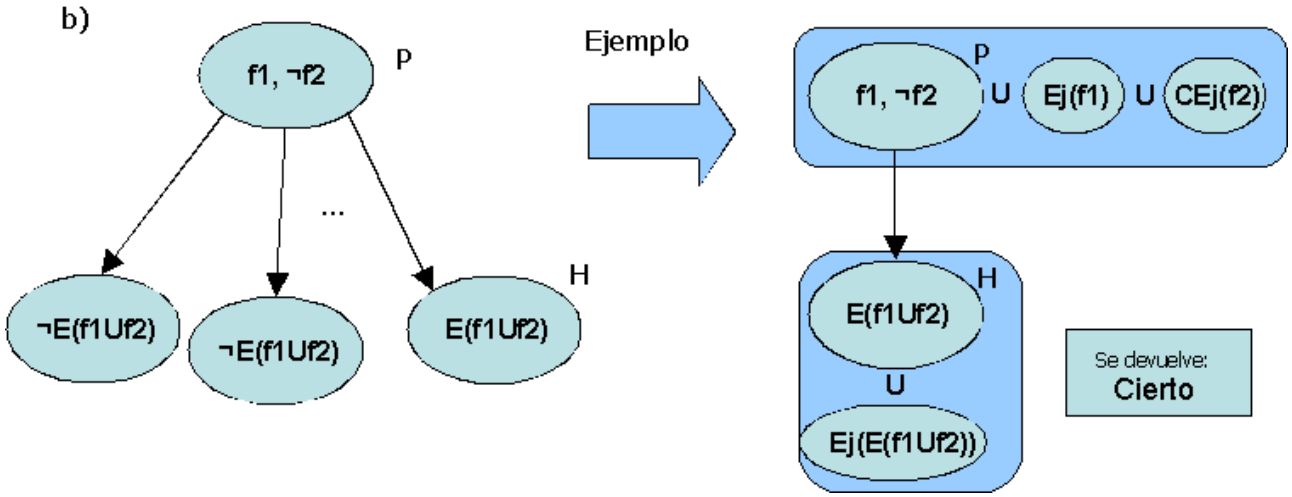


Ilustración 12 Generación de Ejemplo para "E(a U b)"

c)

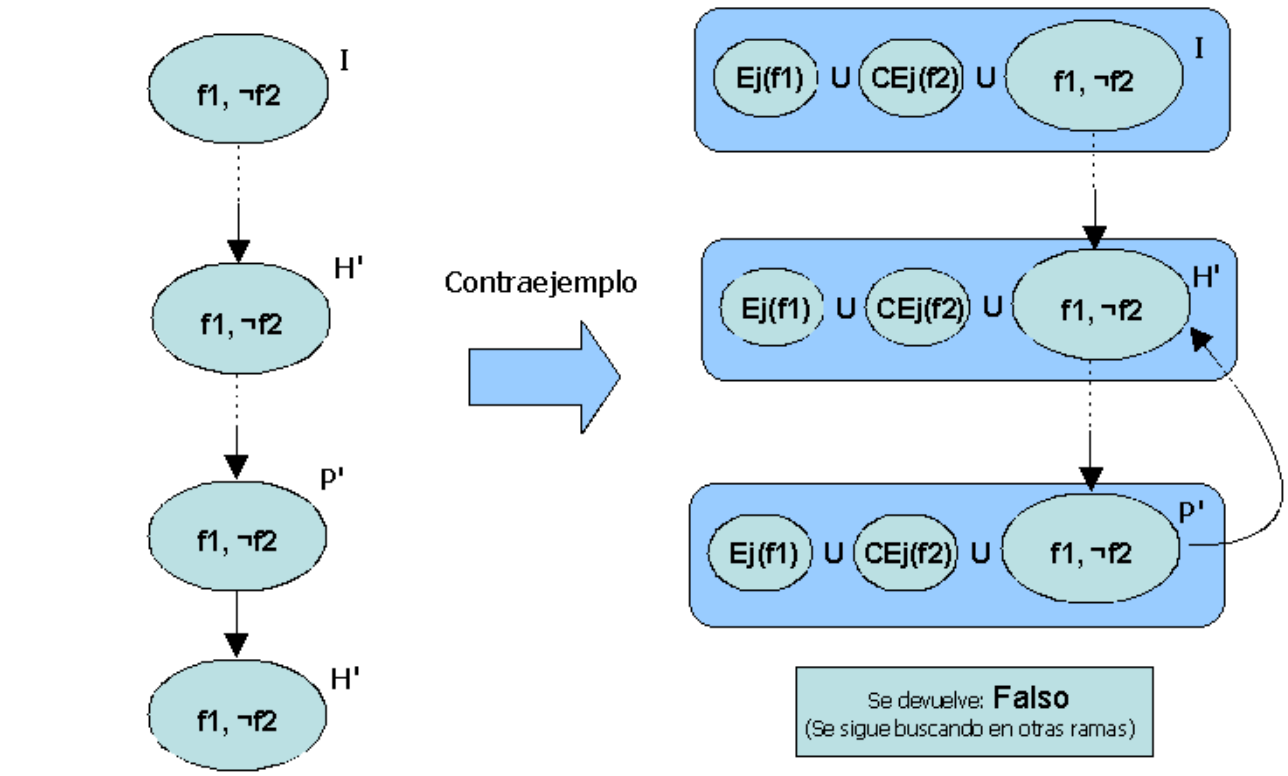


Ilustración 13 Generación de Contra-Ejemplo para "E(a U b)" caso de camino infinito

Como optimización hemos añadido que cuando un padre P tiene un hijo H que ya ha sido visitado como hijo de otro padre P', únicamente hay que añadir al potencial contraejemplo la arista (P, H) y no hay que volver a procesar el estado H. Dicho de otra forma, los caminos a partir del estado H estarán repetidos, por lo que con computar el más corto es suficiente para probar la fórmula. Esto supone una gran optimización en una implementación que guarde durante el cómputo todos los caminos ejemplo. En este caso se reconoce que un sub-grafo está repetido y sólo se procesa una vez.

Este algoritmo se ha implementado usando la idea de búsqueda en anchura para poder explorar máquinas de estados infinitos. Para el contraejemplo se usa un único grafo, que en caso de ser falsa la fórmula será todo el sub-grafo procesado a partir del estado inicial. Para los posibles ejemplos que se están computando, necesitamos guardar para cada estado explorado todo su camino hasta la raíz. Para que ello no suponga un gran coste en memoria, se optimiza el espacio compartiendo sub-grafos.

III.5.8.2 A(f1 U f2)

El caso de A(f1 U f2) es análogo al de E(f1 U f2). En este caso, el ejemplo es todo el grafo procesado y si se encuentra un contraejemplo se devuelve un camino del grafo, posiblemente con ciclos, unido a todos los ejemplos de f1 y contraejemplos de f2 evaluados en cada uno de sus estados. En los casos básicos se procede exactamente

igual que con E(f1 U f2): si se cumple f2 se devuelve cierto y el estado inicial unido al ejemplo de f2 como ejemplo; si no se cumple f2 ni tampoco f1 se devuelve falso y el estado inicial junto a los contraejemplos de f1 y f2 como contraejemplo; si sólo se cumple f1, se exploran los hijos, comprobando que todos los hijos cumplan A(f1 U f2).

Si es necesario explorar los hijos, pero el estado inicial no tiene hijos, se devuelve el contraejemplo de f2 unido al ejemplo de f1 y al estado inicial como contraejemplo, ya que no se cumple f2 en ningún momento. Si tiene hijos, se procede a explorar los hijos recursivamente. Llamemos P al estado actual donde se quiere verificar la fórmula. Para todos los hijos H que cumplen A(f1 U f2) se añade la transición (P,H) y el ejemplo de A(f1 U f2) al grafo ejemplo potencial. Si todos los hijos cumplen A(f1 U f2), se devuelve cierto y el ejemplo potencial como sub-grafo ejemplo. Si se encuentra un hijo H que no cumple A(f1 U f2), se devuelve falso y un grafo contraejemplo resultado de unir el ejemplo de f1 en P, el contraejemplo de f2 en P y el contraejemplo de A(f1 U f2) en H y añadiéndole la arista (P, H).

En cuanto a los duplicados, aquí es necesario proceder de manera diferente a como se hizo en E(f1 U f2). Si al computar un camino llegamos a un hijo H ya visitado en este camino, entonces hemos encontrado un camino infinito donde nunca se llega a cumplir f2, y consecuentemente la fórmula A(f1 U f2) es falsa y tenemos que devolver como grafo contraejemplo el camino con el ciclo unido a todos los ejemplos de f1 y contraejemplos de f2.

Por otro lado, la optimización análoga a la vista para el caso de E(f1 U f2) consiste en que si estamos evaluando la fórmula A(f1 U f2) en un estado P y tiene un estado H que ya ha sido visitado como hijo de otro padre P' que no pertenece al camino hasta la raíz, se evita la evaluación doble a partir de H detectando el duplicado con un conjunto visitados global a todo el proceso. En tal caso se añade al potencial ejemplo la arista (P, H).

Dadas la sdiferencias en el tratamiento de los nodos duplicados cuando forman parte o no del camino actual, se hace necesario mantener dos conjuntos para visitados: un conjunto visitados del camino actual (el camino seguido desde el estado inicial hasta el estado actual donde se está evaluando A(f1 U f2)) y otro conjunto visitados global a todo el proceso. Así, al detectar un duplicado podremos diferenciar en cuál de las dos situaciones estamos y determinar si la fórmula es falsa o si debemos seguir evaluando pero por otra rama.

III.5.8.3 EX(f)

En este tipo de fórmulas se necesita verificar si existe algún hijo que cumpla f. Como caso básico, un estado que no tiene hijos no cumple la fórmula y el contraejemplo es ese mismo estado. Si tiene hijos, se parte de un potencial contraejemplo que únicamente tiene a este estado (a partir de ahora P) y se procede a explorar todos los hijos. A medida que se exploran hijos H que no cumplen f, se van añadiendo al contraejemplo las aristas (P, H) y los contraejemplos de evaluar f en cada estado H. En caso de que ningún hijo cumpla f se devuelve falso y como contraejemplo el contraejemplo potencial que se ha calculado. Si por el contrario se encuentra un hijo H’ que cumple f, se devuelve cierto y como ejemplo el ejemplo de f evaluado en H’ añadiéndole la arista (P, H’).

III.5.8.4 AX(f)

Este caso es parecido al anterior, pero buscando algún hijo que no cumpla f. El caso básico es un estado que no tiene hijos y no cumple la fórmula, y el contraejemplo es ese mismo estado. Si tiene hijos, se parte de un potencial ejemplo que únicamente tiene a este estado (a partir de ahora P) y se procede a explorar todos los hijos. A medida que se exploran hijos H que cumplen f, se van añadiendo al ejemplo las aristas (P, H) y los ejemplos de evaluar f en cada estado H. En caso de que todos los hijos cumplan f, se devuelve cierto y como ejemplo el ejemplo potencial que se ha calculado. Si por el contrario se encuentra un hijo H' que no cumple f, se devuelve falso y como contraejemplo el contraejemplo de f evaluado en H' añadiéndole la arista (P, H').

III.5.9 Los resultados del chequeo

Los resultados del proceso de chequeo se clasifican en:

- Ejemplo.
- Contraejemplo.
- Error.

El resultado del chequeo es un grafo si no se produce un error, generalmente de agotamiento de recursos. El proceso de obtención de este grafo es mencionado en el apartado anterior. La categorización en ejemplo o contraejemplo se realiza por el rol que desempeña el resultado. Si resulta que la fórmula se cumple en el espacio de estados, entonces el resultado será una muestra de su cumplimiento, al que denominamos ejemplo. Si la fórmula no se cumple en el espacio de estados, el resultado mostrará que por las transiciones posibles no se alcanza un estado de éxito.

Hay que tener en cuenta que la propiedad preguntada y como se representa la fórmula correspondiente influyen en el tamaño del "grafo de respuesta". Tal consideración puede ser interesante cuando se desea verificar cierta propiedad controlando que no se tenga un grafo demasiado grande para explorar.

III.5.10 Visualizador de resultados

En el caso de que al usuario le interese algo más que una simple respuesta de cierto o no, puede extender la visualización básica del resultado. Como ya sabemos, el resultado consiste en un ejemplo (para el caso de que la respuesta sea cierta) que demuestra la que la propiedad es cierta, o un contraejemplo (para el caso de una respuesta falsa) que muestra los casos en el que la propiedad es errónea. Este ejemplo o contraejemplo consiste en un grafo que contiene un sub-conjunto de estados y transiciones que demuestra o desmiente la propiedad a comprobar.

Para extender la visualización básica, este Framework incluye una serie de componentes, descritos a continuación.

III.5.11 El visualizador básico por ventanas

En sí mismo consiste en una ventana Swing. La ventana se divide en 2 secciones

- Panel de visionado: es un panel sobre el que se visualiza la información del estado por el que estamos navegando.
- Panel de botones: Muestra las posibles transiciones que se pueden realizar desde el estado actual (el estado mostrado en el panel de visionado). Incluye un botón retrocede que ejecuta la acción de retroceder al estado anterior al mostrado en el panel de visualización.

Todo resultado es un grafo de estados y transiciones. El componente de navegación simplemente recorre este grafo. El navegador es el encargado de interactuar con el intérprete para conseguir pares de transiciones-etiquetas. Llamamos pares transiciones-etiquetas a la forma de identificar transiciones. Su propósito es mejorar la comprensión de los cambios que pasan por el sistema. La forma de navegar consiste en avanzar o retroceder (u opcionalmente ir a un estado concreto).

Ilustración 14 Esquema del panel de navegacion

La opción de avanzar se determina por la transición seleccionada desde el panel de transiciones. Tenemos la opción de retroceder para permitir al desarrollador que explore diferentes caminos sobre el grafo resultado. Como se muestra en la Ilustración 15, si se considera realizar la operación de avanzar, se puede seleccionar entre las tres posibles transiciones (1, 2, 3). Si elegimos la transición 2, se visita ese nodo del grafo resultado y se pueden realizar otras transiciones nuevas (4 y 5). Si se selecciona retroceder se retorna al nodo inmediatamente anterior en el grafo resultado. En la figura se denota el camino seguido por las aristas coloreadas de naranja. Retroceder supone retrocede un paso en el camino.

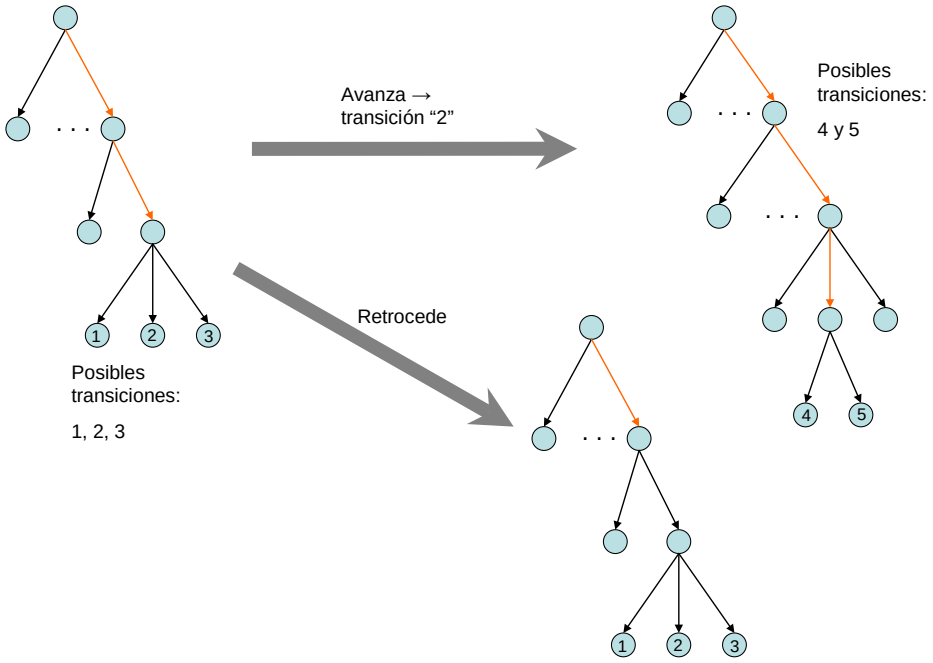


Ilustración 15 Esquema del proceso de navegación

III.5.12 El Navegador

Nace como una solución para facilitar el manejo de transiciones por el grafo resultado. Es un componente que se encarga de realizar las operaciones de navegación sobre el grafo resultado, además de suministrar información sobre transiciones y el recorrido realizado. Además permite separar completamente la navegación de la forma de representar el resultado, y por ello generar sistemas de visualización independientes (por consola, web,...). También ofrece la posibilidad de lanzar diferentes sistemas de visualización a la vez sin que estos se estorben, ya que implementa un patrón "Listener".

Con estos componentes ya implementados, el desarrollador puede optar por:

- Usar el framework ya definido, con lo cual sólo debería definir un objeto de tipo "*Drawer*". Este objeto se encargaría de rellenar un objeto *JPanel* con componentes gráficos de ventana (swing, awt, ...) para representar un estado dado, dejando así que el componente Visualizador se encargue del resto.

- Generar un nuevo *Visualizador*. Para hacerlo tendría que generar un objeto que herede de "*Animador*" y registrarlo como oyente del *Navegador*. De esta manera se podría usar el navegador como interfaz para recorrer el grafo respuesta. El resto queda en manos del desarrollador.

IV. Caso de estudio: El Laberinto

IV.1 Modelo

Un laberinto es un objeto de estudio y entretenimiento que creció con el avance de la informática. Presenta como característica que permite ajustar su complejidad al nivel del análisis que se desee. En este caso, el interés por este estudio es el de formalizar un primer contacto con los algoritmos de *Model Checking*.

Sirviendo al propósito anteriormente mencionado las características del laberinto son simples y fáciles de manejar. En primer lugar el laberinto es bidimensional, similar a un tablero de ajedrez y dividido en casillas. La división en casillas se corresponde a la representación del mismo que se revelará avanzado este caso de estudio.

Además del tablero, se hace uso de un personaje que corresponde a la persona que se maneja por el laberinto. Este personaje, para simplificar, sólo puede moverse a las casillas contiguas y en 4 direcciones: norte, sur, este y oeste. El laberinto dispone de una casilla inicial, que es donde aparece por primera vez el personaje, y una casilla de salida, la cual indica la salida del laberinto.

La representación de las casillas es simple: se identifican por las coordenadas en el tablero, igual que en el ajedrez, y tienen una única propiedad, ser transitable o no. Todas estas características son inmutables a lo largo de todo el proceso de *Model Checking*: la identificación de las casillas es única e invariable, y si una casilla es intransitable lo será siempre. El personaje sólo puede ejecutar una acción de movimiento si la casilla destino es transitable.

La casilla de salida no es necesario que sea única, sino que para este laberinto se pueden usar varias. Para simplificar este estudio, la mayoría de las pruebas que se han realizado han considerado una única salida, sin embargo se ha probado el soporte para varias.

Con todas estas características la traslación a máquina de estados es sencilla. Las operaciones de transición se corresponden con las operaciones de movimiento del personaje por el laberinto. Como las propiedades de una casilla son inmutables, se observa que lo que varia a lo largo del recorrido es únicamente la posición del personaje. A partir de esta observación se considera que el estado de la máquina se corresponde con la posición del personaje en el laberinto.

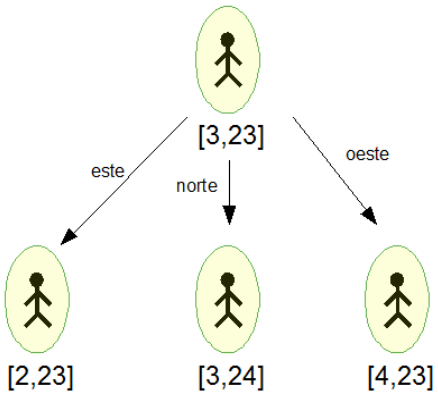


Ilustración 16 Esquema máquina de estados para el Laberinto

IV.2 Intérprete

La formalización del intérprete a partir de las características que se describieron en el apartado anterior tiene en cuenta que el laberinto posee información estática, las casillas que son inmutables, e información dinámica, la posición del personaje. En cuanto a la información estática, se refiere al tamaño del laberinto, las casillas del mismo y las propiedades de transición por ellas.

El intérprete también se encarga de suministrar las funciones de transición del personaje. Para realizar la transición en el laberinto se comprueban las casillas contiguas y aquellas que sean transitables se manejan como posibles movimientos del personaje. El personaje puede moverse en los 4 sentidos cardinales, sin embargo limitamos sus posibilidades de movimiento antes de avanzar, en vez de comprobar que la casilla a la que llegó es intransitable. Este aspecto se contempla al definir las funciones de transición de un estado a otro.

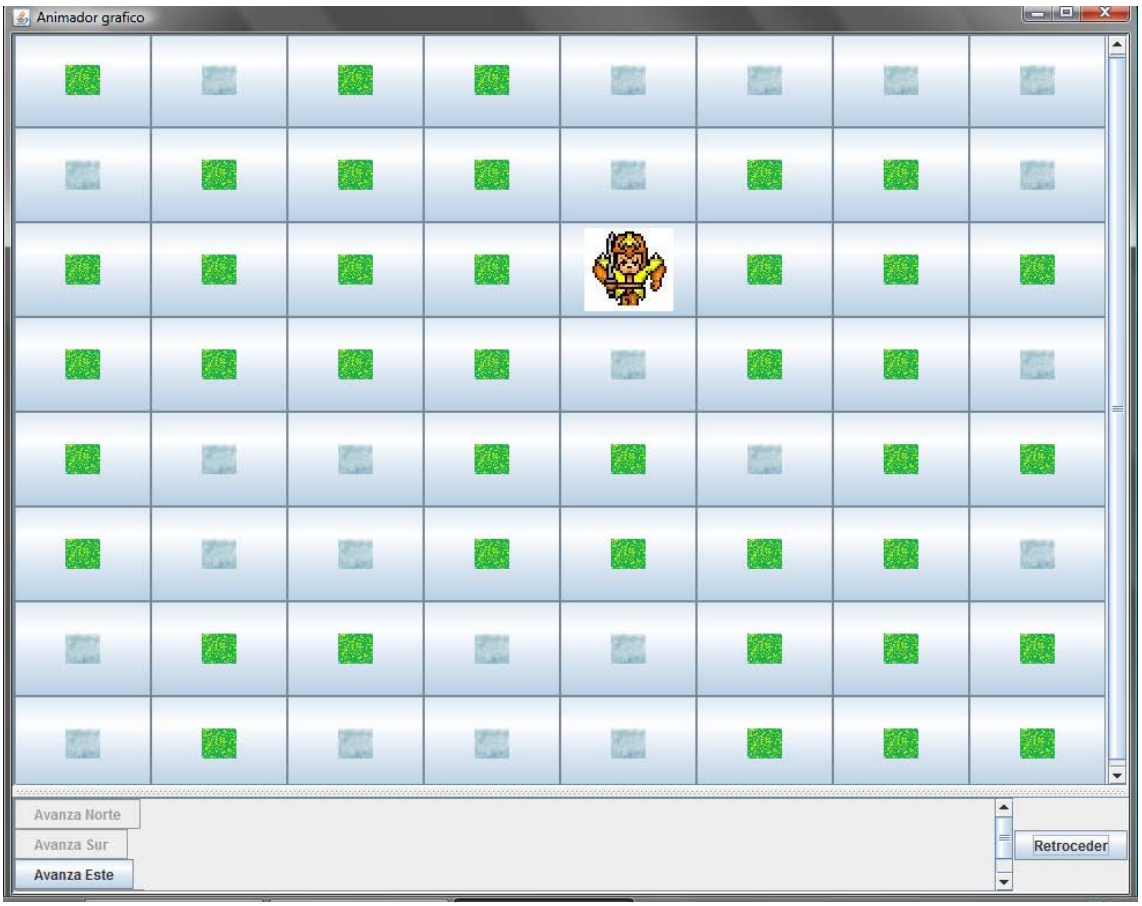


Ilustración 17 Ejemplo del laberinto, opciones de avance.

Como se puede suponer a partir de las líneas anteriores, el estado del laberinto está definido por la información dinámica del mismo, que se corresponde con la posición del personaje. Así encaja con las funciones de transición: el estado actual es la posición del personaje, las transiciones proponen casillas a las que avanzará el personaje y que a su vez son futuros estados del laberinto.

El propósito principal del *Model Checking* que se aplica sobre el laberintoes encontrar una serie de "estados futuros" que garantizan que el personaje llega a la salida del laberinto. Además, para poder visualizar el resultado con detalle, el intérprete dispone de etiquetado de transiciones. Este aspecto se explicará con más detalle en los apartados de resultados y visualizador.

IV.3 Proposiciones

Las proposiciones se rigen por los datos que intervienen en la definición de estado. En este caso, las proposiciones están ligadas a las posiciones sobre las que se transita.

Las proposiciones en este caso de estudio mantienen un carácter simple. Las posibles proposiciones que se han planteado para el laberinto se corresponden con las características de éste. Para determinar si una casilla es el punto de inicio, se necesitará una proposición que evalúe la casilla e indique si es el comienzo. De forma análoga se realiza la consulta de la salida del laberinto. Otro aspecto que nos interesa será la identificación de la casilla, pues siempre puede surgir la idea de preguntar por una casilla por la que pase necesariamente el personaje antes de llegar a la salida, o especificar aquellas que se desea que no formen parte de la solución.

A la hora de especificar la fórmula que se quiere verificar, hay que tener especial cuidado en la forma de construirla. Un ejemplo de ello sería lo siguiente: "Todos los caminos llevan a la salida." Aunque el laberinto tenga todas las casillas transitables, por las operaciones de transición se puede conseguir un contra-ejemplo que demuestra que existen caminos que nunca llevan a la salida. Basta con entrar en un bucle, por ejemplo: norte, sur, norte, sur, ...

Para evitar obtener este tipo de respuestas, que realmente no es lo que interesa, hay que jugar con las operaciones de transición, las propiedades de las casillas, y hasta habilitar restricciones en el contexto que se tienen en cuenta a la hora de la verificación.

Una forma fácil de filtrar el resultado del anterior ejemplo consistiría en no permitir que el personaje vuelva hacia atrás. En este caso se le limitaron los movimientos a sur y oeste. La restricción de movimientos provoca que el número de caminos posibles(teóricos, sin optimizaciones) para un laberinto sin obstáculos de NxN con casilla salida en la casilla (N,N) sea $\binom{2(N-1)}{N-1}$. Esto se puede verificar si se comprueba que para cada casilla, el número de caminos que llegan a ella son los caminos que pasan por su casilla contigua al norte más los caminos que pasan por la casilla contigua al oeste. Si se dibuja en una cuadrícula el número de caminos que pasa por cada casilla, se verá que corresponde al desarrollo matricial del triángulo de pascal. De todas formas, una demostración más formal puede hacerse por combinatoria: Para un laberinto de NxN, para llegar a la casilla (N,N) se tienen que caminar exactamente N-1 casillas hacia el este y N-1 casillas hacia el sur, se siga el camino que se siga. Cada camino es una secuencia ordenada de estos movimientos y todos los caminos forman el conjunto de todas las permutaciones posibles de estos $2N-2$ movimientos, distinguiendo los movimientos que son diferentes. Por combinatoria, tenemos que calcular las permutaciones con repetición de $2N-2$ movimientos donde se repiten los movimientos hacia el este N-1 veces y los movimientos hacia el sur N-1 veces. El número de caminos es entonces:

$$P_{2N-2}^{N-1,N-1} = \frac{(2N-2)!}{(N-1)!(N-1)!} = \binom{2N-2}{N-1}.$$

IV.4 Resultados

El resultado de la verificación que se obtiene de una consulta al laberinto es un grafo de posiciones por las que se ha desplazado el personaje, ya sea ejemplo o contraejemplo. De hecho, el concepto que se tiene es que si la verificación es correcta, se obtiene un grafo con el conjunto de todas las posiciones por las que pasa el personaje para cumplir la fórmula. Desde el punto de vista del laberinto, se correspondería con el conjunto de caminos por los que pasa el personaje para verificar la fórmula. En caso de que la fórmula no llegue a verificarse con éxito, se obtiene un camino que muestra la forma de violar la propiedad. El contraejemplo es el primer camino que se encuentra, que se supone es el más simple, aunque no se garantiza.

En este caso de estudio se ha optado por realizar un navegador que permita recorrer el grafo resultado. Para poder recorrerlo no sólo se necesitan las posiciones por las que se ha pasado, sino que también es necesario saber las operaciones que llevaron a ese estado. Por ello es necesario etiquetar las transiciones que se realizan a lo largo de la verificación. Dicho etiquetado es importante en el momento de la representación del resultado. Con él, el visualizador es capaz de mostrar las operaciones realizadas en la verificación, y en caso de que haya varias operaciones disponibles permitir al usuario decidir por cuál quiere continuar. Las decisiones en las transiciones están delimitadas por aquellas operaciones que la verificación ha usado para obtener el resultado.

Este caso de estudio se ha realizado bajo una minuciosa inspección de la complejidad de los resultados parciales y de cómo éstos componen el resultado final. Este análisis ha intentado que el rendimiento sea máximo proponiendo las estrategias descritas en los apartados de *Model Checking*.

IV.5 Visualizador

La visualización de los resultados obtenidos de la verificación se refleja en el visualizador básico proporcionado por el *Framework*. En el lienzo, donde se representa el estado del laberinto se ha usado un tablero de casillas. El estado del laberinto comentamos que estaba regido por la posición del personaje. Sin embargo, para obtener una visualización correcta y comprensible se ha representado también la información estática. Las casillas transitables se representan con un icono de color verde, que representa césped; en cambio, las no transitables, de color azul grisáceo, representan arroyos. El personaje aparece como un caballero que a lo largo de la navegación por el resultado cambia de posición. En la parte inferior del visualizador se encuentra el conjunto de operaciones realizadas en la verificación y disponibles para aplicarlas en la navegación por el resultado.

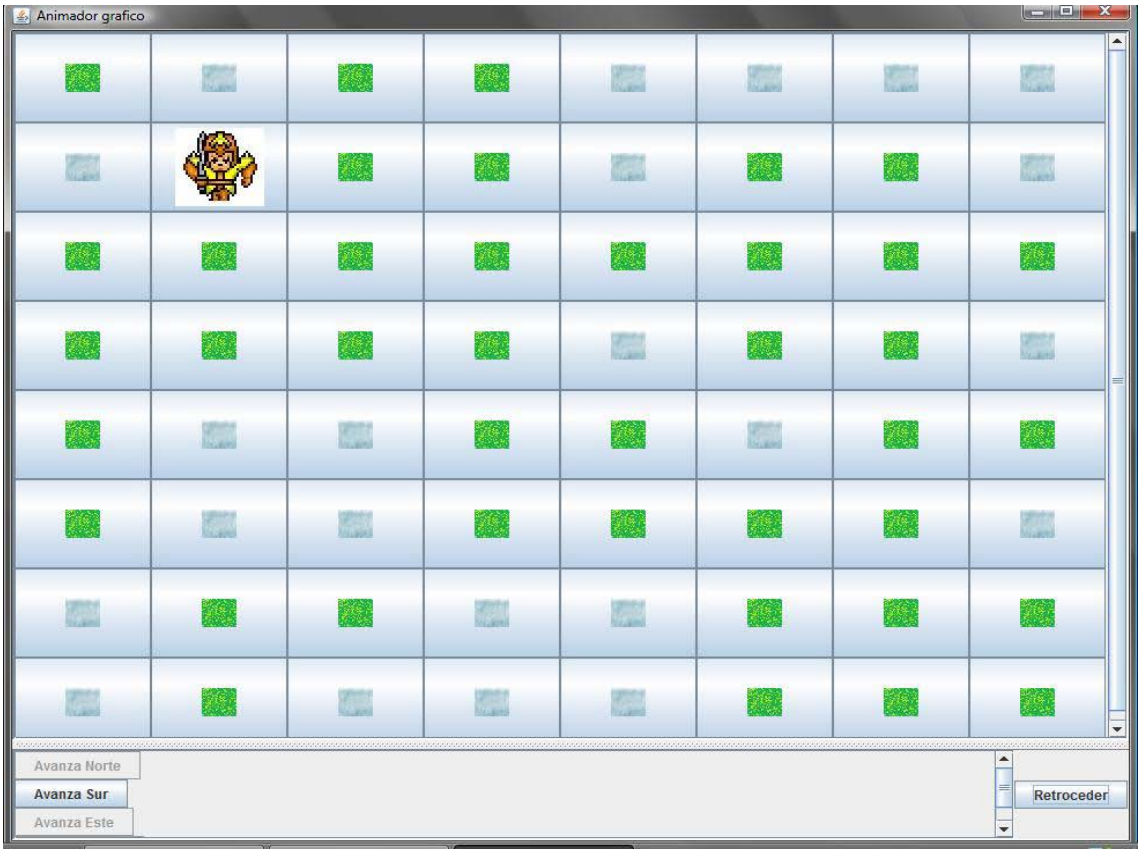


Ilustración 18 Representación del laberinto

V. Caso de estudio: Sistema de Actividades

V.1 Modelo Teoría de la Actividad

El segundo caso de estudio es la verificación de un sistema de actividades. En Teoría de la Actividad, un sistema está formado por comunidades que se pueden organizar jerárquicamente. Cada comunidad, además de un sistema de actividades, tiene un conjunto de artefactos o ítems. En este ejemplo se restringirá el modelo a una sola comunidad con un único sistema de actividades.

El sistema de actividades que aquí se modela es un conjunto de actividades. Dichas actividades pueden estar anidadas, es decir, que una actividad puede estar formada por varias sub-actividades. Se pueden ejecutar varias actividades paralelamente, pero cada actividad se puede ejecutar una sola vez. Si está formada por varias actividades, la actividad padre terminará de ejecutarse cuando finalicen todas sus actividades hijas. Por otro lado, las actividades hijas no se pueden ejecutar si su actividad padre no está en ejecución, ya que, conceptualmente, una actividad hija es "parte" de una actividad padre. En este modelo, cada actividad hija sólo tiene una actividad padre.

Los ítems pueden participar en las actividades ejerciendo diferentes roles: sujeto, objeto, objetivo, herramienta o producto. Además, las actividades pueden tanto generar como destruir ítems. Para poder ejecutarse, una actividad necesita que los ítems que participen en ella estén presentes (no estén destruidos) y que tampoco estén siendo usados por otra actividad. En consecuencia, si dos actividades necesitan un mismo ítem, una de ellas tendrá que esperar. Sin embargo, una actividad compuesta puede reservar ítems para que los utilicen sus actividades hijas. Dicho de otra forma, una actividad hija puede disponer de un ítem ocupado si éste está participando en la actividad padre.

En el caso de destrucción de ítems, las actividades pueden destruir ítems al finalizar, pero si el ítem está participando en otra actividad (una actividad antecesora de ésta), no pueden destruirlo.

Por último, para controlar la ejecución de actividades, éstas tienen un conjunto de condiciones de aplicabilidad. Una actividad sólo se puede ejecutar si se cumplen todas sus condiciones.

El estado del sistema de actividades está constituido por los estados en los que se encuentran todas sus actividades (en espera, ejecutando o finalizada) e ítems (disponible, ocupado o destruido). Más adelante explicamos el significado de estos estados y su interpretación en el paradigma de nuestro intérprete de Teoría de la Actividad.

Teniendo en cuenta la discusión anterior, el modelo de ejecución del sistema es como sigue. El modelo no contempla el tiempo que transcurre entre la disponibilidad de ejecución de una actividad y su ejecución: si una actividad se puede ejecutar, los ítems ocuparán sus roles y en el siguiente estado pasará a ejecución. En el momento en que se encuentre un conflicto entre actividades se considera el indeterminismo que existe en el modelo y los estados siguientes serán las diferentes posibles transiciones.

Cuando hablamos de conflicto entre actividades nos referimos al hecho de que haya más de una actividad dispuesta a ejecutarse en el sistema, con la peculiaridad de que tienen que competir por los ítems necesarios para iniciar la ejecución. Las transiciones posibles desde un estado con actividades en conflicto llevarán a varios estados, cada uno con los diferentes conjuntos de actividades posibles en ejecución que se pueden ejecutar simultáneamente.

V.2 El intérprete de Teoría de la Actividad

El intérprete se encarga de transformar toda la información sobre el sistema de actividades en una máquina de estados. El intérprete traduce la definición del sistema de actividades a un estado inicial del sistema, y luego se encarga de proveer las transiciones de cada estado. En cada uno de estos estados únicamente se almacena el estado de las actividades y el estado de los ítems.

Para las actividades, definimos tres estados:

- *Waiting*. La actividad se encuentra en espera. Ya sea porque no están disponibles todos los ítems necesarios o porque sus condiciones de aplicabilidad no se cumplen.
- *Executing*. La actividad se encuentra ejecutándose. En el caso de que la actividad sea compuesta, seguirá en este estado hasta que sus actividades hijas pasen a un estado finalizado (*Finalized*).
- *Finalized*. La actividad ya se ha ejecutado. Al finalizarse la actividad ya no puede volver a ejecutarse, aparecen sus productos y sus objetivos, y desaparecen los ítems que destruye.

Para el caso de los ítems, definimos tres estados también:

- *Free*. El ítem se encuentra libre y puede ser usado por cualquier actividad que lo necesite en los estados siguientes.
- *Busy*. El ítem está siendo usado por alguna actividad en ese estado.
- *Disposed*. El ítem ha sido eliminado o no existía en el estado inicial. Por lo tanto, no puede ser usado por una actividad. Sin embargo este estado puede ser revocado cuando otra actividad vuelva a crear la misma entidad como resultado de su ejecución.

En el código ejemplo, la especificación del modelo se hace en un lenguaje de objetos Java. La especificación del sistema sigue el siguiente proceso:

1. Se declaran todos los ítems que van a existir en el sistema, independientemente del estado que tomarán en el estado inicial del sistema.
2. Se especifican las actividades del sistema. Dicha especificación debe indicar los ítems que participan en ellas y los que van a ser creados o destruidos tras su ejecución.

3. Se especifica al intérprete la estructura jerárquica entre actividades; se registran los ítems y actividades creados y, por último, se le proporciona el estado inicial del sistema.

El enlace de todos estos elementos constituye la definición del sistema de actividades. Como ya mencionamos anteriormente el intérprete se encarga de proporcionar los estados iniciales. Para este caso de estudio, el estado inicial contendrá los estados iniciales de los ítems y los estados iniciales de las actividades.

En cada transición se finalizan las actividades que están completadas, se liberan los ítems que usa, y opcionalmente se crean o destruyen ítems dependiendo de su definición. Después de eso se decide qué conjunto de actividades se ejecutará. Si hay conflictos habrá varios estados siguientes posibles.

Para conseguir esto establecemos un grafo de compatibilidad, Cada nodo de este grafo representa una actividad. Las aristas entre nodos representan la relación de compatibilidad. Como ya mencionamos, decimos que dos actividades son compatibles cuando se pueden ejecutar a la vez en el sistema bajo el mismo estado, es decir, definimos que dos actividades son compatibles cuando se cumple alguna de las siguientes condiciones:

- Son parientes: una actividad es pariente de otra, sin importar el salto generacional.
- No son conflictivas entre sí: las actividades no comparten ningún ítem necesario para su ejecución.

Una vez construido el grafo de compatibilidad, tenemos para cada actividad el conjunto de actividades con las que puede ejecutarse simultáneamente. En cada transición usamos este conjunto y aplicamos un algoritmo de backtracking para obtener las posibles combinaciones de conjuntos de actividades paralelas.

Es importante observar que hemos tomado la decisión de lanzar conjuntos de actividades disponibles y no actividades de una en una. El impacto sobre la máquina de estados es que hay una menor cantidad de estados; en cuanto a la definición del sistema implica que no se han contemplado los casos en los que una actividad pueda ejecutarse y en vez de hacerlo espere.

V.3 Proposiciones de la Teoría de Actividades

Las proposiciones para este caso de estudio consisten en consultas sobre actividades e ítems, ya que, partiendo de la descripción del estado del sistema de actividades, es natural formular preguntas sobre los estados de los elementos que cambian en cada transición.

Mantenemos la estructura planteada de proposición. Ésta se basa en el estado del sistema. Diferenciamos dos tipos de proposiciones:

- Proposiciones de Actividades: estas proposiciones sirven para consultar el estado de una actividad.
- Proposiciones de Ítems: estas sirven para consultar el estado de un ítem (sus 3 estados y los roles que ha ejercido en los sistemas de actividad).

Combinando estos 2 tipos de proposiciones conseguimos definir cualquier proposición básica de nuestro sistema, concluyendo que es un sistema generador de estados en cuanto a que para todo estado del sistema, se puede especificar una fórmula que sólo sea cierta en ese estado y que especifique toda la información que ese estado codifica.

V.4 El visualizador

En este caso utilizamos la facilidad del visualizador básico del *Framework*. Como mencionamos anteriormente, éste se compone de 2 paneles.

Uno de los paneles contiene los botones con las posibles transiciones y un botón de control retroceso. En cada estado, estos botones se actualizan y se activan los que conducen a las siguientes transiciones posibles del ejemplo o contraejemplo. Para ello en nuestro intérprete solo tenemos que implementar la función *damePosibles()*.

En el otro panel representamos el estado, es decir, el conjunto de actividades del sistema, junto con sus estados y el estado de los ítems.

- Los ítems están en una lista a la izquierda, cada uno con icono que representa su estado. Un círculo coloreado significa que está en estado *Free*. Si el círculo está rodeado de una arandela roja significa que está en estado *Busy*. Si el círculo está tachado con una cruz roja significa que está en estado *Disposed*.

- Las actividades están organizadas en árboles en el centro. Si la actividad tiene letras rojas y un icono de un reloj es que está en estado *Waiting*. Si la actividad tiene un icono de un engranaje y letras negras es que está en estado *Executing*. Cuando una actividad está en estado *Finalized* sólo aparece su nombre desactivado.

Hay un visualizador en detalle de actividades a la derecha. A la izquierda de la actividad está la lista de ítems que participan en ella y a su derecha los ítems que genera. Si la actividad tiene letras rojas, un icono de un reloj y los ítems en blanco y negro, es que está en estado *Waiting*. Si la actividad tiene un icono de un engranaje, letras negras y sus ítems están coloreados, es que está en estado *Executing*. Cuando está en estado *Finalized* sólo aparece su nombre desactivado.

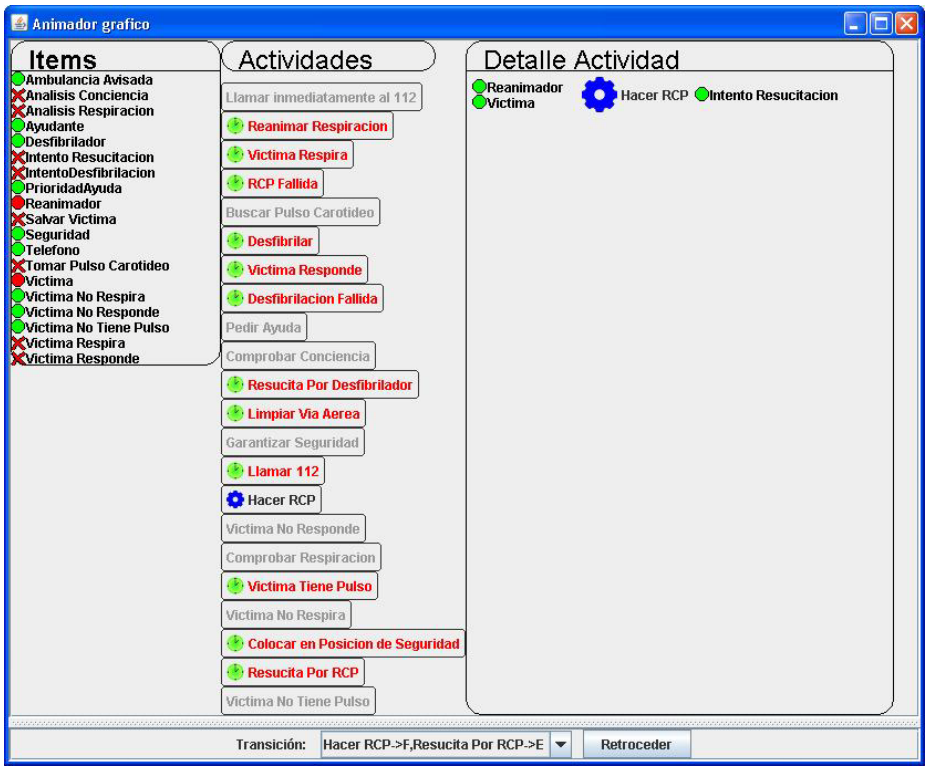


Ilustración 19 Visualizador de La Teoría de Actividades

V.5 Resultados

Aquí vamos a especificar un sistema de actividades y vamos a comprobar algunas propiedades sobre él. Se tomará como ejemplo las instrucciones de la Resucitación CardioPulmonar (RCP).

Las instrucciones de las que partimos son las siguientes:

RESUCITACION CARDIOPULMONAR BÁSICA:

1. *Garantizar la seguridad del reanimador y de la víctima.*
2. *Comprobar la conciencia (gritarle y zarandearlo suavemente). Si no responde, pedir ayuda a otros testigos.*
3. *Abrir la vía aérea (maniobra frente mentón)*
4. *Comprobar la respiración (ver, oír, sentir, máximo 10 segundos)*
   - *4.1. Si respira colocarlo en la posición de seguridad y llamar al 112.*
   - *4.2. Si no respira, llamar inmediatamente al 112*
     - *4.2.1. Si hay dos reanimadores, uno debe iniciar las maniobras de RCP y el otro ha de pedir ayuda.*
     - *4.2.2. Si hay solo un reanimador, se debe pedir ayuda aunque para ello se retrase unos minutos el inicio de las compresiones.*
5. *Valorar los signos de circulación: se debe observar la presencia de cualquier*

movimiento. Se debe buscar el pulso carotideo si en 10 segundos no lo detecta claramente, debe considerarlo ausente e iniciar la RCP efectuando 30 compresiones torácicas (100 por minuto) e intercalando dos ventilaciones. Sólo se para para comprobar la circulación (pulso) o si la víctima inicia movimientos o respiración espontánea. El punto de masaje se debe localizar en el centro del pecho; los ventiladores con el boca-boca se deben realizar insuflando durante un segundo y comprobando que se eleva el pecho.

6. *Si se dispone de desfibrilador semiautomático: encender el equipo, colocar las palas autoadhesivas en el pecho de la víctima y se han de seguir las instrucciones de los mensajes de voz y de texto hasta que reciba ayuda cualificada.*

Vamos a comprobar las siguientes propiedades:

Propiedad 1: Si una víctima esta grave, se llama a una ambulancia.

Propiedad 2: Es posible salvar la vida de una víctima grave.

Propiedad 3: Siempre que se salva, se ha analizado respiración y pulso.

La traducción de dichas propiedades a CTL lo haremos una vez definida la especificación del sistema de actividades. La siguiente especificación sería un ejemplo de especificación de dicho modelo en un lenguaje específico de dominio:

Items:

Reanimador, Victima, Seguridad, Salvar Victima, Analisis Conciencia, Victima Responde, Victima No Responde, Analisis Respiracion, Victima Respira, Victima No Respira, Telefono, Ambulancia Avisada, Ayudante, Prioridad Ayuda, Tomar Pulso Carotideo, Victima Tiene Pulso, Victima No Tiene Pulso, Via Aerea Libre, Intento Resucitacion, Desfibrilador, Intento Desfibrilacion.

Conditions:

haySeguridad: Free(Items.Seguridad);

noResponde: Free(Items.Victima No Responde);

respiraC: Free(Items.Victima Respira);

victimaNoSalvada: Free(Items.Salvar Victima);

ambulanciaNoAvisada: Not(Free(Items.Ambulancia Avisada));

norespiraC: Free(Items.Victima No Respira);

ambulanciaC: Free(Items.Ambulancia Avisada);

prioridadAyudaC: Free(Items.PrioridadAyuda);

tienePulsoC: Free(Items.Victima Tiene Pulso);

```
  notienePulsoC: Free(Items.Victima No Tiene Pulso);
  viaAereaLibreC: Free(Items.via Aerea Libre);
  rcpFallidaC: Finalized(Activities.RCP Fallida);
Activities:
 "Garantizar Seguridad"{
    Subjects: Reanimador;
    Objects: Victima;
    Objectives: Seguridad;
    ItemsToGenerate: Seguridad;
 }
 "Comprobar Conciencia"{
    Subjects: Reanimador;
    Outcomes: Analisis Conciencia;
    ItemsToGenerate: Analisis Conciencia;
    Conditions: haySeguridad;
 }
 "Victima Responde"{
    Subjects: Victima;
    Objects: Analisis Conciencia;
    Outcomes: Victima Responde;
    ItemsToDispose: Analisis Conciencia;
    ItemsToGenerate: Victima Responde, Salvar Victima;
 }
 "Victima No Responde"{
    Subjects: Victima;
    Objects: Analisis Conciencia;
    Outcomes: Victima No Responde;
    ItemsToDispose: Analisis Conciencia;
    ItemsToGenerate: Victima No Responde;
 }
 "Comprobar Respiracion"{
    Subjects: Reanimador;
```

```
  Outcomes: Analisis Respiracion;
  ItemsToGenerate: Analisis Respiracion;
  Conditions: haySeguridad, noResponde;
}
"Victima Respira"{
  Subjects: Victima;
  ItemsToDispose: Analisis Respiracion;
  ItemsToGenerate: Victima Respira;
}
"Victima No Respira"{
  Subjects: Victima;
  ItemsToDispose: Analisis Respiracion;
  ItemsToGenerate: Victima No Respira;
}
"Colocar en Posicion de Seguridad"{
  Subjects: Reanimador;
  Objects: Victima;
  Objectives: Salvar Victima;
  ItemsToGenerate: Salvar Victima;
  Conditions: haySeguridad, respiraC, victimaNoSalvada;
}
"Llamar 112"{
  Subjects: Reanimador;
  Objectives: Ambulancia Avisada;
  Tools: Telefono;
  ItemsToGenerate: Ambulancia Avisada;
  Conditions: haySeguridad,respiraC,ambulanciaNoAvisada;
}
"Llamar inmediatamente al 112"{
  Subjects: Reanimador;
  Objectives: Ambulancia Avisada;
```

```
  Tools: Telefono;
  ItemsToGenerate: Ambulancia Avisada;
  Conditions: haySeguridad,norespiraC;
}
"Pedir Ayuda"{
  Subjects: Reanimador;
  Objects: Ayudante;
  Objectives: PrioridadAyuda;
  ItemsToGenerate: PrioridadAyuda;
  Conditions: haySeguridad,norespiraC,ambulanciaC;
}
"Buscar Pulso Carotideo"{
  Subjects: Reanimador;
  Objects: Victima;
  Objectives:Tomar Pulso Carotideo;
  ItemsToGenerate: PrioridadAyuda,Tomar Pulso Carotideo;
  Conditions: haySeguridad, norespiraC, prioridadAyudaC;
}
"Victima Tiene Pulso"{
  Subjects: Victima;
  ItemsToDispose: Tomar Pulso Carotideo;
  ItemsToGenerate: Vitima Tiene Pulso;
}
"Victima Tiene Pulso"{
  Subjects: Victima;
  ItemsToDispose: Tomar Pulso Carotideo;
  ItemsToGenerate: Vitima No Tiene Pulso;
}
"Limpiar Via Aerea"{
  Subjects: Reanimador;
  Objects: Victima;
  Objectives: via Aerea Libre;
```

```
  ItemsToGenerate: via Aerea Libre;
  Conditions: haySeguridad,norespiraC,prioridadAyudaC,tienePulsoC;
}
"Reanimar Respiracion"{
  Subjects: Reanimador;
  Objects: Victima;
  Objectives: Salvar Victima;
  ItemsToDispose: Victima No Respira;
  ItemsToGenerate: Salvar Victima, Victima Respira;
  Conditions: haySeguridad, norespiraC, prioridadAyudaC, tienePulsoC, viaAereaLibreC;
}
"Hacer RCP"{
  Subjects: Reanimador;
  Objects: Victima;
  Objectives: Intento Resucitacion;
  ItemsToGenerate: Intento Resucitacion;
  Conditions: haySeguridad, norespiraC, prioridadAyudaC, notienePulsoC;
}
"Resucita Por RCP"{
  Subjects: Victima;
  Outcomes: Salvar Victima, Victima Respira, Victima Tiene Pulso;
  ItemsToDispose: Intento Resucitacion, Victima No Respira, Victima No Tiene Pulso;
  ItemsToGenerate: Salvar Victima, Victima Respira, Victima Tiene Pulso;
}
"RCP Fallida"{
  Subjects: Victima;
  ItemsToDispose: Intento Resucitacion;
}
"Desfibrilar"{
  Subjects: Reanimador;
  Objects: Victima;
```

```
    Objectives: IntentoDesfibrilacion;
    Tools: Desfibrilador;
    ItemsToGenerate: IntentoDesfibrilacion;
    Conditions: haySeguridad, norespiraC, prioridadAyudaC, notienePulsoC, rcpFallidaC;
  }
  "Resucita Por Desfibrilador"{
    Subjects: Victima;
    Outcomes: Salvar Victima, Victima Respira, Victima Tiene Pulso;
    ItemsToDispose: IntentoDesfibrilacion, Victima No Respira, Victima No Tiene Pulso;
    ItemsToGenerate: Salvar Victima, Victima Respira, Victima Tiene Pulso;
  }
  "Desfibrilacion Fallida"{
    Subjects: Victima;
    ItemsToDispose: IntentoDesfibrilacion;
  }
```

Como se ha visto, en este ejemplo particular no se debe utilizar jerarquía entre actividades ya que toda actividad realizada por el reanimador necesita que la víctima responda a estímulos, pero cada actividad del sistema de actividades es determinista, por lo que las respuestas de la víctima deben codificarse con dos actividades distintas. El problema surge en que si se realiza una actividad de respuesta positiva, la actividad negativa correspondiente no se puede nunca realizar y la actividad padre de ésta nunca terminará.

La traducción de propiedades a CTL se ven afectadas por esto, aunque siguen teniendo lógica. Por ejemplo, para determinar que una persona está grave, existe siempre un tiempo de incertidumbre hasta que se le examina la respiración y el pulso. Por lo tanto se toman proposiciones positivas como negativas, ya que para saber que la víctima no respira no se puede tomar la negación de que el ítem “Víctima respira” está libre, puesto que eso es cierto en todos los momentos de incertidumbre. Eso llevaría a que una víctima estaría grave sin haber comprobado nada. En este ejemplo se han tenido en cuenta estos detalles. Las propiedades se traducirían a las siguientes fórmulas CTL:

Proposiciones usadas: noRespira, noTienePulso, llamaAmbulancia, respira (como proposición de actividad realizada), tienePulso, propSalvada, respiraI (como proposición de Item).

1) Propiedad 1: "Si una víctima esta grave, se llama a una ambulancia."

La propiedad quiere indicar que cada vez que se detecta que la víctima está grave, el siguiente paso inmediato es llamar a una ambulancia. Pero antes de determinar que la víctima está grave, hay un tiempo de incertidumbre en el cual no se sabe si respira o no

y tampoco se sabe si tiene pulso o no. No se puede determinar que una víctima está grave si no se la ha examinado, así que se crea una fórmula auxiliar que codifica dicha incertidumbre. Antes del estado en que se determina que la víctima está grave, todos los estados anteriores son de incertidumbre, por lo que la fórmula se codifica como que no existe un camino en el que hay estados de incertidumbre hasta que se llega a un estado en el que la víctima está grave y tiene una transición a un estado en el que no llama a la ambulancia.

a) Fórmulas auxiliares (en código java):

```
Formula grave = new Or(noRespira,noTienePulso);

Formula siguienteLlamaAmbulancia = new AX(llamaAmbulancia);

Formula incertidumbre = new Not(new Or(new Or(noRespira,respira),
                                new Or(noTienePulso,tienePulso)));

Formula graveynollama = new And(grave,new Not(siguienteLlamaAmbulancia));
```

b) La propiedad se traduce en la siguiente fórmula:

```
Formula formula = new Not(new EU(incertidumbre,graveynollama));
```

2) Propiedad 2: "Es posible salvar la vida de una víctima grave."

a) Fórmulas auxiliares:

```
Formula nopropSalvada = new Not(propSalvada);

Formula graveymastardesesalva = new And(grave,new EU(nopropSalvada,propSalvada));
```

b) La segunda propiedad se traduce en la siguiente fórmula:

```
Formula formula  = new EU(incertidumbre, graveymastardesesalva);
```

3) Propiedad 3: "Siempre que se salva, se ha analizado respiración y pulso."

a) Fórmulas auxiliares:

```
Formula signos = new And(respiraI, tienepulso);
```

b) La tercera propiedad se traduce en la siguiente fórmula:

```
formula = new AU(nopropSalvada,signos);
```

VI. Desarrollo del proyecto.

VI.1 Organización y trabajo

El método de organización temporal seguido es parecido al proceso software de *Extreme Programming* [24, 25], puesto que es adecuado a la disponibilidad que podíamos ofrecer como equipo de desarrollo. La implementación concreta del proceso ha sido como sigue.

Se han realizado reuniones semanales con el *máster* del proyecto, que eran los docentes que lo supervisan. Estas reuniones duraban un mínimo de treinta minutos, pero podían extenderse más allá de una hora dependiendo de los objetivos, problemas y análisis de anomalías hora tratar.

En cuanto al grupo de desarrollo, nos concentrábamos en reuniones semanales de puesta en común. En estas reuniones analizábamos el comportamiento y funcionamiento de los módulos que estaban bajo desarrollo bien en equipo o en parejas.

Con la distribución de tareas, conseguimos bastante autonomía a la hora de desarrollar individualmente, lo cual resulto ser esencial dado que la dedicación semanal variaba según las restricciones de disponibilidad de los miembros del equipo.

VI.2 Requisitos del proyecto

Tal y como fueron planteados en la especificación del proyecto de la asignatura de Sistemas Informáticos, y en las reuniones iniciales con los docentes, los requisitos de este proyecto son los siguientes.

- Debe modelarse un *framework* que permita verificar una fórmula en sistemas de transiciones basados en semánticas operacionales para lenguajes específicos de dominio.
- El modelo deberá ser accesible mediante un intérprete que proveerá estados iniciales y transiciones.
- Los algoritmos de *Model Checking* deben ser modulares y extensibles.
- La respuesta de los algoritmos puede ser cierto, falso, o indefinido si no hay recursos suficientes.
- Si la respuesta es falso, se proveerá un contraejemplo, no necesariamente mínimo, en el que se pueda comprobar que la fórmula es falsa.
- Los contraejemplos deberán tener estructura de grafo.
- Se deberá definir las interfaces necesarias para poder navegar por los contraejemplos y visualizarlos, de tal manera que el usuario pueda estudiar sus contraejemplos en varios formatos diferentes.

VI.3 Estructura del framework

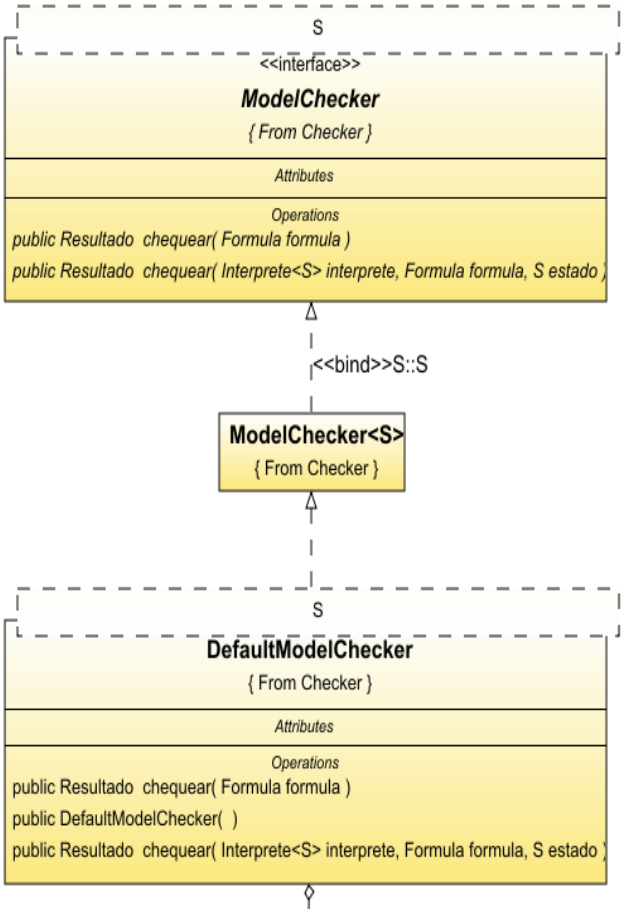
VI.3.1 Checker

Los elementos necesario para el motor de chequeo están almacenados en el paquete "ucm.si.Checker"

ModelChecker: es la interfaz para crear diferentes algoritmos de *Model Checking*.

DefaultModelChecker: (hereda de *ModelChecker*) es un *Model Checker* por defecto que usa lógica CTL para el proceso de chequeo. El algoritmo que usa para la evaluación de las fórmulas se ha detallado en la sección 4.8 del apartado III (III.5.8).

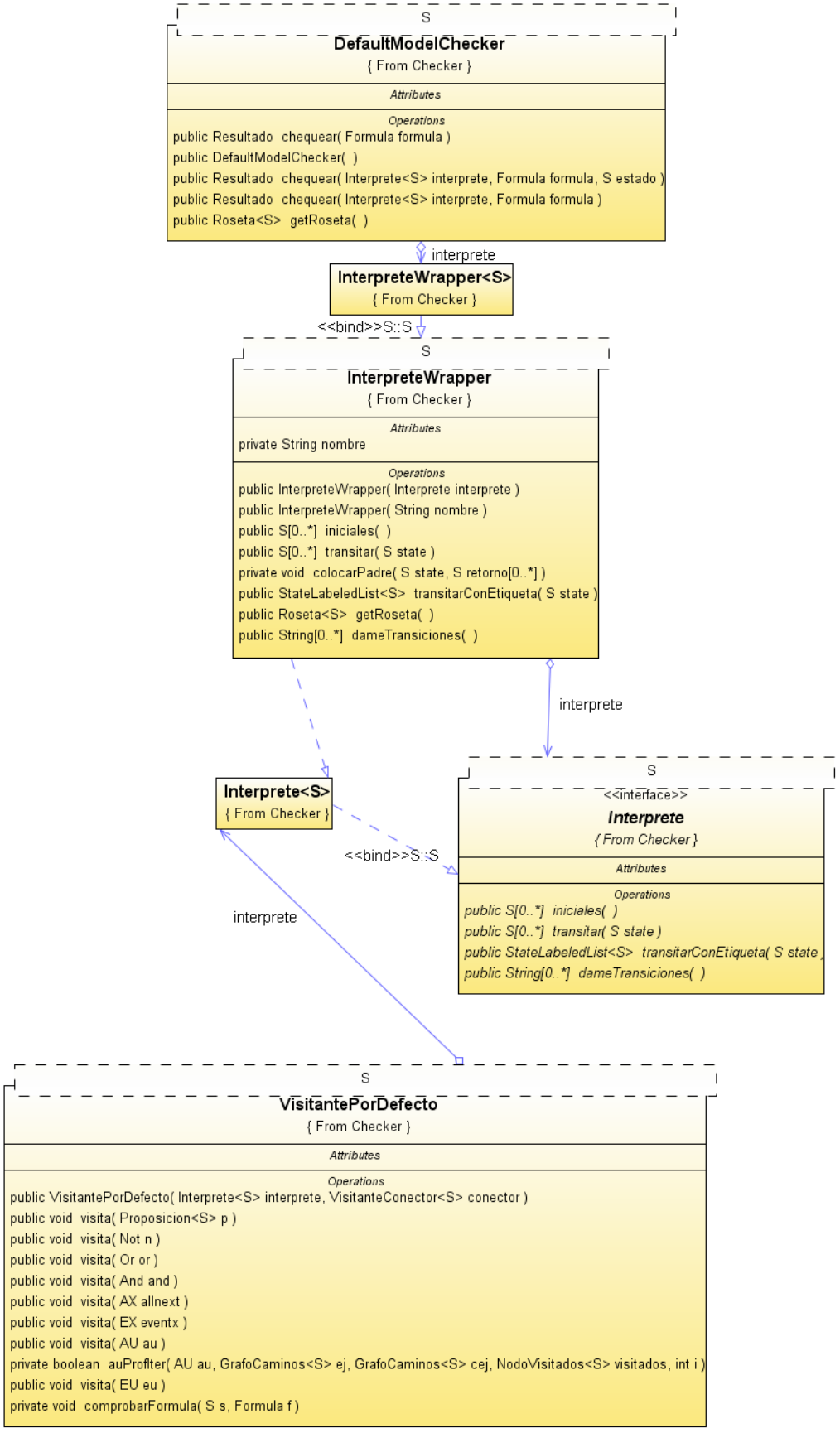
Interprete: es la interfaz que define los métodos necesarios para construir diferentes intérpretes. El intérprete se encarga de generar los estados iniciales por defecto y, dado un estado, devolver el conjunto de estados a los que transita.



IntérpreteWrapper: La clase es un *wrapper* que envuelve el intérprete, que se le pasa en la creación. Con este objeto pretendemos 3 objetivos:

- Evitar nuevas llamadas sobre el intérprete para estados ya calculados.
- Conseguir que para un estado solo exista una copia en memoria durante la ejecución del proceso de *Model Checking* y navegación por el ejemplo o contraejemplo.
- Guardar la estructura de grafo del mapa de estados y el nombre de las transiciones

(aristas). Es decir, para un estado dado, poder obtener la información sobre los estados a los que puede transitar (hijos) y los estados desde los cuales podemos transitar al estado dado (padres).


S
DefaultModelChecker
{ From Checker }
Attributes
Operations
public Resultado chequear(Formula formula)
public DefaultModelChecker( )
public Resultado chequear(Interprete<S> interprete, Formula formula, S estado)
public Resultado chequear(Interprete<S> interprete, Formula formula)
public Roseta<S> getRoseta( )
interprete
InterpreteWrapper<S>
{ From Checker }
<<bind>>S::S
S
InterpreteWrapper
{ From Checker }
Attributes
private String nombre
Operations
public InterpreteWrapper(Interprete interprete)
public InterpreteWrapper(String nombre)
public S[0..*] iniciales( )
public S[0..*] transitar(S state)
private void colocarPadre(S state, S retorno[0..*])
public StateLabeledList<S> transitarConEtiqueta(S state)
public Roseta<S> getRoseta( )
public String[0..*] dameTransiciones( )
interprete
Interprete<S>
{ From Checker }
<<bind>>S::S
S
<<interface>>
Interprete
{ From Checker }
Attributes
Operations
public S[0..*] iniciales( )
public S[0..*] transitar(S state)
public StateLabeledList<S> transitarConEtiqueta(S state
public String[0..*] dameTransiciones( )
interprete
S
VisitantePorDefecto
{ From Checker }
Attributes
Operations
public VisitantePorDefecto(Interprete<S> interprete, VisitanteConector<S> conector)
public void visita(Proposicion<S> p)
public void visita(Not n)
public void visita(Or or)
public void visita(And and)
public void visita(AX allnext)
public void visita(EX eventx)
public void visita(AU au)
private boolean auProfIter(AU au, GrafoCaminos<S> ej, GrafoCaminos<S> cej, NodoVisitados<S> visitados, int i)
public void visita(EU eu)
private void comprobarFormula(S s, Formula f)

Visitante: es la clase que funciona como la interfaz interna a *DefaultModelChecker*, Las clases que la implementan se encargan de navegar a través de las formulas y realizar el chequeo del modelo a partir de los objetos que se le proporciona al crearse (En ese caso el interprete del modelo y un estado inicial).

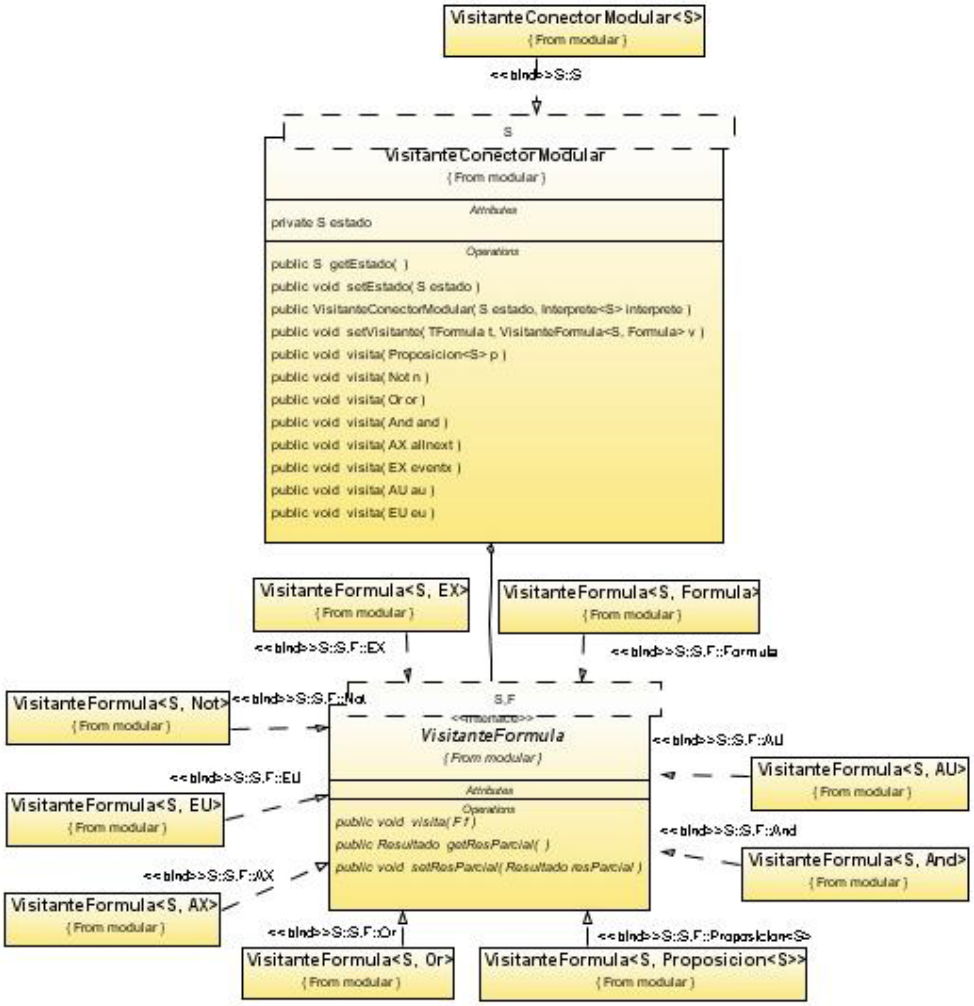
Resultado: Es el objeto que devuelven los *ModelChecker*. En él se almacena el resultado que puede ser True o False, y el ejemplo o contraejemplo asociado a ese resultado.

VI.3.2 Paquete: Checker.modular

El paquete guarda las interfaces necesarias para implementar un algoritmo de *Model Cheking* modular. Este tipo de arquitectura esta pensada como un conjunto de módulos que cada uno de ellos se encarga de evaluar un tipo de operador CTL por separado.

VisitanteConectorModular: Esta es la clase que se encarga de guardar los algoritmos dedicados a cada operador de lógica temporal (en este caso CTL). Estos algoritmos están contenidos en una clase *VisitanteFormula* paramétrica en el estado S y la Formula F. Esta clase también se encarga de seleccionar el *VisitanteFormula* adecuado a cada fórmula.

VisitanteFormula: Esta interfaz define un algoritmo adecuado para un tipo de operador de una lógica.



VI.3.3 Paquete: Checker.modular.defecto

El paquete contiene las implementaciones por defecto de los algoritmos para las primitivas CTL.

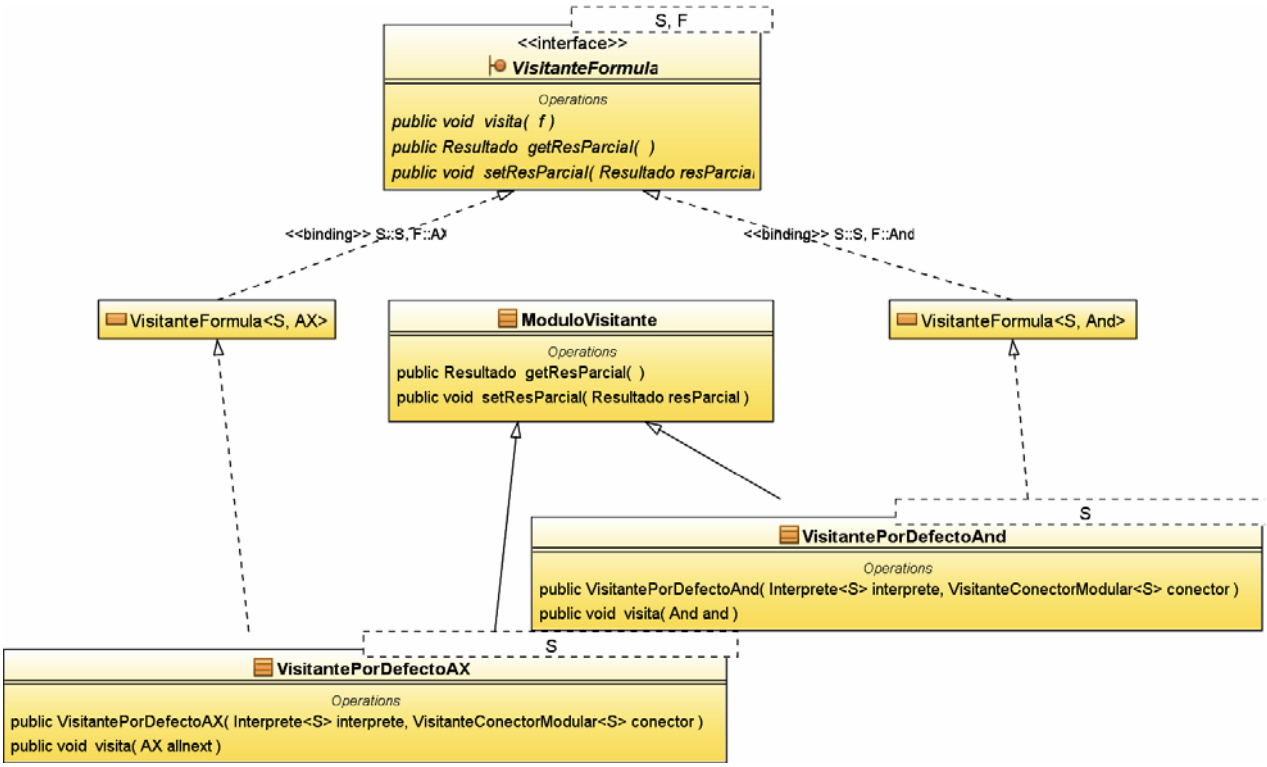
ModuloVisitante: Es una clase de ayuda que define operaciones comunes a todos los Visitantes Modulares por Defecto de este paquete. Las siguientes clases indica cuales son las clases que funcionan como Vistatentes Modulares.

VisitantePorDefectoProposicion: Es una clase que hereda de *ModuloVisitante* y de *VisitanteFormula*. Implementa el algoritmo para evaluar una proposición definida por el desarrollador.

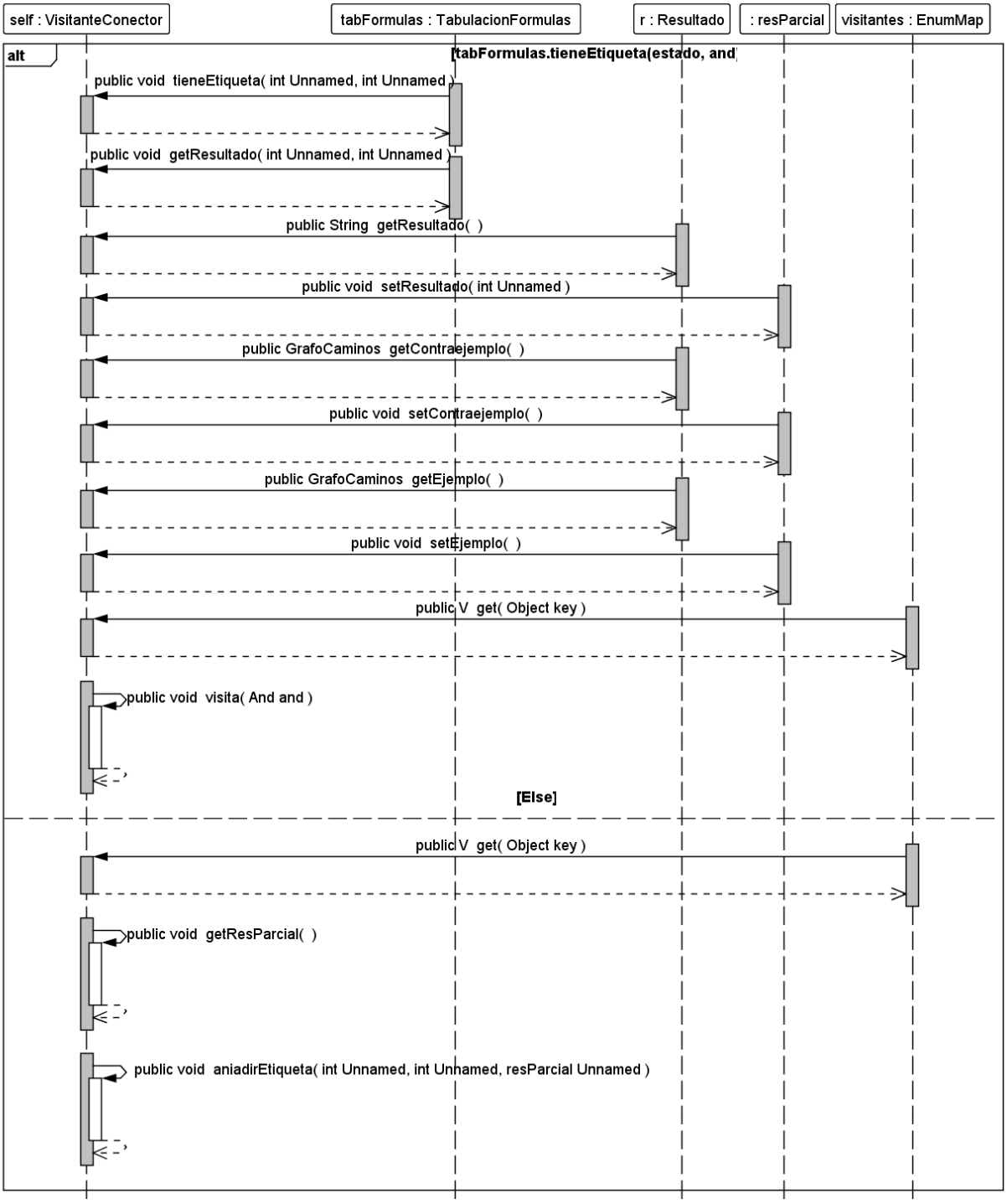
VisitantePorDefectoAU: Es una clase que hereda de *ModuloVisitante* y de *VisitanteFormula*. Implementa el algoritmo para evaluar la operación *Always Until* ($A(f_0 \, U \, f_1)$).

VisitantePorDefectoAX, VisitantePorDefectoEX, VisitantePorDefectoEU, VisitantePorDefectoAnd, VisitantePorDefectoOr, VisitantePorDefectoNot: Son como la clase anterior pero implementan los algoritmos para evaluar respectivamente las operaciones *Always Next* ($AX(f_0)$), *Eventually Next* $EX(f_0)$, *Eventually Until*

$E(f_0 \ U \ f_1)$, and lógica, or lógica y la negación lógica.



El siguiente diagrama de secuencia ejemplifica la interacción del visitante modular con el resto de componentes. Se describe el caso de la evaluación de una operación And (momento en el que se visita un nodo And en el árbol que representa la fórmula CTL a evaluar).

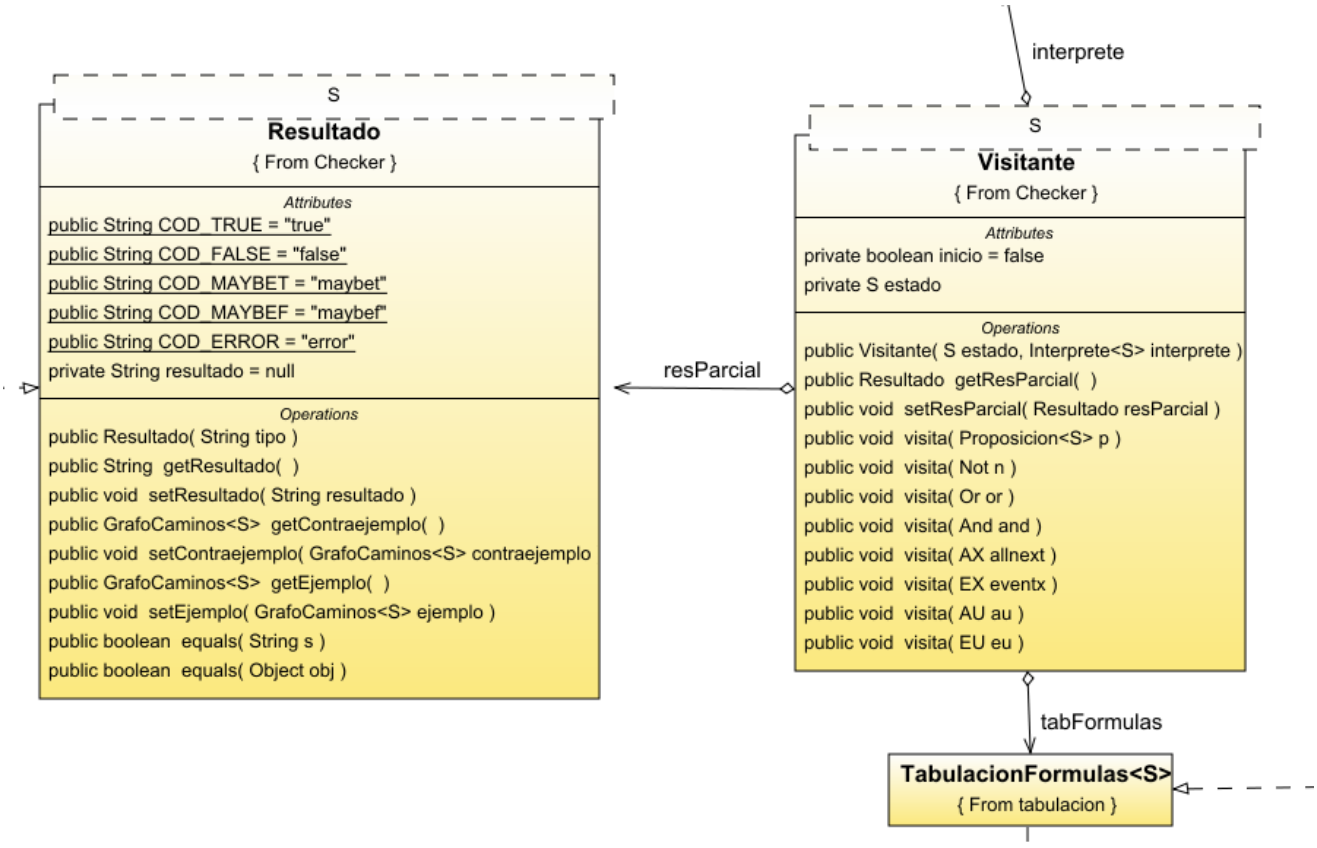

self : VisitanteConector
tabFormulas : TabulacionFormulas
r : Resultado
: resParcial
visitantes : EnumMap
alt
[tabFormulas.tieneEtiqueta(estado, and
public void tieneEtiqueta(int Unnamed, int Unnamed)
public void getResultado(int Unnamed, int Unnamed)
public String getResultado( )
public void setResultado(int Unnamed)
public GrafoCaminos getContraejemplo( )
public void setContraejemplo( )
public GrafoCaminos getEjemplo( )
public void setEjemplo( )
public V get(Object key)
public void visita(And and)
[Else]
public V get(Object key)
public void getResParcial( )
public void aniadirEtiqueta(int Unnamed, int Unnamed, resParcial Unnamed)

VI.3.4 Paquete: Checker.tabulacion

TabulacionFormulas: Es la interfaz a través de la cual se generan los tabuladores de fórmulas. Estos objetos hacen el papel de caché para el algoritmo, guardando para cada estado y para cada fórmula, el objeto *Resultado* correspondiente. Sus objetivos son:

- Evitar probar una misma fórmula varias veces sobre un estado concreto.
- Ayudar al navegador y animador en el recorrido asistido del contraejemplo (o ejemplo) del resultado final.

TabulacionMemSistema: Implementación de *TabulacionFormulas*. Almacena los datos en memoria, usando *TreeMaps* para almacenar el par Estado (Clave) – Lista de *Formulas* con sus *Resultados* correspondientes (implementada como *HashTable*).



VI.3.5 Paquete Laberinto

Este paquete recoge un caso sencillo de ejemplo y demo para el uso del *Framework*. Incluye la implementación de los objetos necesarios para representar un laberinto sencillo en el Framework. Está compuesto por las siguientes interfaces y clases:

Laberinto: implementa la interfaz *checker.Interprete*, Es la encargada de generar los estados (*Posición*) para el caso de un laberinto sencillo.

Posicion: representa los estados para el laberinto.

Final: es una proposición que es verdadera si el estado coincide con dicha posición, y se utiliza para especificar la posición final.

LaberintoPropo: es una proposición que es verdadera si el estado en el laberinto es transitable (es una posición válida).

VI.3.6 Paquete Navegador

AnimadorInterface[I]: clase que funciona como interface para la creación de animadores. Esta clase también se encarga de la interacción con el *Navegador* e implementa los métodos necesarios para la gestión de los eventos generados por éste. Los métodos abstractos definidos son los que se instancian para tratar la llegada de los diferentes eventos de un navegador.

NavegadorInterface[II]: clase que funciona como interface para la creación de navegadores. Incluye funcionalidad para el lanzamiento de eventos a los oyentes que registre.

Navegador: implementación de *NavegadorInterface*. Recibe un ejemplo o un contraejemplo, a partir del cual puede realizar operaciones tales como avanzar, retroceder ir_a (GoTo).

AnimadorBasico: es un objeto ejecutable, que implementa la clase *AnimadorInterface*. Se trata de un animador básico que muestra información e interactúa se realizan a través de consola.

VI.3.7 Paquete util

Contiene clases que proporcionan utilidades diversas. Actualmente solo contiene las clases necesarias para representar un grafo de caminos para un ejemplo o contraejemplo.

TLG: se encarga de guardar y representar un grafo genérico. Se usa como pieza fundamental en muchas de las implementaciones de los *GrafosCaminos*.

GrafoCaminos: es una clase abstracta que funciona como clase tipo o interface para implementar diferentes estrategias de almacenaje y representación de los conjuntos de caminos que son parte de un ejemplo o contraejemplo.

GrafoDoble: actualmente no se usa. Representaba y almacenaba dos objetos de tipo *GrafoCaminos*. La idea era usarlo para mejorar el coste de la unión de dos grafos. El

---

[I] AnimadorInterface y NavegadorInterface, implementan un patrón Observer. AnimadorInterface es el Observer u Oyente de un objeto que implemente NavegadorInterface, y cualquier objeto Navegador Interface generará eventos que tratarán los animadores. Todo esto corre de forma oculta al programador.

[II] AnimadorInterface y NavegadorInterface, implementan un patrón Observer. AnimadorInterface es el Observer u Oyente de un objeto que implemente NavegadorInterface, y cualquier objeto Navegador Interface generará eventos que tratarán los animadores. Todo esto corre de forma oculta al programador.

elevado coste de las demás operaciones desaconseja su uso.

GrafoUnico: inicialmente pensado para ser el contenedor de un único grafo. Actualmente representa una lista n-aria de *GrafosCaminos*. Además incluye un objeto TLG que representa el grafo real según se van calculando las relaciones padres-hijos en la lista de *GrafosCaminos* contenida en la clase.

VI.3.8 Paquete AnimadorGUI

Contiene la carga de clases que componen la GUI (*Graphic User Interface*) que sirve para visualizar los resultados obtenidos.

AnimadorGrafico: esta clase es la base del motor de animación. Implementa la clase *AnimadorInterface*. Se encarga de la interacción entre el *front-end* y el navegador.

Por defecto se utiliza un *Frame* para visualizar el resultado.

FrameAnimador: esta clase es el componente principal de la aplicación de visualización. Se compone de:

- Un panel (*Panel*) destinado a la representación de estado.
- Botones de Avanzar y Retroceder.
- Un *ComboBox* donde el usuario podrá seleccionar el paso que desea avanzar

ComboBoxExtend: extensión de la clase JComboBox.

Contexto: interfaz de contexto bajo el que trabaja el frame.

ContextoLaberinto: contexto del laberinto Contiene la información del laberinto (casillas).

Drawer: es la clase que funciona como interfaz de pintado y repintado del componente *Panel*.

DrawerLaberinto: implementa la interfaz *Drawer* donde se especifica la forma de pintar y repintar el estado, es decir, los botones que representan el laberinto.

Panel: clase abstracta que especifica el uso de la interfaz *Drawer* y el contexto.

PanelJpane: concreción de la clase *Panel*, para el caso del laberinto.

VI.3.9 Paquete basico.ecuaciones

Contiene clases que se encargan de representar una formula cualquiera CTL, además de una interface a partir de la cual programar una proposición cualquiera para el modelo a comprobar. Destaca decir que los elementos de este paquete son parte de la implementación de un patrón visitante.

Formula: es la clase a partir de cual heredan todas las demás clases del paquete.

Operacion: clase intermedia usada para diferenciar entre operaciones y preposiciones. Hereda de *Formula*.

OperacionBinaria y **OperacionUnaria**: clases intermedias usadas para diferenciar

aquellas operaciones binarias de las unarias. Heredan de *Operación*.

Or, And, **AU**, **EU**: representan las operaciones Or, And de la lógica clásica y las operaciones *Always Until* y *Eventually Until* de la lógica temporal CTL respectivamente. Heredan de la clase *OperacionesBinarias*, al ser operaciones lógicas binarias.

Not, **AX**, **EX**: representan la operación lógica Not y las operaciones *Always Next* y *Eventually Next* de la lógica temporal CTL respectivamente. Heredan de la clase *OperacionUnaria*, al ser operaciones lógicas de un solo operando.

Proposición: representa una clase paramétrica que funciona como marco para la programación de cualquier proposición propia al modelo a comprobar. Hereda de la clase *Formula*.

VII. Conclusiones.

Como ya indico Holzmann [26] los retos a los que se enfrenta el futuro del *Model Checking* son varios:

1. *Especificación de Modelos*: Se intenta buscar métodos para especificar modelos más complejos, o modelos pertenecientes a nuevas áreas del conocimiento.

2. *Algoritmos Eficientes*: Se intenta buscar cada vez algoritmos más eficientes que permitan verificar modelos cuyos espacios de búsqueda son cada vez mayores. En esta búsqueda nacen las técnicas de *Model Checking* simbólico.

3. *Extracción de modelos directamente del código fuente*: En la última década se ha propuesto la extracción automática de abstracciones de comportamiento de un programa a partir del código fuente de éste. Con esto se pretende conseguir aplicar técnicas de *Model Checking* directamente al nivel de implementación de un desarrollo software. Esta operación se realiza mediante software de "extracción de modelos", que se encuentra implementado actualmente en algunos *Model Checkers* para lenguajes en plataformas concretas como los sistemas empotrados.

4. *Simplificacion del uso de las técnicas de Model Checking*: Es el último reto propuesto. Consiste en facilitar el uso de las técnicas de *Model Checking* para asi acercar su uso a otros ámbitos y popularizarlo. Aunque estas técnicas han demostrado ser eficientes y útiles en varias áreas de la ingeniería, su uso no es muy popular. La principal razón para ello es la dificultad para definir el modelo a verificar o algunas propiedades en las lógicas temporales existentes.

VII.1Aportaciones.

En el presente proyecto se ha intentado dar una línea para solucionar el problema descrito en los puntos 1 y 4 del anterior apartado. Para ello se persigue simplificar las formas de definir los modelos y sus propiedades; al mismo tiempo se trata de mejorar el uso de la herramienta, facilitando su extensibilidad y mejorando la forma en que se trabaja con sus resultados. Para conseguir el punto 1, se ha apostado por permitir al desarrollador elegir la forma de representar su modelo. Al hacer esto no queda restringido a una forma de representación o a un lenguaje concreto. No se aporta ningún lenguaje de especificación, pero tampoco se obliga al uso de uno en concreto. En la actual implementación el *framework* simplemente necesita un elemento que, dado un estado, informe de los siguientes posibles estados a los que se puede transitar y opcionalmente se proporcione una etiqueta a las transiciones que llevan a estos estados alcanzables. Con esto se pretende conseguir que las especificaciones de los modelos sean flexibles y sencillas. Para conseguir el punto 4 se ha intentado simplificar la definición de los modelos como se indicaba antes, y aportar herramientas que puedan ayudar o hacer atractivo el manejo del framework. Se puede considerar como facilidad principal el proporcionar una herramienta para visualizar el ejemplo o contraejemplo proporcionado por el proceso de verificación. Ya que la herramienta sólo necesita un componente que "dibuje" la representación de un estado.

Para los puntos 2 y 3 no se ha dado una solución explícita, pero sí se da soporte para construir otros algoritmos, mejorar los existentes, o mezclarlos generando un algoritmo

mixto. También se intenta que se pueda modificar o extender fácilmente el *framework*. Una forma de extender el *framework* puede consistir en añadir otras herramientas.

Las características señaladas muestran la filosofía del proyecto. Ésta se centra en proporcionar un framework flexible sobre el cual se puedan extender, implementar y reutilizar diferentes alternativas en el trabajo con *Model Checkers*, y que pueda ser reutilizada en otros ámbitos.

VII.2 Trabajo Futuro

Aunque se has satisfecho los objetivos iniciales del proyecto, siempre quedan aspectos que se pueden mejorar o extender. Algunos de estos puntos son los siguientes:

- Incluir soporte para algunos de los métodos de *Model Checking* simbólico, como puede ser el método clásico mostrado en el apartado 6 del Estado del arte en el se habla del *framework* SMV. De todas formas sería más interesante incluir algunos de los métodos modernos de *model checking* simbólico como BMD[I] [27] o MTBDD[II] [28].

- Añadir más facilidades para facilitar la construcción de intérpretes.

- Mejorar la interfaz de visualización de contra-ejemplos. Ésta puede adaptarse para proporcionar formas más complejas de visualización de contraejemplos.

VII.3Conclusiones Académicas

La realización del presente proyecto ha requerido que hiciésemos uso de las siguientes habilidades que fueron aprendidas durante la carrera:

- Estructura de Datos de la Información: conocimientos sobre las estructura de almacenamiento de datos tales como listas, colas, pilas o tablas hash. Dado el contexto del proyecto, también se han empleado nociones sobre verificación formal.
- Inteligencia Artificial: conocimientos sobre búsquedas en espacios de estados.
- Metodologías y Tecnologías de la Programación: recorrido de grafos backtracking y nociones sobre la eficiencia de los algoritmos.
- Ingeniería del Software: gestión, análisis, y diseño de aplicaciones.

Durante la realización del proyecto los integrantes han aprendido nociones sobre:

- Las teorías y técnicas de *Model Checking*, así como varias de sus herramientas y su funcionamiento.
- Lógicas temporales CTL y LTL.
- Especificación de restricciones y propiedades sobre lógicas temporales.
- Nociones sobre la Teoría de la Actividad.

[I] Binary Moment Diagram: Es un método similar al de los BDD (Binary Decision Diagrams) solo que mejoras sus carencias.

[II] Multi-Terminal Binary Decision Diagrams: Es otra forma de definir diagramas simbólicos, este en concreto ha demostrado ser eficiente en sistemas probabilísticos.

VIII. Referencias Bibliográficas.

[1] E. M. Clarke, O. Grumberg and D. Peled. (1999, January-1999). Model checking. *MIT Press* [unknow]. *1(1),* pp. 1-35. Available: mitpress.mit.edu/

[2] M. ALFONSECA, ***TEORIA DE AUTOMATAS Y LENGUAJES FORMALES***. ,1ª ed., vol. 1, Madrid: McGRAW-HILL, 2007, pp. 464.

[3] R. Stuart and N. Peter, *Artificial Intelligence: A Modern Approach.* ,2nd ed., vol. 1, US: Prentice Hall, 2003, pp. 250-56.

[4] L. Fix, "Fifteen Years of Formal Property Verification in Intel," *25 Years of Model Checking*, pp. 139-144, 2008.

[5] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan and D. L. Dill, "Symbolic model checking for sequential circuit verification," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* vol. 13; 13, pp. 401-424, 1994.

[6] Q. Chen, C. Zhang and S. Zhang, "A Verification Model for Electronic Transaction Protocols," *Advanced Web Technologies and Applications,* pp. 824-833, 2004.

[7] G. J. Holzmann, "The model checker SPIN," *Software Engineering, IEEE Transactions on,* vol. 23, pp. 279-295, 1997.

[8] A. Cimatti, E. Clarke, F. Giunchiglia and M. Roveri, "NuSMV: A New Symbolic Model Verifier," *Computer Aided Verification,* pp. 682-682, 1999.

[9] 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," *Computer Aided Verification,* pp. 241-268, 2002.

[10] E. Mikk, Y. Lakhnech, M. Siegel and G. J. Holzmann, "Implementing statecharts in PROMELA/SPIN," *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on,* pp. 90-101, 1998.

[11] N. Guelfi and A. Mammar, "A formal semantics of timed activity diagrams and its PROMELA translation," *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific,* pp. 8 pp., 2005.

[12] O. R. Ribeiro and J. M. Fernandes, "Translating Synchronous Petri Nets into PROMELA for Verifying Behavioural Properties," *Industrial Embedded Systems, 2007. SIES '07. International Symposium on,* pp. 266-273, 2007.

[13] E. Clarke, "The Birth of Model Checking," *25 Years of Model Checking,* pp. 1-26, 2008.

[14] E. Emerson, "The Beginning of Model Checking: A Personal Perspective," *25 Years of Model Checking,* pp. 27-45, 2008.

[15] E. M. Clarke Jr., "Automatic verification of finite-state concurrent systems," *Logic in Computer Science, 1994. LICS '94. Proceedings. , Symposium on,* pp. 126, 1994.

[16] A. Pnueli, "The temporal semantics of concurrent programs," *Semantics of Concurrent Computation,* pp. 1-20, 1979.

[17] S. Owicki and L. Lamport, "Proving Liveness Properties of Concurrent Programs," *ACM Trans. Program. Lang. Syst.,* vol. 4, pp. 455-495, 1982.

[18] S. Owicki and D. Gries, "Verifying properties of parallel programs: an axiomatic approach," *Commun ACM,* vol. 19, pp. 279-285, 1976.

[19] D. L. Dill and R. Melton, "Annotated Murphi Reference Manua," *No,* vol. 1, pp. 1-1, March, 1993. 1993.

[20] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Logic in Computer Science, 1990. LICS '90, Proceedings. , Fifth Annual IEEE Symposium on e,* pp. 428-439, 1990.

[21] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *Computers, IEEE Transactions on*, vol. C-35; C-35, pp. 677-691, 1986.
[22] P. Wolper, M. Y. Vardi and A. P. Sistla, "Reasoning about infinite computation paths," *Foundations of Computer Science, 1983. , 24th Annual Symposium on*, pp. 185-194, 1983.
[23] W. Visser, K. Havelund, G. Brat and Seungjoon Park, "Model checking programs," *Automated Software Engineering, 2000. Proceedings ASE 2000. the Fifteenth IEEE International Conference on*, pp. 3-11, 2000.
[24] F. Macias, M. Holcombe and M. Gheorghe, "A formal experiment comparing extreme programming with traditional software construction," *Computer Science, 2003. ENC 2003. Proceedings of the Fourth Mexican International Conference on*, pp. 73-80, 2003.
[25] J. Kivi, D. Haydon, J. Hayes, R. Schneider and G. Succi, "Extreme programming: a university team design experience," *Electrical and Computer Engineering, 2000 Canadian Conference on*, vol. 2; 2, pp. 816-820 vol.2, 2000.
[26] G. Holzmann, R. Joshi and A. Groce, "New Challenges in Model Checking," *25 Years of Model Checking*, pp. 65-76, 2008.
[27] Y. C. Randal E. Bryant, "Verification of Arithmetic Circuits with Binary Moment Diagrams," *Design Automation, 1995. DAC '95. 32nd Conference on*, pp. 535-541, 1995.
[28] F. Wang and M. Kwiatkowska, "An MTBDD-Based Implementation of Forward Reachability for Probabilistic Timed Automata," *Automated Technology for Verification and Analysis*, pp. 385-399, 2005.