UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Verificación Automática del Comportamiento de un Robot Asistente

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: León Guerrero, Alejandro
Tutora: Alpuente Frasnedo, María
Directora experimental: Sapiña Sanchis, Julia

Curso 2018-2019

Resumen

Los asistentes robóticos autónomos están diseñados para ofrecer compañía a personas que viven en un ambiente doméstico. Este tipo de robot usa diversos sensores para detectar y adaptarse a cualquier imprevisto en el entorno, como evitar obstáculos u otras personas. El desarrollo industrial de los asistentes robóticos carece actualmente de un contexto coherente que pueda asegurar propiedades críticas como su seguridad. Para que los robots asistentes establezcan interacciones avanzadas con humanos de manera segura y confiable, nuevas herramientas y técnicas deben ser desarrolladas para verificar y validar los asistentes robóticos.

A fin de lograrlo, en este proyecto nos basamos principalmente en Maude, un lenguaje reflexivo de altas prestaciones diseñado para la especificación formal de sistemas, y en su *model checker* que, apoyado en la lógica temporal LTL, nos permitirá verificar de manera automática y exhaustiva las propiedades de interés en nuestro modelo de un sistema robótico. Este sistema es el robot asistente Care-O-Bot, dotado de capacidades inteligentes y con una serie de características que deberemos representar en el lenguaje Maude. Posteriormente, aplicando la técnica de *model checking* se comprobarán diferentes propiedades de alcanzabilidad, seguridad y vivacidad, demostrando de esta forma, con un caso de estudio actual, la potencia y versatilidad de nuestro enfoque.

Palabras clave: Maude, verificación de modelos, robot, modelo

Resum

Els assistents robòtics autònoms estan dissenyats per oferir companyia a persones que viuen en un ambient domèstic. Aquest tipus de robot utilitza diversos sensors per detectar i adaptar-se a qualsevol imprevist en el entorn, com evitar obstacles o altres persones. El desenvolupament industrial dels assistents robòtics manca actualment d'un context coherent que puga garantir propietats crítiques com és la seguretat. Perquè els robots assistents establisquen interaccions avançades amb humans de manera segura y fiable, noves ferramentes i tècniques han de ser desenvolupades per a verificar i validar els assistents robòtics.

A fi de d'aconseguir-ho, en aquest projecte ens basem principalment en Maude, un llenguatge reflexiu d'altes prestacions dissenyat per a l'especificació formal de sistemes, i en el seu *model checker* que, recolzat en la lògica temporal LTL, ens premetrà verificar de manera automàtica i exhaustiva les propietats d'interés en el nostre model d'un sistema robòtic. Aquest sistema és el robot asistent Care-O-Bot, dotat de capacitats intel·ligents i amb una sèrie de caraterístiques que haurem de representar en el llenguatge Maude. Posteriorment, aplicant la tècnica de *model checking* es comprovaran diferents propietats d'alcançabilitat, seguretat y vivacitat, demostrant d'aquesta manera, amb un cas d'estudi actual, la potència y versatilitat del nostre enfocament.

Paraules clau: Maude, verificació de models, robot, model

Abstract

Autonomous robotic assistants are designed to provide companionship for a person living in a typical domestic environment. This type of robot uses several sensors to locate and adapt to any unforeseen in the environment, such as avoiding obstacles or other people. The industrial development of robotic assistants currently lacks a coherent framework that can ensure critical properties such as their safety. For assistant robots to engage

in advanced interactions with humans in a trustworthy and safe manner, new tools and techniques must be developed to verify and validate robotic assistants.

The aim of this work is to help verify the robot's behavior by means of an automatic verification tool. This amounts to the design, realisation and evaluation of a model of the robot's behaviour that is then used as the input model for the automated verifier. First, a model of the robot functionality is specified by a set of control rules that determine how the robot acts in its environment. Then, a model checker will be used to validate the model properties automatically, thus proving the versatility and powerfulness of our approach.

Key words: Maude, model checking, robot, model

Índice general

CAPÍTULO 1
Introducción

La verificación de modelos mediante *model checking* nos permite, de manera automática, comprobar que un sistema formal dado satisface las propiedades de interés mediante el análisis exhaustivo de todos los posibles caminos de ejecución del sistema.

El caso de estudio de este trabajo, al que aplicaremos un software de verificación automática para comprobar la corrección de su modelo, es un robot asistente destinado a ayudar a personas vulnerables por enfermedad, discapacidad o edad. Este robot, llamado Care-O-Bot [2], posee capacidades inteligentes y es capaz de captar señales del entorno, así como órdenes de su usuario para realizar servicios tales como advertir de una llamada a la puerta, recordar un evento programado o hacer compañía a su dueño.

La Universidad de Hertfordshire proporcionó una base de datos con las reglas de control del Care-O-Bot, que fue a su vez transcrita a un modelo expresado en NuSMV input [6]. Nos basamos en este modelo inicial para realizar una especificación formal como una teoría de reescritura escrita en el lenguaje Maude, y posteriormente verificarlo con el *model checker* asociado a este lenguaje.

1.1 Motivación

Este trabajo pretende aplicar la verificación de modelos a un caso práctico característico de los sistemas de inteligencia artificial utilizando el lenguaje Maude. Maude es un lenguaje de altas prestaciones pensado para especificar sistemas, que además dispone de un *model checker* o verificador de modelos propio, lo que lo hace idóneo para esta tarea.

Cada vez es más frecuente el uso de software en toda suerte de dispositivos, desde vehículos o electrodomésticos hasta armamento militar. Este software es cada vez más complejo y más común en nuestras vidas, teniendo a veces nuestra seguridad en sus manos, por lo que se hace necesario garantizar su corrección. Por supuesto, esta garantía es cada vez más difícil de proporcionar, especialmente en los sistemas de inteligencia artificial, los cuales son muy potentes pero no necesariamente confiables a menos que su corrección pueda probarse formalmente [10].

1.2 Objetivos

El principal objetivo de este proyecto es aplicar la verificación automática de modelos a la especificación formal de un sistema robótico. Para ello, antes es necesario desarrollar su modelo en el lenguaje que hemos elegido, extrapolando nuestros conocimientos sobre el sistema y adaptándolos a las características de Maude.

1.3 Estructura de la memoria

Tras este capítulo de introducción, para un mejor entendimiento del trabajo realizado, se presentan los contenidos teóricos necesarios para llevar a cabo este proyecto. El Capítulo 2 corresponde al leguaje Maude. El Capítulo 3 está dedicado al *model checking*. En el Capítulo 4 se explican las características del Care-O-Bot. Posteriormente, en el Capítulo 5 se muestra en detalle el modelo desarrollado de dicho robot. En el Capítulo 6 se describe el proceso de análisis y *model checking* realizados para el mismo. Finalmente, en el Capítulo 7 se presentan las conclusiones obtenidas y se relacionará lo estudiado en este trabajo con los estudios cursados.

CAPÍTULO 2
El lenguaje Maude

Este capítulo consiste en una explicación básica del funcionamiento y la sintaxis del lenguaje Maude. Maude es un lenguaje reflexivo y de altas prestaciones que soporta la programación y la especificación de sistemas tanto en lógica ecuacional como en lógica de reescritura [5]. Se puede encontrar públicamente en `http://maude.cs.illinois.edu`.

Un programa en Maude está constituido por uno o varios módulos, que pueden ser módulos funcionales o módulos de sistema. Los módulos funcionales se diferencian de los módulos de sistema en que éstos últimos además admiten reglas, integrando la lógica de reescritura y permitiendo la transición entre estados.

Los módulos están compuestos de los siguientes elementos [5]:

- *header*, que contiene el identificador del módulo,
- importaciones, en las que se indica cuáles serán los submódulos del módulo actual,
- *sorts*, los nombres de los diferentes tipos de datos,
- *subsorts*, definen la organización jerárquica de los tipos,
- *operators*, que definen las operaciones que se aplicarán a los datos,
- axiomas de membresía, que especifican los términos que tienen un tipo determinado,
- ecuaciones o reglas (sólo en los módulos de sistema) que describen el comportamiento del sistema.

Maude espera que las ecuaciones sean terminantes y confluentes, de manera que un módulo funcional sea determinista. Mientras tanto, las reglas pueden abrir otros caminos de ejecución que no confluyan y además no terminen. Los módulos funcionales comienzan con la palabra `fmod` y terminan con `endfm`, los módulos de sistema comienzan con `mod` y acaban con `endm`.

2.1 Importaciones

Para importar un módulo en otro se utiliza la palabra clave `including` o, abreviado, `inc`, seguida de los nombres de los submódulos separados por un signo «+», de esta manera:

```
including mod1 + mod2 .
```

Esto debe hacerse tras la cabecera del módulo. El espacio y el punto indican el final de la línea y son siempre necesarios en Maude.

Hay otra manera muy común de importar módulos en Maude, que es mediante el uso de la palabra clave `protecting` o `pr`. La ventaja de este método es que evita añadir elementos pertenecientes a nuestro módulo que no puedan ser utilizados o que causen confusión en el submódulo que importamos.

2.2 *Sorts* y *Subsorts*

Un tipo se declara utilizando la palabra `sort` seguida de un identificador, por ejemplo:

```
sort animal .
```

También se pueden declarar varios tipos al mismo tiempo, tantos como se quiera, utilizando la palabra `sorts` seguida de los correspondientes identificadores separados por espacios.

```
sorts animal dog .
```

Destacar que en realidad `sort` y `sorts` son equivalentes, pero se utilizan por convenio para declarar un tipo y para varios, respectivamente.

Los tipos pueden ser organizados jerárquicamente mediante la palabra `subsort`, de esta manera:

```
subsort dog < animal .
```

Se está indicando que el primer tipo es un subtipo del segundo. Igual que en la declaración de los *sorts*, también existe la palabra `subsorts` para definir múltiples subtipos al mismo tiempo:

```
subsorts pitbull cocker < dog < animal .
```

`subsort` y `subsorts` también son equivalentes.

2.3 Operadores

Los operadores son estructuras con una lista de *sorts* como argumentos y uno más como resultado. Un operador se declara con la palabra `op` y con la siguiente estructura:

```
op O : S1 S2 ... Sn -> S [AttrSet] .
```

donde `S1 S2 ... Sn` son los *sorts* de los argumentos del operador `O`, `S` es el tipo del resultado y `AttrSet` es un conjunto de posibles atributos que puede tener. También es posible declarar varios operadores al mismo tiempo que compartan la misma estructura utilizando la palabra `ops`:

```
ops O O2 ... Om : S1 S2 ... Sn -> S [AttrSet] .
```

2.3.1. Atributos

Los atributos son etiquetas que aportan información adicional sobre el operador y se escriben entre corchetes al final de la declaración. Algunos de estos atributos son:

- `ctor`: indica que el operador es un constructor.
- `assoc`: indica que el operador cumple la propiedad asociativa.

- `comm`: indica que el operador cumple la propiedad conmutativa.
- `idem`: indica que el operador cumple la propiedad de idempotencia.
- `id`: permite definir la identidad del operador. El valor de las expresiones con este operador será independiente de la presencia del elemento identidad definido con esta propiedad (e.g., `[id: null]`).

2.3.2. La notación infija

Una característica interesante de los operadores en Maude y que lo dota de una gran flexibilidad sintáctica es la notación infija. Los operadores descritos anteriormente utilizan una notación prefija, porque no se ha indicado lo contrario, con lo cual se escribe primero el nombre del operador seguido de sus argumentos entre paréntesis y separados por comas. Sin embargo, Maude permite definir un operador con guiones bajos en los lugares del nombre en los que se esperan los diferentes argumentos, por ejemplo:

```
op _+_ : Int Int -> Int .
```

De esta manera, en vez de utilizar la notación prefija como sería en `+(2, 3)`, basta con escribir los argumentos donde se encuentran los guiones bajos, como en `2 + 3`.

2.4 Axiomas de membresía

Son declaraciones que, dados un patrón *P* y un *sort S*, especifica que los términos que hacen *matching* con el patrón son del tipo *S*. Respetan el siguiente esquema, precedido por la palabra clave `mb`.

```
mb P : S .
```

También existe una versión condicional en la que han de cumplirse las condiciones (`C1 and C2 and ... and Ck`) para que la expresión pertenezca al tipo especificado. Se utiliza la palabra clave `cmb`.

```
cmb P : S if C1 and C2 and ... and Ck .
```

2.5 Ecuaciones y reglas

En Maude, las ecuaciones y las reglas definen el significado de los operadores y el comportamiento del sistema. Las ecuaciones tienen el siguiente aspecto:

```
eq lhs = rhs .
```

Donde `lhs` y `rhs` hacen referencia a unas expresiones que conforman la parte izquierda y derecha de la regla, respectivamente. Entonces, cuando una expresión coincida con la parte izquierda, será reducida a la de la parte derecha. Las ecuaciones pueden ser condicionadas por una o más expresiones de esta manera:

```
ceq lhs = rhs if C1 and C2 and ... and Ck .
```

En primer lugar, se indica que es una ecuación condicional con la palabra `ceq` seguida de las partes `lhs` y `rhs`, después se escribe `if` precediendo a la o las condiciones. Maude espera que las ecuaciones sean terminantes y confluyentes, esto es, que no generen bucles y que sin importar su orden de aplicación el resultado sea el mismo, de manera determinista.

Las reglas, no obstante, se comportan de forma distinta, a pesar de que la sintaxis sea similar:

```
rl lhs => rhs .
```

También existen las reglas condicionales, indicado por la palabra clave `crl`:

```
crl lhs => rhs if C1 and C2 and ... and Ck .
```

Como se ha dicho anteriormente, las reglas solo pueden utilizarse en módulos de sistema, y no tienen las restricciones de terminación y confluencia de las ecuaciones. Las reglas especifican la dinámica del sistema, mientras que las ecuaciones representan la parte estática.

2.6 Paso de reescritura (*Maude step*)

El paso de reescritura propio de Maude, el *Maude step*, es una transición entre un estado s a otro s' en la que se aplican las ecuaciones y una regla. Para ello se utiliza la siguiente estrategia:

$$s \rightarrow^*_{E,Ax} (s{\downarrow_{E,Ax}}) \rightarrow_{R,Ax} s' \rightarrow^*_{E,Ax} (s'{\downarrow_{E,Ax}})$$

donde E es el conjunto de ecuaciones del programa, Ax es el conjunto de axiomas (asociatividad, conmutatividad y/o identidad), y R el conjunto de reglas.

Primero se usan todas las ecuaciones posibles del conjunto E y los axiomas de Ax hasta alcanzar la forma irreductible del estado s, después se aplica una sola regla del conjunto R para alcanzar el estado s', al que se le vuelven a aplicar las ecuaciones y axiomas para llevarlo a su forma irreductible. De esta manera, el convencional *pattern-matching* se sustituye por un *pattern-matching módulo axiomas*, en el que también se tienen en cuenta estos atributos. Al final, una traza de ejecución de un programa Maude es una secuencia de *Maude steps*.

CAPÍTULO 3
Verificación de modelos (*Model checking*)

El *model checking* es una técnica de verificación ágil para modelos concurrentes, distribuidos y reactivos que nos permite explorar todos los estados de un sistema automáticamente y de manera exhaustiva. A partir de una estructura de Kripke (que en esencia es un grafo etiquetado y dirigido) M, un estado s y una fórmula expresada en lógica temporal f, podemos comprobar mediante *model checking* que $M, s \models f$ [4] o, en caso contrario, encontrar un contraejemplo. Un *model checker* explora todos los estados posibles mediante fuerza bruta, examinando todos los escenarios de manera sistemática. Los algoritmos actuales pueden manejar sistemas con un número de estados explícitos del orden de 10^9 [3], que puede aumentar hasta 10^{120} en *model checkers* simbólicos.

La Figura 3.1 es un esquema que representa el método de *model checking* para la depuración y corrección de sistemas. En nuestro caso, el sistema que hemos modelado es el asistente robótico Care-O-Bot. Por otro lado, hemos identificado los requerimientos que queremos comprobar y los hemos formalizado para especificar propiedades escritas en lógica temporal. Entonces, utilizando el *model checker* de Maude, hemos sido capaces de probar si estas propiedades se satisfacen o, en caso negativo, encontrar algún fallo en nuestro sistema basándonos en el contraejemplo y en las simulaciones de estos casos perjudiciales gracias a la herramienta Anima [1].

Este método formal presenta ciertas ventajas frente a otras técnicas de verificación. Por un lado, los métodos experimentales como la simulación o el *testing* no son exhaustivos, por lo que podrían ignorar defectos fatales en el sistema. Incluso entre los métodos formales, el *model checking* resulta ser el más ligero, rápido y, además, totalmente automático. Sin embargo, su exhaustividad solo puede ser fehaciente en un sistema con un número de estados finito, requiriéndose el uso de técnicas de abstracción para abordar sistemas con infinitos estados.

Las propiedades que se pueden verificar de un modelo pueden clasificarse en diferentes tipos según los aspectos del sistema que contempla.

En primer lugar, las **propiedades de alcanzabilidad** son las que garantizan que un estado deseado puede darse. Son por ejemplo propiedades de alcanzabilidad que un vehículo encuentre la salida de una autopista, que una máquina expendedora sirva una bebida o que un semáforo se ponga en verde.

Por otro lado, las **propiedades de seguridad** son las que garantizan que una situación perjudicial no se va a dar. La seguridad es la inversa de la alcanzabilidad. Una propiedad de seguridad es, por ejemplo, la que garantice la exclusión mutua en la zona crítica del

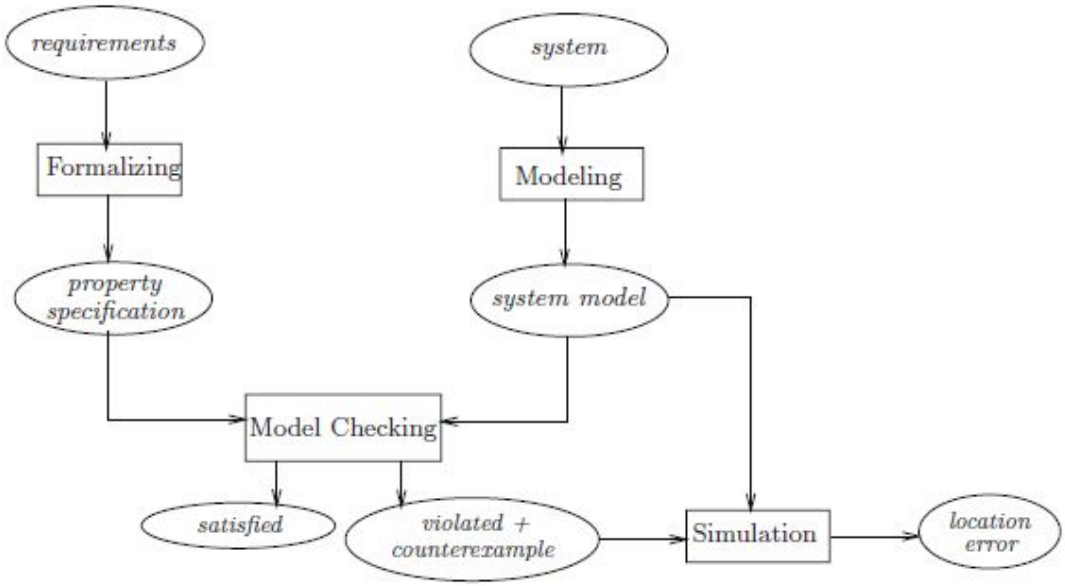

Figura 3.1: Esquema general del método de *model checking* [3].

sistema, buscando un caso en el que varios procesos hayan accedido a ella al mismo tiempo.

Las **propiedades de vivacidad** sirven para asegurar progreso en el sistema, esto es, que si hay una acción habrá una repercusión. Es vivacidad el caso en el que si un proceso pide un recurso, se le proporcionará acceso a él en algún momento. También son de vivacidad las propiedades que garanticen que un proceso no finaliza sin motivo, que ningún proceso muere por inanición y que no se producen interbloqueos.

Por último, las **propiedades de equidad** garantizan que una propiedad se dará un número infinito de veces. Incluye la recurrencia, en la que una propiedad se da cada cierto tiempo infinitas veces; y también la persistencia, en la que una propiedad se da y se mantiene infinitamente. Pueden servir para garantizar la disponibilidad de un recurso infinitas veces, de manera intermitente o a partir de cierto estado.

3.1 Lógica temporal

Las fórmulas que es capaz de verificar un *model checker* requieren de una lógica capaz de especificar las propiedades sobre caminos de ejecución infinitos; ésta es la lógica temporal, un formalismo para describir cambios en el tiempo de un conjunto finito de estados. Concretamente, la lógica temporal que utilizaremos en este proyecto será la *Linear Time Logic* o LTL que introdujo Amir Pnueli, Premio Turing ACM 1996.

La lógica LTL modela una única línea de tiempo, por lo que no es adecuada para sistemas asíncronos, donde son posibles varios futuros, dependiendo de qué componente evoluciona. Los operadores lógicos que utiliza son $\vee$, $\wedge$, $\neg$, $\rightarrow$, $\leftrightarrow$, además de los valores de verdad *true* y *false*. Estos son algunos de los operadores temporales que incorpora:

- Gp (*Globally*): indica que p se cumple durante todo el camino.
- Fp (*Finally*): indica que p se va a cumplir en algún momento futuro.
- Xp (*Next*): indica que p se cumplirá en el estado siguiente.

- pUq (*Until*): indica que p ha de cumplirse hasta que se cumpla q, que deberá hacerlo en algún momento.
- pWq (*Weak until*): indica que p ha de cumplirse al menos hasta que se cumpla q; si q nunca llega, p permanecerá infinitamente.
- pRq (*Release*): indica que q ha de ser cierta hasta y también en el estado en el que p se cumpla; si p nunca se cumple, q permanecerá infinitamente.

3.2 *Model checking* en Maude

La sintaxis LTL se encuentra especificada en el fichero `model-checker.maude`. El *model checker* LTL de Maude permite la verificación en tiempo de ejecución de modelos concurrentes que sean expresados como teorías de reescritura. En cualquier módulo de sistema de Maude, podemos diferenciar dos niveles de especificación [9]:

- Nivel de especificación del sistema, dado por la teoría de reescritura que define el comportamiento del sistema.
- Nivel de especificación de propiedades, dado por una o más propiedades de interés a verificar.

Para comprobar la especificación del sistema, basta con ejecutarla en el entorno de Maude y ver si se comporta de la manera deseada. Sin embargo, para verificar propiedades, no sólo se requiere su especificación en alguna lógica, sinó también un procedimiento que permita comprobarlas en un rango finito de estados. Para llevar a cabo este procedimiento, Maude utiliza los módulos definidos en el fichero `model-checker.maude`.

El *model checker* LTL de Maude permite verificar propiedades de alcanzabilidad, las cuales aseguran que se alcanzará cierto estado; propiedades de seguridad, que garantizan que cierta configuración no se dará; y propiedades de vivacidad, que aseguran que una acción tendrá su reacción. Sin embargo, aunque la equidad sea expresable en LTL, este *model checker* es incapaz de verificar las propiedades de equidad, que garantizan que una situación se dará un número infinito de veces. Esto se debe a que sólo puede trabajar con estados concretos, un número finito de ellos, por lo que no está preparado para hacer una abstracción ecuacional para determinar si en un futuro teórico estas propiedades se cumplirían.

Para aplicar el *model checking* a una especificación formal, necesitamos definir los predicados relevantes, una serie de proposiciones atómicas. Estos predicados son parte de la especificación de propiedades y, junto a la lógica LTL, conforman las propiedades a verificar. Se definen en un módulo aparte, respetando el siguiente patrón:

```
mod M-PREDS is
  protecting M .
  including SATISFACTION .
  subsort MyState < State .
  ops p1 ... pk : -> Prop .
  eq [StatePattern] |= p1 = [Value] .
  [...]
endm
```

donde:

- `M` es el módulo de sistema en el que se ha definido la teoría de reescritura de nuestro modelo,

- `SATISFACTION` es el módulo contenido en `model-checker.maude` en el que se define el operador `|=`,
- `MyState` es un *subsort* arbitrario del tipo `State`, que es el que utiliza el *model checker* para definir un estado,
- los operadores de tipo `Prop` son las proposiciones relevantes que se van a utilizar,
- las ecuaciones son las que otorgan valores (*[Value]*) a esas proposiciones según el estado (*[StatePattern]*), gracias al módulo `SATISFACTION`.

Con estos módulos ya se puede proceder a comprobar propiedades con el *model checker*. En primer lugar, es necesario cargar el fichero `model-checker.maude`, ya que contiene el módulo `SATISFACTION`, la especificación de la lógica LTL y el módulo `MODEL-CHECKER`, entre otros. Se muestra el contenido del fichero en el apéndice B. A continuación, cargamos el fichero que contiene los módulos `M` y `M-PREDS`. Por último, se formulan las propiedades que se deseen verificar con la siguiente estructura:

```
red modelCheck(s, p) .
```

donde `red` es el comando estándar para reducir una expresión utilizando ecuaciones, en este caso la expresión es la función `modelCheck(State, Formula)`, del módulo `MODEL-CHECKER`, que recibe un estado inicial y una fórmula en LTL, que habrá de verificar a partir de ese estado o devolver un contraejemplo.

CAPÍTULO 4
El robot Care-O-Bot

En este capítulo se detallan las características del robot Care-O-Bot, a partir de las cuales se ha tratado de desarrollar un modelo que cumpla los requisitos de seguridad y funcionalidad que se puedan querer verificar.

El robot es capaz de realizar tareas muy diversas [7], tales como acompañar, emitir recordatorios a cierta hora, operar con el microondas, tareas simples de limpieza, servir de apoyo al caminar, etc. Posee una bandeja que puede mover y usar para transportar objetos, un torso articulado, es capaz emitir mensajes de voz, desplazarse a diferentes lugares preestablecidos de la casa e interactuar con los sensores de la domótica.

Internamente, el robot actúa siguiendo una serie de reglas que pueden ser divididas en dos tipos:

- Precondiciones: enunciados proposicionales que pueden ser *true* o *false*. Su función es evaluar si una determinada acción puede ser ejecutada o no.
- Acciones: cualquier acción atómica como un desplazamiento, modificar un valor, quedarse en espera, etc.

Estas reglas conforman órdenes más complejas que son planificadas para su ejecución por el Algoritmo 4.1. Una acción no planificable es aquella acción trivial que forma parte de una orden y que se ejecuta mediante ella.

La variable *scheduled* representa la orden que se encuentra en ejecución, S es un conjunto que se llena con todas las órdenes ejecutables de O. Si no hay órdenes en ejecución y S contiene algún elemento, el más prioritario de estos entrará en ejecución. Sin embargo, si hay otro elemento en S con una prioridad mayor, entonces lo sustituirá. Por último, se comprobará que *scheduled* haya terminado sus acciones para así retirarlo, en caso contrario se ejecutará su siguiente acción.

Algorithm 4.1 Algoritmo de planificación del Care-O-Bot [6].

```
1: Input: a set O of orders
2: scheduled = none
3: while true do
4:    S ← { o ∈ O : preconditions_hold(o) and schedulable(o)}
5:    if (scheduled = none or scheduled has no other action to perform) and S ≠ ∅ then
6:       scheduled = o ∈ S such that priority(o) = max{priority(s) : s ∈ S}
7:    else if ∃o ∈ S.(priority(o) > priority(scheduled)) and interruptible(scheduled) then
8:       scheduled = o
9:    end if
10:   if scheduled has another action to perform then
11:      perform the next action
12:   else
13:      scheduled = none
14:   end if
15: end while
```

CAPÍTULO 5
Especificación formal del sistema Care-O-Bot

En este capítulo se describe el proceso de la especificación del modelo en el lenguaje Maude. En primer lugar, fue necesario familiarizarse con este lenguaje, para lo cuál se utilizó la herramienta de análisis online Anima [1], que muestra de manera visual e intuitiva el árbol de ejecución de una teoría de reescritura dada y, sin olvidar el detalle, también describe la traza de las ecuaciones aplicadas y los cambios que provocan en el sistema. Esta herramienta facilita en gran medida el análisis de alcanzabilidad en etapas tempranas del diseño y permite explorar posibles estados no deseados que puedan deberse a errores en el código.

Para representar los posibles cambios del sistema durante la ejecución, es necesario definir un estado que recoja diversos parámetros. En este caso son:

- Las variables internas del robot.
- El entorno, definido por las posibles señales externas que pueden afectar al comportamiento del robot.
- El planificador, consistente en una lista de órdenes clasificadas por su prioridad y su ejecutabilidad.
- El valor del tiempo, el instante actual.

El constructor derivado de esta idea fue:

```
op _|_|_|_|_ : Robot Env AcOrder Schedule Time -> MyState .
```

donde los *sorts* `Robot`, `Env`, `AcOrder`, `Schedule`, `Time` y `MyState` tienen el significado esperado y que se describe más abajo.

5.1 Parámetro `Robot`

El tipo `Robot` engloba las diferentes variables internas del robot que se han identificado, su constructor fue definido de la siguiente forma:

```
op <_,_,_,_,_> : Place LColor Tray Torso Message -> Robot .
```

- `Place` indica el lugar de la casa en el que se encuentra el robot de entre los que pueden asignarse. Los valores que puede tomar son `kitchen`, `sofa`, `table`, `user`, `charger`, `tv` y `lroom`.
- `LColor` especifica el color de la luz LED que tiene el robot; este valor puede ser `off`, `white` o `yellow`.
- `Tray` informa sobre la posición y el estado de la bandeja que porta el robot, tomando los valores `empty`, `raised` o `lowered`.
- `Torso` indica la posición del torso del robot respecto a su eje, pudiendo ser `left`, `right` o `straight`.
- `Message` informa de que el robot está articulando un mensaje de voz en el instante dado; puede tomar los valores `TV`, `medicine5pm`, `medReminder`, `doorbellRang`, `fridgeDoor` y `noMsg`.

5.2 Entorno y señales

De la palabra *environment*, el parámetro `Env` simula el entorno y todo lo externo al robot, salvo el tiempo, que influye en él. Esto se representa en forma de señales, que se construyen como un par:

```
op {_ _} : Qid Time -> Signal .
```

donde `Qid` es un *quoted identifier*, que permite usar como identificador cualquier cadena precedida del carácter «'», y `Time`, que es un tipo creado *ad hoc* que engloba los naturales, el cero y el -1, y que utilizamos para representar el tiempo. En las señales, el atributo `Time` indica el instante de tiempo en el que se generaron. Las señales son *subsorts* del tipo `Env`, por lo que una sola señal (o ninguna) ya puede constituir un entorno.

La estructura de un `Env` es la siguiente:

```
op null : -> Env .
op (_ _) : Env Env -> Env [assoc id: null] .
```

Se trata de una lista no ordenada de entornos, que bien pueden ser señales, o bien pueden tomar el valor `null`.

5.3 Órdenes

En este modelo, las órdenes no solo representan las acciones que el robot puede realizar, también se utilizan para simular un entorno cambiante. Esta característica nos deja dos tipos de órdenes: las órdenes de acción (`AcOrder`) y las órdenes de señal (`SgOrder`). Ambas son subtipos de `Order`, que a su vez es un subtipo de `Schedule`, el tipo que representa el planificador.

```
subsorts AcOrder SgOrder < Order < Schedule .
```

Las órdenes siguen la siguiente estructura:

```
op [_ _] : Qid Nat -> SgOrder .
op [_ _ _ _ _] : Qid Nat Int Int Bool -> AcOrder .
op noOrd : -> AcOrder .
```

donde el natural (`Nat`) que sigue a su identificador indica la prioridad de la orden en el planificador; los dos enteros (`Int`) de las órdenes de acción representan los ciclos o tiempo que tarda en realizarse y los ciclos restantes para terminar, respectivamente; y el tipo `Bool` es el *flag* que determina si la orden de acción es interrumpible o no. Por ejemplo, en la orden `['moveKitchen 20 5 3 true]`, puede verse que tiene una prioridad igual a 20, una duración de 5 segundos y que ya ha comenzado porque le restan 3 segundos para terminar; nótese también que es una orden interrumpible.

Las órdenes de señal no utilizan varios de estos parámetros porque se consumen en un solo ciclo sin modificar el tiempo. El objetivo de las órdenes de señal es tan solo crear la correspondiente señal en el entorno, simulando, por ejemplo, que ha sonado el timbre.

5.4 Planificador

El planificador, representado por el tipo `Schedule`, es una lista ordenada de elementos del tipo `Schedule`, que pueden constituirse de uno, varios o ningún elemento del tipo `Order`:

```
op null : -> Schedule .
op _;_ : Schedule Schedule -> Schedule [assoc id: null] .
```

Con el propósito de emular el funcionamiento del planificador original, fue necesario escribir una serie de ecuaciones, funciones y reglas que se detallan a continuación.

En primer lugar, en la propia estructura de un estado hay un elemento `AcOrder` separado del resto. Este elemento se reserva para la orden que se encuentra en ejecución. Nótese que no se trata de una `SgOrder`, porque estas no se ejecutan como tal, únicamente se consumirán generando la señal correspondiente. De modo que para empezar necesitamos dos ecuaciones: una que lleve a una orden a ejecución y otra que la retire cuando la tarea haya finalizado.

```
ceq [iniTask] : BOT | ENV | noOrd | AORD ; SCH | T =
                BOT | ENV | AORD | SCH | T
                if (T >= 0) and prec(AORD, ENV, T) .

eq [finishTask] : BOT | ENV | [ID PR INS 0 B] | SCH | T =
                  BOT | ENV | noOrd | SCH | T .
```

Para inicializar una tarea, se toma la primera orden del planificador (que estará previemente ordenado) y se llevará a ejecución, con la condición de que se cumplan sus precondiciones (la función `prec` se explicará más adelante). Para finalizarla, se mira que ya no le queden ciclos por ejecutar y se elimina directamente.

El orden del planificador viene dado por dos funciones, una función `updatefunc`, que se encarga de separar las órdenes que cumplen sus precondiciones de las que no lo hacen; y una orden `ordenarPrio`, que ordena una lista de órdenes de manera ascendente según su prioridad.

```
op updatefunc : Schedule Schedule Schedule Env Time -> Schedule .
   eq [update] : updatefunc( null, SCH2, SCH3, ENV, T ) =
                 ordenarPrio(SCH2) ; ordenarPrio(SCH3) .
   ceq [update] : updatefunc( ORD ; SCH, SCH2, SCH3, ENV, T ) =
                  updatefunc( SCH, SCH2 ; ORD, SCH3, ENV, T )
                  if prec(ORD, ENV, T) .
   ceq [update] : updatefunc( ORD ; SCH, SCH2, SCH3, ENV, T ) =
                  updatefunc( SCH, SCH2, SCH3 ; ORD, ENV, T )
                  if not(prec(ORD, ENV, T)) .
```

```
op ordenarPrio : Schedule -> Schedule .
   ceq [ordenarPrio] : ordenarPrio(SCH ; [ID PR] ; [ID2 PR2] ; SCH2) =
                       ordenarPrio(SCH ; [ID2 PR2] ; [ID PR] ; SCH2)
                       if PR2 < PR .

   ceq [ordenarPrio] : ordenarPrio(SCH ; [ID PR D INS B] ;
                                   [ID2 PR2 D2 INS2 B2] ; SCH2) =
                       ordenarPrio(SCH ; [ID2 PR2 D2 INS2 B2] ;
                                     [ID PR D INS B] ; SCH2)
                       if PR2 < PR .

   ceq [ordenarPrio] : ordenarPrio(SCH ; [ID PR D INS B] ;
                                   [ID2 PR2] ; SCH2) =
                       ordenarPrio(SCH ; [ID2 PR2] ; [ID PR D INS B] ;
                                     SCH2)
                       if PR2 < PR .

   ceq [ordenarPrio] : ordenarPrio(SCH ; [ID PR] ; [ID2 PR2 D2 INS2 B2] ; SCH2)
                       =
                       ordenarPrio(SCH ; [ID2 PR2 D2 INS2 B2] ; [ID PR] ; SCH2)
                       if PR2 < PR .

   eq [ordenarPrio] : ordenarPrio(SCH:Schedule) = SCH:Schedule [owise] .
```

La función *update* se dispara después de cada regla y al comienzo de la ejecución. Trabaja con 3 listas del tipo `Schedule`, que utiliza para desglosar el planificador (la primera lista), en otras dos: una con todas las órdenes que cumplen sus precondiciones y otra con las que no lo hacen, de modo que la cabeza de la lista será una orden ejecutable siempre que haya alguna. Después las ordena por separado llamando a la función `ordenarPrio` y las une en lo que será el nuevo planificador.

Para comprobar que una orden cumple sus precondiciones, se utiliza la función `prec`, que recibe una `Order`, un `Env` y un `Time` y devuelve un `Bool`, indicando si es ejecutable o no.

```
op prec : Order Env Time -> Bool .
```

Hay una ecuación para cada orden, en la que se evalúa, si es pertinente, las señales que haya en el entorno y el instante de tiempo actual para determinar el resultado.

Otro aspecto importante del planificador que se ha modelado es la posibilidad de interrupciones entre las órdenes, para casos en los que llega un orden espontáneamente o se cumple la precondición de otra que estaba a la espera. Esto se consigue mediante la siguiente ecuación:

```
ceq [interruption] :
    BOT | ENV | [ID PR D INS true] | [ID2 PR2 D2 INS2 B2] ; SCH | T =
    BOT | ENV | [ID2 PR2 D2 INS2 B2] | [ID PR D INS true] ; SCH | T
    if (PR2 < PR) and prec([ID2 PR2 D2 INS2 B2], ENV, T) .
```

Si la orden en ejecución es interrumpible y hay una con mayor prioridad en la cabeza de la lista, esta sustituirá a la primera, que irá a la cabeza de la cola en su estado actual. El planificador se reordenará al ejecutarse la primera acción de la nueva orden en curso.

5.5 El parámetro *tiempo*

El tiempo se representa en este modelo con un tipo `Time`, que contiene el conjunto de los naturales.

```
subsort Nat < Time .
```

Este parámetro representa el tiempo en segundos y se reinicia cada veinticuatro horas (86400 segundos). Para esto se usa la función `time24`, que al final de cada regla decide si reiniciar el contador o aumentarlo en un segundo:

```
op time24 : Nat -> Nat .
   ceq time24(T) = 0 if T == 86400 .
   eq time24(T) = T + 1 [owise] .
```

El tiempo se utiliza, por un lado, para cuantificar la evolución del sistema, porque cada cambio de estado se produce en un segundo; y por otro, para las precondiciones u órdenes que tienen restricciones temporales. Particularmente, hay una orden que ha de activarse todos los días a las 17:00, con la que el robot emite un mensaje de voz recordando la hora de la medicina:

```
ceq [5pmMedOrd] :
   BOT | ENV | AORD | SCH | T =
   BOT | ENV | AORD | updatefunc(['say5pm 5 3 3 false] ; SCH, null, null, ENV,
       T) | T
   if (T == 61200) and not(comp5pm(AORD, SCH)) .

op comp5pm : AcOrder Schedule -> Bool .
   eq comp5pm(AORD, SCH ; ['say5pm 5 3 INS false] ; SCH2) = true .
   eq comp5pm(['say5pm 5 3 INS false], SCH) = true .
   eq comp5pm(AORD, SCH) = false [owise] .
```

Con la primera ecuación el sistema añade la orden de dar el mensaje cuando el tiempo llegue a la hora especificada (en segundos) y además no exista otra orden similar en el planificador o en ejecución. Esto sirve tanto como para que no se repita la misma acción acumulada de varios días como para que la ecuación sea terminante.

5.6 Modelado del comportamiento

Las reglas permiten la transición entre estados y hacen avanzar el tiempo. En esta especificación formal hay una o varias reglas para cada orden y, por lo general, en su parte izquierda se busca la orden correspondiente en la zona de ejecución, además de alguna señal en el entorno si fuera necesario; y en la parte derecha se decrementa el contador de esa orden en 1, se consume la señal si se ha consultado, se cambian los parámetros que correspondan en el robot, se llama a la función de *update* para reordenar el planificador y se llama a `time24` para avanzar o reiniciar el tiempo. Por ejemplo, las reglas que definen la acción de moverse a la cocina:

```
crl [moveKitchen] :
< PLACE , LC , TRAY , TORSO , MSSG > | ENV |
['moveKitchen PR 5 INS true] | SCH | T =>
< PLACE , LC , TRAY , TORSO , MSSG > | ENV |
['moveKitchen PR 5 (INS - 1) true] |
updatefunc(SCH, null, null, ENV, time24(T)) | time24(T)
if INS > 1 .
```

```
rl [moveKitchen2] :
< PLACE , LC , TRAY , TORSO , MSSG > | ENV |
['moveKitchen PR 5 1 true] | SCH | T =>
< kitchen , LC , TRAY , TORSO , MSSG > | ENV |
['moveKitchen PR 5 0 true] |
updatefunc(SCH, null, null, ENV, time24(T)) | time24(T) .
```

La primera regla se dedica a decrementar el contador simulando el acercamiento del robot a su destino con el paso del tiempo, entonces la segunda se podrá disparar cuando el contador llegue a 1, cambiando el lugar en el que el robot se encuentra al modificar el valor a `kitchen`.

Cuando no hay ninguna orden que ejecutar, el robot permanecerá inactivo, pero a la espera de nuevas tareas. Para esto hemos usado la orden `idle`, mostrada a continuación:

```
crl [idle] : BOT | ENV | noOrd | ORD ; SCH | T =>
BOT | ENV | noOrd | updatefunc(ORD ; SCH, null, null, ENV, time24(T))
 | time24(T)
if (T < 86399) and (T >= 0) and not(prec(ORD, ENV, T)) .

crl [idle2] : BOT | ENV | noOrd | null | T =>
              BOT | ENV | noOrd | null | time24(T)
              if (T < 86399) and (T >= 0) .
```

Las dos versiones de esta regla están preparadas para dispararse cuando ninguna orden pueda entrar a ejecución (`noOrd`); ya sea porque ninguna cumple su precondición, en el caso de `idle`, o porque el planificador está vacío, en el caso de `idle2`.

CAPÍTULO 6
Análisis y verificación del modelo

Anteriormente hemos visto la clasificación de las propiedades de un sistema en cuatro tipos: alcanzabilidad, seguridad, vivacidad y equidad. Sin embargo, no son propiedades completamente disjuntas, sino que, por ejemplo, la alcanzabilidad se puede interpretar a veces como vivacidad existencial; la alcanzabilidad y la vivacidad repetidas son formas de equidad; la vivacidad y la seguridad son complementarias; y cualquier propiedad de un sistema reactivo se puede expresar como la conjunción de una propiedad de vivacidad y una de seguridad. Por esto último, y dado que el *LTL model checker* es incapaz de verificar la equidad, nos centraremos especialmente en las propiedades de seguridad y vivacidad.

6.1 Análisis de alcanzabilidad

Antes de verificar otras propiedades, dedicaremos cierta atención al análisis de la alcanzabilidad, pues es útil para comprobar directamente que se alcanzan los estados deseados y, de la misma manera, no se alcanza ningún estado perjudicial. La herramienta online Anima [1] permite explorar visualmente el conjunto de estados, pero para una verificación más exhaustiva se ha utilizado el comando `search` de Maude, que dado un estado inicial y otro objetivo busca los diferentes caminos que pueden llegar del primero al segundo.

Por ejemplo, podemos comprobar que, dado un estado inicial como el siguiente, en el que se crea la señal de que ha sonado el timbre, en algún momento el robot reproduce el mensaje de aviso.

```
search in CARE-O-BOT : < sofa,off,lowered,straight,noMsg > | null | noOrd | [
    'moveUser 2 5 5 true] ; ['moveKitchen 20 5 5 true] ; ['moveTable 80 5 5
    true] ; ['moveTV 50 5 5 true] ; ['doorbell 1] ; ['sayDB 1 3 3 false] ; [
    'sayFrid 1 3 3 false] ; ['fridgeAlert 29] | -1 =>* < PLACE,LC,TRAY,TORSO,
    doorbellRang > | ENV | AORD | SCH | T .

Solution 1 (state 12)
states: 13  rewrites: 672 in -25749077661069ms cpu (4ms real) (~
    rewrites/second)
PLACE --> kitchen
LC --> off
TRAY --> lowered
TORSO --> straight
ENV --> (null).Env
AORD --> ['sayDB 1 3 2 false]
SCH --> ['fridgeAlert 29] ; ['moveTV 50 5 5 true] ; ['moveTable 80 5 5 true] ;
    ['sayFrid 1 3 3 false]
T --> 11
```

```
Solution 2 (state 13)
states: 14  rewrites: 705 in -25749077661069ms cpu (12ms real) (~
    rewrites/second)
PLACE --> kitchen
LC --> off
TRAY --> lowered
TORSO --> straight
ENV --> (null).Env
AORD --> ['sayDB 1 3 1 false]
SCH --> ['fridgeAlert 29] ; ['moveTV 50 5 5 true] ; ['moveTable 80 5 5 true] ;
    ['sayFrid 1 3 3 false]
T --> 12

Solution 3 (state 14)
states: 15  rewrites: 743 in -25749077661069ms cpu (23ms real) (~
    rewrites/second)
PLACE --> kitchen
LC --> off
TRAY --> lowered
TORSO --> straight
ENV --> (null).Env
AORD --> noOrd
SCH --> ['fridgeAlert 29] ; ['moveTV 50 5 5 true] ; ['moveTable 80 5 5 true] ;
    ['sayFrid 1 3 3 false]
T --> 13

No more solutions.
states: 86402  rewrites: 1038276 in -25749077661069ms cpu (1577ms real) (~
    rewrites/second)
```

Efectivamente, se encuentran tres estados alcanzables desde el estado inicial que cumplen que el robot está avisando del timbre. Sin embargo, este análisis de alcanzabilidad no permite establecer una relación de causalidad entre que se haya generado la señal de timbre y que el robot haya emitido el mensaje. Esto correspondería a una propiedad de vivacidad, en la que se tiene una acción (suena el timbre) que produce una reacción (el robot avisa). Este es un ejemplo con doble propósito, por un lado, mostrar que la alcanzabilidad pura se puede verificar simplemente ejecutando y buscando los estados de interés y, por otro lado, que un análisis de alcanzabilidad no es suficiente para determinar la corrección de un modelo.

6.2 *Model checking*

En esta sección utilizaremos el *LTL model checker* de Maude para verificar una serie de propiedades de nuestra especificación formal del Care-O-Bot. En primer lugar, hemos creado un módulo de sistema secundario, el módulo `MODEL-PREDS`, en el que se especifican los predicados relevantes para la composición de las propiedades. Este módulo sigue el esquema explicado en la Sección 3.2 y puede consultarse en el anexo A.

Como anotación, para la realización de estas pruebas se han restringido las órdenes *idle* de modo que solo lleguen hasta el instante de tiempo 86399, impidiendo así que se complete el ciclo de las 24 horas y se reinicie el día. Esto es porque al *LTL model checker* no se le puede pedir que explore hasta una determinada profundidad el árbol de ejecución, por lo que, si hay un bucle de infinitos estados como es el ciclo de los días, el programa colapsará.

El *model checker* utiliza otra notación en algunos operadores respecto a lo explicado en la Sección 3.1, a saber:

- El símbolo «[]» es equivalente a «G» o *Globally*.
- El símbolo «<>» es equivalente al «F» o *Finally*.
- El símbolo «∼» representa la negación.

Para realizar las pruebas hemos utilizado dos estados iniciales diferentes de nuestro tipo `MyState`, definidos el módulo `MODEL-PREDS` de la siguiente manera:

```
ops init1 init2 : -> MyState .

  eq init1 = < sofa , off , lowered , straight , noMsg > | null | noOrd |
 ['moveUser 2 5 5 true] ; ['moveKitchen 20 5 5 true] ; ['moveTable 80 5 5 true]
 ; ['moveTV 50 5 5 true] ; ['doorbell 1] ; ['sayDB 1 3 3 false] ;
['sayFrid 1 3 3 false] ; ['fridgeAlert 29] | -1 .

  eq init2 = < table , yellow , empty , straight , noMsg > | null | noOrd |
['moveUser 2 5 5 true] ; ['moveTable 80 5 5 true] ; ['sayDB 1 3 3 false] ;
['fridgeAlert 29] ; ['LWhite 25 1 1 true] ; ['trayRai 33 2 2 false] ;
 ['sayTV 17 3 3 false] ; ['torsoL 40 2 2 false] | -1 .
```

6.2.1. Verificación de la alcanzabilidad

A pesar de lo que se dijo anteriormente por la posibilidad de analizar la alcanzabilidad observando la traza de ejecución, hemos querido añadir un ejemplo sobre cómo se puede formular y analizar esta clase de propiedades mediante el *model checker* de Maude.

«El robot puede llegar a la cocina.»

Esta es una propiedad muy simple por la que solo necesitamos formular un predicado `kitchen` en el módulo `MODEL-PREDS` que sea cierto cuando el robot se encuentre en la cocina.

```
eq < kitchen , LC , TRAY , TORSO , MSSG > | ENV | AORD | SCH | T
   |= kitchen = true .
```

Con el predicado podemos formular la propiedad en lógica LTL:

```
<> kitchen
```

Ahora utilizamos el comando de Maude para comprobar la propiedad, tanto con un estado inicial como con el otro, de la siguiente manera:

```
red modelCheck(init1, <> kitchen) .

red modelCheck(init2, <> kitchen) .
```

El resultado, mostrado a continuación, prueba que la propiedad se cumple para el primer caso, pero no para el segundo. Nos abstenemos de enseñar el resultado del segundo porque se trata de un contraejemplo conformado por todos los estados, desde el primero hasta el 86399, en los que el robot no se encuentra en la cocina.

```
Maude> red modelCheck(init1, <> kitchen) .
reduce in MODEL-PREDS : modelCheck(init1, <> kitchen) .
rewrites: 650 in 3ms cpu (3ms real) (0 rewrites/second)
```

```
result Bool: true

Maude> red modelCheck(init2, <> kitchen) .
reduce in MODEL-PREDS : modelCheck(init2, <> kitchen) .
rewrites: 2678669 in 4916 cpu (4916ms real) (~ rewrites/second)
result ModelCheckResult: counterexample(...)
```

6.2.2. Verificación de la seguridad

Las propiedades de seguridad sirven para garantizar que una situación indeseable no se dará. Por ejemplo, en nuestro sistema no queremos que en ningún caso el robot abandone una tarea ininterrumpible sin haberla finalizado, de modo que empezaremos por enunciar, formular y verificar esta propiedad.

«No es alcanzable el estado en el que el robot deje a medias una tarea ininterrumpible.»

Esta propiedad es de seguridad condicional, en la que no puede interrumpirse un proceso si no cumple la condición de ser interrumpible. Un predicado `interrupted`, que indica que una tarea ha sido interrumpida, y otro predicado `interruptible`, que es cierto cuando la orden puede interrumpirse, nos servirán para formular la propiedad.

```
ceq BOT | ENV | AORD | [ID PR D INS B] ; SCH | T |= interrupted = true
    if D > INS .
eq BOT | ENV | AORD | [ID PR D INS true] ; SCH | T |= interruptible = true .
eq BOT | ENV | AORD | [ID PR D INS false] ; SCH | T |= interruptible = false .
```

Para todos los estados ha de cumplirse que una orden no será interrumpida salvo que sea interrumpible.

```
[] (~ interrupted W interruptible)
```

Los comandos de Maude serán:

```
red modelCheck(init1, [] (~ interrupted W interruptible)) .

red modelCheck(init2, [] (~ interrupted W interruptible)) .
```

El resultado indica que la propiedad se cumple para ambos estados iniciales.

```
Maude> red modelCheck(init1, [] (~ interrupted W interruptible)) .
reduce in MODEL-PREDS : modelCheck(init1, [](~ interrupted W interruptible)) .
rewrites: 1038390 in 2489ms cpu (2489ms real) (0 rewrites/second)
result Bool: true

Maude> red modelCheck(init2, [] (~ interrupted W interruptible)) .
reduce in MODEL-PREDS : modelCheck(init2, [](~ interrupted W interruptible)) .
rewrites: 2851499 in 5469ms cpu (5469ms real) (0 rewrites/second)
result Bool: true
```

La siguiente propiedad de seguridad que queremos verificar es que el robot emita un mensaje de voz cuando está ejecutando una orden del tipo «say».

«Todos los estados satisfacen que el robot no ejecuta una orden 'say' sin decir ningún mensaje.»

La propiedad es demasiado trivial para formularla como una de seguridad condicional, pero funciona de la misma manera. Podemos formularla con un predicado `say`, que indica que hay una orden del tipo «say» en curso, y otro predicado `message`, que es cierto cuando el robot está emitiendo un mensaje.

```
ceq BOT | ENV | [ID PR 3 INS false] | SCH | T |= say = true if INS < 3 .
ceq < PLACE , LC , TRAY , TORSO , MSSG > | ENV | AORD | SCH | T
    |= message = true if (MSSG =/= noMsg) .
```

Para todos los estados ha de cumplirse que, si una orden «say» está en ejecución, se estará diciendo un mensaje.

```
[] (say -> message)
```

Los comandos de Maude para los dos estados iniciales serán:

```
red modelCheck(init1, [] (say -> message)) .

red modelCheck(init2, [] (say -> message)) .
```

El resultado indica que la propiedad se cumple en todos los casos.

```
Maude> red modelCheck(init1, [] (say -> message)) .
reduce in MODEL-PREDS : modelCheck(init1, [](say -> message)) .
rewrites: 1038309 in 2359ms cpu (2359ms real) (0 rewrites/second)
result Bool: true

Maude> red modelCheck(init2, [] (say -> message)) .
reduce in MODEL-PREDS : modelCheck(init2, [](say -> message)) .
rewrites: 2678687 in 5207ms cpu (5207ms real) (0 rewrites/second)
result Bool: true
```

«Cuando se ejecute una tarea, ha de cumplir sus precondiciones.»

Esta es una propiedad importante para el correcto funcionamiento del robot. Podemos utilizar un predicado `running`, que especifique que hay una orden cualquiera en ejecución, y otro predicado `preconditions`, que es cierto cuando la orden en ejecución cumple sus precondiciones.

```
ceq BOT | ENV | AORD | SCH | T |= running = true if (AORD =/= noOrd) .
ceq BOT | ENV | AORD | SCH | T |= preconditions = true if prec(AORD, ENV, T) .
```

Para todos los estados ha de cumplirse que siempre que una orden se encuentre en ejecución, cumplirá sus precondiciones.

```
[] (~ running W preconditions)
```

Los comandos de Maude para los dos estados iniciales serán:

```
red modelCheck(init1, [] (~ running W preconditions)) .

red modelCheck(init2, [] (~ running W preconditions)) .
```

El resultado indica que la propiedad se cumple en todos los casos.

```
Maude> red modelCheck(init1, [] (~ running W preconditions)) .
reduce in MODEL-PREDS : modelCheck(init1, [](~ running W preconditions)) .
rewrites: 1124818 in 2661ms cpu (2661ms real) (0 rewrites/second)
result Bool: true

Maude> red modelCheck(init2, [] (~ running W preconditions)) .
reduce in MODEL-PREDS : modelCheck(init2, [](~ running W preconditions)) .
rewrites: 2765165 in 5259ms cpu (5259ms real) (0 rewrites/second)
result Bool: true
```

«La orden de las 5pm no podrá estar en el planificador antes de las 17:00.»

La orden de las 5pm es el ejemplo de orden programada que ya se vio en la Sección 5.5. Esta orden es un recordatorio que emite un mensaje de voz a la hora de la medicina para el usuario. Es una orden que se añade al planificador a las cinco de la tarde con una determinada prioridad y que será atendida como cualquier otra cuando no haya otras órdenes más prioritarias impidiéndolo. Esto implica que esta propiedad no se cumpliría en el improbable caso de que el robot estuviera ocupado desde las 17:00 hasta las 0:00 con otras órdenes más prioritarias. El predicado `5pmOrder` es cierto si la orden se encuentra en el planificador o en ejecución, mientras que `time17` indica que la hora es igual o superior a las cinco de la tarde (medido en segundos).

```
  ceq BOT | ENV | AORD | SCH | T |= 5pmOrder = true if comp5pm(AORD, SCH) .
  ceq BOT | ENV | AORD | SCH | T |= time17 = true if (T >= 61200) .
```

Para todos los estados ha de cumplirse que no habrá una orden de las 5pm hasta que la hora no sea igual o superior a las 17:00.

```
[] (~ 5pmOrder W time17)
```

Los comandos de Maude para los dos estados iniciales serán:

```
red modelCheck(init1, [] (~ 5pmOrder W time17)) .

red modelCheck(init2, [] (~ 5pmOrder W time17)) .
```

El resultado indica que la propiedad se cumple en los dos casos, aunque no se podría cumplir en todos por lo explicado anteriormente.

```
Maude> red modelCheck(init1, [] (~ 5pmOrder W time17)) .
reduce in MODEL-PREDS : modelCheck(init1, [](~ 5pmOrder W time17)) .
rewrites: 1236330 in 2758ms cpu (2758ms real) (0 rewrites/second)
result Bool: true

Maude> red modelCheck(init2, [] (~ 5pmOrder W time17)) .
reduce in MODEL-PREDS : modelCheck(init2, [](~ 5pmOrder W time17)) .
rewrites: 2876706 in 5366ms cpu (5366ms real) (0 rewrites/second)
result Bool: true
```

6.2.3. Verificación de la vivacidad

Las propiedades de vivacidad sirven para garantizar que si se ha producido un cambio o una acción el sistema, este cambio tendrá una repercusión, de manera que el sistema no se estanca. Un buen ejemplo de vivacidad es la siguiente orden a comprobar:

«Si ha sonado el timbre, el robot avisará.»

En el análisis de la alcanzabilidad se puso como ejemplo la verificación de que el robot avisase del timbre, pero si lo enunciamos como una propiedad de vivacidad podemos establecer esa relación causal entre que el timbre realmente haya sonado con el aviso del robot. Para ello utilizaremos los sigueintes predicados:

```
eq BOT | (ENV {'dbellring T2} ENV2) | AORD | SCH | T |= doorbell = true .
eq BOT | ENV | ['sayDB PR 3 INS false] | SCH | T |= saydb = true .
```

Para todos los estados ha de cumplirse que, si la señal del timbre está activa, eventualmente el robot avisará de ello.

```
[] (doorbell -> <> saydb)
```

Los comandos de Maude serán:

```
red modelCheck(init1, [] (doorbell -> <> saydb)) .

red modelCheck(init2, [] (doorbell -> <> saydb)) .
```

El resultado verifica la propiedad en ambos casos.

```
Maude> red modelCheck(init1, [] (doorbell -> <> saydb)) .
reduce in MODEL-PREDS : modelCheck(init1, [](doorbell -> <> saydb)) .
rewrites: 1038323 in 2478ms cpu (2478ms real) (0 rewrites/second)
result Bool: true

Maude> red modelCheck(init2, [] (doorbell -> <> saydb)) .
reduce in MODEL-PREDS : modelCheck(init2, [](doorbell -> <> saydb)) .
rewrites: 2678692 in 6256ms cpu (6256ms real) (0 rewrites/second)
result Bool: true
```

«En algún momento el robot finalizará todas las tareas que cumplen sus precondiciones.»

Para esta propiedad formulamos un predicado `pendingOrders` que, cuando es cierto, determina que hay una o más órdenes que cumplen sus precondiciones y pueden ser atendidas. Si la orden en ejecución o la que se encuentra en la cabeza del planificador cumple su precondición podemos asumir que el predicado será cierto.

```
ceq BOT | ENV | AORD | SCH | T |= pendingOrders = true
    if (AORD =/= noOrd) .
ceq BOT | ENV | AORD | ORD ; SCH | T |= pendingOrders = true
    if prec(ORD, ENV, T) .
ceq BOT | ENV | AORD | ORD ; SCH | T |= pendingOrders = false
    if not(prec(ORD, ENV, T)) .
```

En todos los estados se cumple que en algún momento no habrá órdenes pendientes:

```
[] (<> ~ pendingOrders)
```

Ahora utilizamos el comando de Maude para comprobar la propiedad, con los dos estados iniciales:

```
red modelCheck(init1, [] (<> ~ pendingOrders)) .

red modelCheck(init2, [] (<> ~ pendingOrders)) .
```

El resultado, mostrado a continuación, prueba que la propiedad se cumple para ambos casos:

```
Maude> red modelCheck(init1, [] (<> ~ pendingOrders)) .
reduce in MODEL-PREDS : modelCheck(init1, []<> ~ pendingOrders) .
rewrites: 1124957 in 2538ms cpu (2538ms real) (0 rewrites/second)
result Bool: true

Maude> red modelCheck(init2, [] (<> ~ pendingOrders)) .
reduce in MODEL-PREDS : modelCheck(init2, []<> ~ pendingOrders) .
rewrites: 3369759 in 6461ms cpu (6461ms real) (0 rewrites/second)
result Bool: true
```

«Si hay órdenes ejecutables pendientes, alguna vez se atenderán.»

En este caso se trata de una propiedad básica de vivacidad que busca garantizar la no inanición del sistema. Para lograr verificarla reutilizaremos los predicados `pendingOrders` y `running`. Se vuelven a mostrar a continuación:

```
  ceq BOT | ENV | AORD | SCH | T |= pendingOrders = true if (AORD =/= noOrd) .
  ceq BOT | ENV | AORD | ORD ; SCH | T |= pendingOrders = true
      if prec(ORD, ENV, T) .
  ceq BOT | ENV | AORD | ORD ; SCH | T |= pendingOrders = false
      if not(prec(ORD, ENV, T)) .
  ceq BOT | ENV | AORD | SCH | T |= running = true if (AORD =/= noOrd) .
```

Siempre que haya órdenes ejecutables pendientes, en algún momento habrá alguna orden en ejecución.

```
[] (pendingOrders -> <> running)
```

Los comandos de Maude serán:

```
red modelCheck(init1, [] (pendingOrders -> <> running)) .

red modelCheck(init2, [] (pendingOrders -> <> running)) .
```

El resultado verifica la propiedad en ambos estados iniciales.

```
Maude> red modelCheck(init1, [] (pendingOrders -> <> running)) .
reduce in MODEL-PREDS : modelCheck(init1, [](pendingOrders -> <> running)) .
rewrites: 1125027 in 2575ms cpu (2575ms real) (0 rewrites/second)
result Bool: true

Maude> red modelCheck(init2, [] (pendingOrders -> <> running)) .
reduce in MODEL-PREDS : modelCheck(init2, [](pendingOrders -> <> running)) .
rewrites: 3369812 in 6368ms cpu (6368ms real) (0 rewrites/second)
result Bool: true
```

«A las 17:00 se creará la orden de las 5pm en el planificador.»

De nuevo una propiedad con restricciones temporales sobre la orden de las 5pm que, sin embargo, no es redundante, pues podría cumplirse la propiedad vista en la sección de seguridad sin que se cumpla esta. Además de reutilizar el predicado `5pmOrder`, añadiremos otro `5pm` que indique que la hora es exactamente las 17:00.

```
ceq BOT | ENV | AORD | SCH | T |= 5pm = true if (T == 61200) .
ceq BOT | ENV | AORD | SCH | T |= 5pmOrder = true if comp5pm(AORD, SCH) .
```

Siempre ha de cumplirse que a las cinco de la tarde la orden de las 5pm se encuentra en el planificador.

```
[] (5pm -> <> 5pmOrder)
```

Los comandos de Maude para los dos estados iniciales serán:

```
red modelCheck(init1, [] (5pm -> <> 5pmOrder)) .

red modelCheck(init2, [] (5pm -> <> 5pmOrder)) .
```

La propiedad queda verificada para ambos casos.

```
Maude> red modelCheck(init1, [] (5pm -> <> 5pmOrder)) .
reduce in MODEL-PREDS : modelCheck(init1, [](5pm -> <> 5pmOrder)) .
rewrites: 1124717 in 2564ms cpu (2564ms real) (0 rewrites/second)
result Bool: true

Maude> red modelCheck(init2, [] (5pm -> <> 5pmOrder)) .
reduce in MODEL-PREDS : modelCheck(init2, [](5pm -> <> 5pmOrder)) .
rewrites: 2765096 in 5484ms cpu (5484ms real) (0 rewrites/second)
result Bool: true
```

CAPÍTULO 7
Conclusiones

En este proyecto se ha desarrollado en lenguaje Maude la especificación formal de un sistema clásico de inteligencia artificial, concretamente, un robot asistente. Posteriormente se ha analizado esta especificación rigurosamente mediante *model checking*.

Respecto a los objetivos formulados en la Sección 1.2, se puede afirmar que éstos se han alcanzado satisfactoriamente. Por un lado, se ha conseguido enlazar los conocimientos sobre el sistema y sobre el lenguaje Maude para crear un modelo funcional que supere las pruebas y cumpla las propiedades que se le exigían. Por otro lado, se han formulado las propiedades en lógica temporal y se ha utilizado de manera exitosa el *model checker* de Maude para verificarlas.

Se ha adquirido una extensa cantidad de conocimientos que incluyen el uso práctico y la teoría del lenguaje Maude, teoría de lenguajes, *model checking*, lógica temporal y la expresión y análisis de propiedades.

Se han encontrado diversas dificultades en el desarrollo de este trabajo. La primera ha sido la asimilación de los conocimientos teóricos necesarios para realizar el proyecto, seguida del aprendizaje del lenguaje Maude desde cero, en lo que hizo falta cierto periodo de tiempo para practicarlo antes de poder aplicarlo a modelar un sistema propio de la inteligencia artificial con él. Por supuesto, a lo largo del desarrollo del modelo hubo problemas relacionados con la representación en Maudel del sistema; al principio por la inexperiencia, más tarde por la complejidad que había tomado el modelo, que dificultaba la identificación de errores. Una vez llegados a la etapa de la verificación automática aparecieron más piedras en el camino porque había propiedades que, por sutilezas, el modelo no cumplía, pero como de eso trata este proyecto, fueron solucionadas refinando el modelo para que se verficasen con éxito todas ellas y, de paso, quedó probada la utilidad del *model checking* en la práctica.

Hay aspectos mejorables en cierto sentido de este trabajo. Por la parte del modelo se podrían haber incluido más órdenes y características del robot, si bien las que se escogieron se consideran suficientemente representativas. También se podría dar cierto grado extra de realismo a las órdenes, como el tener más en cuenta los tiempos para su consecución, para lo que habría sido necesario investigar la disponibilidad de algún otro lenguaje que extienda a Maude (y a su *model checker*) para gestionar tiempo real o que, por ejemplo, se considere si ha habido un desplazamiento a otro lugar cuando el robot reanuda una orden de moverse que había sido interrumpida. Estas son mejoras que, sin embargo, no afectan a la representatividad del sistema diseñado, y que han sido omitidas en pos de una mejor definición del planificador y las capacidades inteligentes del sistema. Por la parte del *model checking*, destacar que a pesar de que el *model checker* que se ha utilizado es incapaz de verificar las propiedades de equidad (*fairness*), existen *model checkers* para

Maude que sí son capaces de hacerlo, como el *LTLR model checker* [8]. Estas mejoras se tendrán en cuenta en posibles proyectos futuros relacionados con este trabajo.

Por último, a título personal, hablaré de la relación del trabajo que se ha realizado con los estudios que he cursado. He encontrado diversas dificultades en el desarrollo que, no obstante, me han enriquecido profesionalmente. Pertenezco a la rama de ingeniería de computadores, y estoy interesado en la informática aplicada a la automática industrial y la robótica. Esto me ha dificultado la comprensión de una teoría de lenguajes y lógica de la que no tenía apenas fundamento, y he tenido que afrontar el aprendizaje del lenguaje Maude y del uso de un *model checker*. Como futuro profesional, creo que ser capaz de especificar un modelo a partir de la información de un sistema y verificar las propiedades que debe cumplir. Puede resultarme de utilidad en mi campo y me da otro punto de vista desde el que entender un sistema y enfrentarme a los problemas de seguridad, corrección y confianza en el mismo.

Bibliografía

[1] M. Alpuente, D. Ballis, F. Frechina, J. Sapiña. Exploring Conditional Rewriting Logic Computations. *Journal of Symbolic Computation,* 69:3-39, 2015.

[2] Página web del producto Care-O-Bot. https://www.care-o-bot.de/en/care-o-bot-4.html

[3] Christel Baier y Joost-Pieter Katoen. *Principles of Model Checking.* MIT Press, enero 2008.

[4] Edmund M. Clarke, E. Allen Emerson, Joseph Sifakis. *Model Checking: Algorithmic Verification and Debugging. Communications of the ACM*, noviembre 2009, vol. 52, nº 11.

[5] M. Clavel, F. Durán, S. Eker, S. Escobar, P.Lincoln, N. Martí-Oliet, J. Messeguer, C. Talcott. *Maude Manual (Version 2.7.1).* http://maude.lcc.uma.es/manual271/maude-manual.html

[6] Paul Gainer. *Computer Science BSc (Hons) Dissertation: Verification for a Robotic Assistant.* Consultado en https://intranet.csc.liv.ac.uk/research/techreports/tr2017/ulcs-17-003.pdf

[7] M. Hans, B. Graf, R.D. Schraft. *Robotic Home Assistant Care-O-bot: Past - Present - Future* Fraunhofer Institute for Manufacturing Engineering and Automation (IPA) Consultado en http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.3350&rep=rep1&type=pdf

[8] Página web del *LTLR model checker* de Maude. http://maude.cs.illinois.edu/tools/tlr/

[9] Guillermo Roselló Gil, María Alpuente Frasnedo, Julia Sapiña Sanchis. *Formal Modeling of Security Policies for Mobile Access Solutions.* Consultado en https://riunet.upv.es/handle/10251/90225

[10] Paul Scharre. *The Real Dangers of an AI Arms Race. Foreign Affairs*, mayo/junio, 2019. También en https://www.foreignaffairs.com/articles/2019-04-16/killer-apps

APÉNDICE A
Código fuente - fichero careobot.maude

```
load model-checker.maude .
mod CARE-O-BOT is inc INT + QID .

  sorts Place MyState LColor Message Robot Time Tray Torso Signal Env Order
        Schedule SgOrder AcOrder .
  subsort Nat < Time .
  subsort Signal < Env .
  subsorts AcOrder SgOrder < Order < Schedule .

  op <_,_,_,_,_> : Place LColor Tray Torso Message -> Robot .
  op _|_|_|_|_ : Robot Env AcOrder Schedule Time -> MyState .

  op [_ _] : Qid Nat -> SgOrder .
  op [_ _ _ _ _] : Qid Nat Int Int Bool -> AcOrder .
  *** id, prioridad, t total, t actual, interrumpible?
  op noOrd : -> AcOrder .
  op null : -> Schedule .
  op _;_ : Schedule Schedule -> Schedule [assoc id: null] .

  op {_ _} : Qid Time -> Signal .
  op null : -> Env .
  op (_ _) : Env Env -> Env [assoc id: null] .

  ***Signals ID
  ***dbellring fridgeA

  ***Orders ID
  ***doorbell fridgeAlert
  ***sayTV sayMed sayDB sayFrid say5pm moveTV moveUser moveSofa moveKitchen
  ***moveCharger moveLRoom moveTable
  ***trayLow trayRai torsoL torsoR torsoS LYellow LWhite LOff wait wait5

  ops kitchen sofa table charger tv user lroom : -> Place .
  ops yellow white off : -> LColor .
  ops raised lowered empty : -> Tray .
  ops right left straight : -> Torso .
  ops TV medicine5pm medReminder doorbellRang fridgeDoor noMsg : -> Message .

  vars T T2 : Time .
  var BOT : Robot .
  vars SCH SCH2 SCH3 : Schedule .
  var ORD : Order .
  var AORD : AcOrder .
  var PLACE : Place .
  var LC : LColor .
```

```
var TRAY : Tray .
var TORSO : Torso .
var MSSG : Message .
vars ENV ENV2 : Env .
var PR PR2 : Nat .
vars ID ID2 : Qid .
vars D INS D2 INS2 : Int .
vars B B2 : Bool .

eq (S:Signal S:Signal) = S:Signal .


op ordenarPrio : Schedule -> Schedule .
ceq [ordenarPrio] : ordenarPrio(SCH ; [ID PR] ; [ID2 PR2] ; SCH2) =
             ordenarPrio(SCH ; [ID2 PR2] ; [ID PR] ; SCH2) if PR2 < PR .

ceq [ordenarPrio] : ordenarPrio(SCH ; [ID PR D INS B] ; [ID2 PR2 D2 INS2 B2]
    ; SCH2) =
             ordenarPrio(SCH ; [ID2 PR2 D2 INS2 B2] ; [ID PR D INS B] ; SCH2
                 ) if PR2 < PR .

ceq [ordenarPrio] : ordenarPrio(SCH ; [ID PR D INS B] ; [ID2 PR2] ; SCH2) =
             ordenarPrio(SCH ; [ID2 PR2] ; [ID PR D INS B] ; SCH2) if PR2 <
                 PR .

ceq [ordenarPrio] : ordenarPrio(SCH ; [ID PR] ; [ID2 PR2 D2 INS2 B2] ; SCH2)
    =
             ordenarPrio(SCH ; [ID2 PR2 D2 INS2 B2] ; [ID PR] ; SCH2) if PR2
                  < PR .

eq [ordenarPrio] : ordenarPrio(SCH:Schedule) = SCH:Schedule [owise] .


eq [updateINI] : BOT | ENV | noOrd | SCH | -1 = BOT | ENV | noOrd |
    updatefunc(SCH, null, null, ENV, 0 ) | 0 .

op updatefunc : Schedule Schedule Schedule Env Time -> Schedule .

  eq [update] : updatefunc( null, SCH2, SCH3, ENV, T ) = ordenarPrio(SCH2) ;
      ordenarPrio(SCH3) .

  ceq [update] : updatefunc( ORD ; SCH, SCH2, SCH3, ENV, T ) = updatefunc(
      SCH, SCH2 ; ORD, SCH3, ENV, T )
         if prec(ORD, ENV, T) .

ceq [update] : updatefunc( ORD ; SCH, SCH2, SCH3, ENV, T ) = updatefunc( SCH,
     SCH2, SCH3 ; ORD, ENV, T )
         if not(prec(ORD, ENV, T)) .


op comp5pm : AcOrder Schedule -> Bool .
 eq comp5pm(AORD, SCH ; ['say5pm 5 3 INS false] ; SCH2) = true .
 eq comp5pm(['say5pm 5 3 INS false], SCH) = true .
 eq comp5pm(AORD, SCH) = false [owise] .

ceq [5pmMedOrd] : BOT | ENV | AORD | SCH | T = BOT | ENV | AORD | updatefunc
    (['say5pm 5 3 3 false] ; SCH, null, null, ENV, T) | T
        if (T == 61200) and not(comp5pm(AORD, SCH)) .

op time24 : Nat -> Nat .
 ceq time24(T) = 0 if T == 86400 .
 eq time24(T) = T + 1 [owise] .
```

```
ceq [said] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | [ID PR D INS B] |
    SCH | T = < PLACE , LC , TRAY , TORSO , noMsg > | ENV | [ID PR D INS B] |
     SCH | T
      if (MSSG =/= noMsg) and (D =/= 3) .


eq [finishTask] : BOT | ENV | [ID PR INS 0 B] | SCH | T = BOT | ENV | noOrd |
     SCH | T .

ceq [iniTask] : BOT | ENV | noOrd | AORD ; SCH | T = BOT | ENV | AORD | SCH |
     T if (T >= 0) and prec(AORD, ENV, T) .

ceq [interruption] : BOT | ENV | [ID PR D INS true] | [ID2 PR2 D2 INS2 B2] ;
    SCH | T =
              BOT | ENV | [ID2 PR2 D2 INS2 B2] | [ID PR D INS true] ; SCH | T
        if (PR2 < PR) and prec([ID2 PR2 D2 INS2 B2], ENV, T) .



op prec : Order Env Time -> Bool .

 eq prec(['doorbell PR], ENV, T) = true .
 eq prec(['fridgeAlert PR], ENV, T) = true .

 eq prec(['sayTV PR 3 INS false], ENV, T) = true .
 eq prec(['sayMed PR 3 INS false], ENV, T) = true .
 ceq prec(['sayDB PR 3 3 false], (ENV {'dbellring T2} ENV2), T) = true if ((T
     - T2) >= 10) .
 ceq prec(['sayDB PR 3 INS false], ENV, T) = true if INS < 3 .
 eq prec(['sayDB PR 3 INS false], ENV, T) = false [owise] .
 ceq prec(['sayFrid PR 3 3 false], (ENV {'fridgeA T2} ENV2), T) = true if ((T
     - T2) >= 30) .
 ceq prec(['sayFrid PR 3 INS false], ENV, T) = true if INS < 3 .
 eq prec(['sayFrid PR 3 INS false], ENV, T) = false [owise] .
 eq prec(['say5pm PR 3 INS false], ENV, T) = true .
 eq prec(['moveTV PR 5 INS true], ENV, T) = true .
 eq prec(['moveUser PR 5 INS true], ENV, T) = true .
 eq prec(['moveSofa PR 5 INS true], ENV, T) = true .
 eq prec(['moveKitchen PR 5 INS true], ENV, T) = true .
 eq prec(['moveCharger PR 5 INS true], ENV, T) = true .
 eq prec(['moveLRoom PR 5 INS true], ENV, T) = true .
 eq prec(['moveTable PR 5 INS true], ENV, T) = true .
 eq prec(['trayLow PR 2 INS false], ENV, T) = true .
 eq prec(['trayRai PR 2 INS false], ENV, T) = true .
 eq prec(['torsoL PR 2 INS false], ENV, T) = true .
 eq prec(['torsoR PR 2 INS false], ENV, T) = true .
 eq prec(['torsoS PR 2 INS false], ENV, T) = true .
 eq prec(['LYellow PR 1 INS true], ENV, T) = true .
 eq prec(['LWhite PR 1 INS true], ENV, T) = true .
 eq prec(['LOff PR 1 INS true], ENV, T) = true .
 eq prec(['wait PR 1 INS true], ENV, T) = true .
 eq prec(['wait5 PR 5 INS true], ENV, T) = true .


rl [doorbellSig] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | noOrd | ['
    doorbell PR] ; SCH | T =>
         < PLACE , LC , TRAY , TORSO , MSSG > | (ENV {'dbellring T}) | noOrd
             | updatefunc(SCH, null, null,(ENV {'dbellring T}), T) | T .

rl [fridgeAlertSig] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | noOrd | ['
    fridgeAlert PR] ; SCH | T =>
         < PLACE , LC , TRAY , TORSO , MSSG > | (ENV {'fridgeA T}) | noOrd |
             updatefunc(SCH, null, null,(ENV {'fridgeA T}), T) | T .
```

```
crl [moveKitchen] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['
    moveKitchen PR 5 INS true] | SCH | T =>
         < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveKitchen PR 5 (INS
              - 1) true] | updatefunc(SCH, null, null, ENV, time24(T)) |
             time24(T)
        if INS > 1 .

rl [moveKitchen2] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['
    moveKitchen PR 5 1 true] | SCH | T =>
         < kitchen , LC , TRAY , TORSO , MSSG > | ENV | ['moveKitchen PR 5 0
             true] | updatefunc(SCH, null, null, ENV, time24(T)) | time24(T)
             .


crl [moveSofa] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveSofa PR 5
     INS true] | SCH | T =>
         < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveSofa PR 5 (INS -
             1) true] | updatefunc(SCH, null, null, ENV, time24(T)) | time24(
             T)
         if INS > 1 .

rl [moveSofa2] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveSofa PR 5
     1 true] | SCH | T =>
         < sofa , LC , TRAY , TORSO , MSSG > | ENV | ['moveSofa PR 5 0 true]
             |  updatefunc(SCH, null, null, ENV, time24(T)) | time24(T) .


crl [moveTable] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveTable PR
     5 INS true] | SCH | T =>
         < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveTable PR 5 (INS -
              1) true] | updatefunc(SCH, null, null, ENV, time24(T)) | time24
             (T)
         if INS > 1 .

rl [moveTable2] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveTable PR
     5 1 true] | SCH | T =>
         < table , LC , TRAY , TORSO , MSSG > | ENV | ['moveTable PR 5 0 true
             ] |  updatefunc(SCH, null, null, ENV, time24(T)) | time24(T) .



crl [moveCharger] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['
    moveCharger PR 5 INS true] | SCH | T =>
         < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveCharger PR 5 (INS
              - 1) true] | updatefunc(SCH, null, null, ENV, time24(T)) |
             time24(T)
         if INS > 1 .

rl [moveCharger2] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['
    moveCharger PR 5 1 true] | SCH | T =>
         < charger , LC , TRAY , TORSO , MSSG > | ENV | ['moveCharger PR 5 0
             true] |  updatefunc(SCH, null, null, ENV, time24(T)) | time24(T)
              .


crl [moveTV] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveTV PR 5 INS
     true] | SCH | T =>
         < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveTV PR 5 (INS - 1)
              true] | updatefunc(SCH, null, null, ENV, time24(T)) | time24(T)
         if INS > 1 .

rl [moveTV2] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveTV PR 5 1
    true] | SCH | T =>
```

```
          < tv , LC , TRAY , TORSO , MSSG > | ENV | ['moveTV PR 5 0 true] |
              updatefunc(SCH, null, null, ENV, time24(T)) | time24(T) .



crl [moveUser] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveUser PR 5
     INS true] | SCH | T =>
          < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveUser PR 5 (INS -
              1) true] | updatefunc(SCH, null, null, ENV, time24(T)) | time24(
              T)
          if INS > 1 .

rl [moveUser2] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveUser PR 5
     1 true] | SCH | T =>
          < user , LC , TRAY , TORSO , MSSG > | ENV | ['moveUser PR 5 0 true]
              |  updatefunc(SCH, null, null, ENV, time24(T)) | time24(T) .



crl [moveLRoom] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveLRoom PR
     5 INS true] | SCH | T =>
          < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveLRoom PR 5 (INS -
               1) true] |  updatefunc(SCH, null, null, ENV, time24(T)) |
              time24(T)
          if INS > 1 .

rl [moveLRoom2] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['moveLRoom PR
     5 1 true] | SCH | T =>
          < lroom , LC , TRAY , TORSO , MSSG > | ENV | ['moveLRoom PR 5 0 true
              ] |  updatefunc(SCH, null, null, ENV, time24(T)) | time24(T) .



rl [sayTV] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['sayTV PR 3 INS
    false] | SCH | T =>
          < PLACE , LC , TRAY , TORSO , TV > | ENV | ['sayTV PR 3 (INS - 1)
              false] | updatefunc(SCH, null, null, ENV, time24(T)) | time24(T)
               .

rl [sayMed5pm] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['say5pm PR 3
    INS false] | SCH | T =>
          < PLACE , LC , TRAY , TORSO , medicine5pm > | ENV | ['say5pm PR 3 (
              INS - 1) false] | updatefunc(SCH, null, null, ENV, time24(T)) |
              time24(T) .

rl [sayMed] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['sayMed PR 3 INS
    false] | SCH | T =>
          < PLACE , LC , TRAY , TORSO , medReminder > | ENV | ['sayMed PR 3 (
              INS - 1) false] | updatefunc(SCH, null, null, ENV, time24(T)) |
              time24(T) .

rl [sayDoorbell] : < PLACE , LC , TRAY , TORSO , MSSG > | (ENV {'dbellring T2
    } ENV2) | ['sayDB PR 3 3 false] | SCH | T =>
          < PLACE , LC , TRAY , TORSO , doorbellRang > | (ENV ENV2) | ['sayDB
              PR 3 2 false] | updatefunc(SCH, null, null, (ENV ENV2), time24(T
              )) | time24(T) .

crl [sayDoorbell2] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['sayDB PR
    3 INS false] | SCH | T =>
          < PLACE , LC , TRAY , TORSO , doorbellRang > | ENV | ['sayDB PR 3 (
              INS - 1) false] | updatefunc(SCH, null, null, ENV, time24(T)) |
              time24(T)
          if INS < 3 .
```

```
rl [sayFridgeDoor] : < PLACE , LC , TRAY , TORSO , MSSG > | (ENV {'fridgeA T2
    } ENV2) | ['sayFrid PR 3 3 false] | SCH | T =>
         < PLACE , LC , TRAY , TORSO , fridgeDoor > | (ENV ENV2) | ['sayFrid
             PR 3 2 false] | updatefunc(SCH, null, null, (ENV ENV2), time24(T
             )) | time24(T) .

crl [sayFridgeDoor2] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['sayFrid
     PR 3 INS false] | SCH | T =>
         < PLACE , LC , TRAY , TORSO , fridgeDoor > | ENV | ['sayFrid PR 3 (
             INS - 1) false] | updatefunc(SCH, null, null, ENV, time24(T)) |
             time24(T)
         if INS < 3 .



rl [trayLowered] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['trayLow PR
    2 INS false] | SCH | T =>
         < PLACE , LC , lowered , TORSO , MSSG > | ENV | ['trayLow PR 2 (INS
             - 1) false] | updatefunc(SCH, null, null, ENV, time24(T)) |
             time24(T) .

rl [trayRaised] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['trayRai PR 2
     INS false] | SCH | T =>
         < PLACE , LC , raised , TORSO , MSSG > | ENV | ['trayRai PR 2 (INS -
              1) false] | updatefunc(SCH, null, null, ENV, time24(T)) |
             time24(T) .

rl [torsoLeft] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['torsoL PR 2
    INS false] | SCH | T =>
         < PLACE , LC , TRAY , left , MSSG > | ENV | ['torsoL PR 2 (INS - 1)
             false] | updatefunc(SCH, null, null, ENV, time24(T)) | time24(T)
              .

rl [torsoRight] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['torsoR PR 2
    INS false] | SCH | T =>
         < PLACE , LC , TRAY , right , MSSG > | ENV | ['torsoR PR 2 (INS - 1)
              false] | updatefunc(SCH, null, null, ENV, time24(T)) | time24(T
             ) .

rl [torsoStright] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['torsoS PR
    2 INS false] | SCH | T =>
         < PLACE , LC , TRAY , straight , MSSG > | ENV | ['torsoS PR 2 (INS -
              1) false] | updatefunc(SCH, null, null, ENV, time24(T)) |
             time24(T) .



rl [lightYellow] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['LYellow PR
    1 INS true] | SCH | T =>
         < PLACE , yellow , TRAY , TORSO , MSSG > | ENV | ['LYellow PR 1 (INS
              - 1) true] | updatefunc(SCH, null, null, ENV, time24(T)) |
             time24(T) .

rl [lightWhite] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['LWhite PR 1
    INS true] | SCH | T =>
         < PLACE , white , TRAY , TORSO , MSSG > | ENV | ['LWhite PR 1 (INS -
              1) true] | updatefunc(SCH, null, null, ENV, time24(T)) | time24
             (T) .

rl [lightOff] : < PLACE , LC , TRAY , TORSO , MSSG > | ENV | ['LOff PR 1 INS
    true] | SCH | T =>
```

```
          < PLACE , off , TRAY , TORSO , MSSG > | ENV | ['LOff PR 1 (INS - 1)
              true] | updatefunc(SCH, null, null, ENV, time24(T)) | time24(T)
              .

  rl [wait] : BOT | ENV | ['wait PR 1 INS true] | SCH | T => BOT | ENV | ['wait
       PR 1 (INS - 1) true] | updatefunc(SCH, null, null, ENV, time24(T)) |
      time24(T) .

  rl [wait5] : BOT | ENV | ['wait5 PR 5 INS true] | SCH | T => BOT | ENV | ['
      wait5 PR 5 (INS - 1) true] | updatefunc(SCH, null, null, ENV, time24(T))
      | time24(T) .

  crl [idle] : BOT | ENV | noOrd | ORD ; SCH | T => BOT | ENV | noOrd |
      updatefunc(ORD ; SCH, null, null, ENV, time24(T)) | time24(T)
        if (T < 86399) and (T >= 0) and not(prec(ORD, ENV, T)) .

  crl [idle2] : BOT | ENV | noOrd | null | T => BOT | ENV | noOrd | null |
      time24(T) if (T < 86400) and (T >= 0) .

endm

mod MODEL-PREDS is
 inc INT + QID .
 protecting CARE-O-BOT .
 including SATISFACTION .
 including LTL-SIMPLIFIER .
 including MODEL-CHECKER .
 subsort MyState < State .
 ops kitchen interrupted interruptible say message preconditions doorbell saydb
      running pendingOrders 5pmOrder time17 5pm : -> Prop .

  vars T T2 : Time .
  var BOT : Robot .
  vars ID ID2 : Qid .
  vars SCH SCH2 SCH3 : Schedule .
  var ORD : Order .
  var AORD : AcOrder .
  var PLACE : Place .
  var LC : LColor .
  var TRAY : Tray .
  var TORSO : Torso .
  var MSSG : Message .
  vars ENV ENV2 : Env .
  var PR PR2 : Nat .
  vars INS D INS2 D2 : Int .
  vars B B2 : Bool .

  ops init1 init2 : -> MyState .
  eq init1 = < sofa , off , lowered , straight , noMsg > | null | noOrd | ['
      moveUser 2 5 5 true] ; ['moveKitchen 20 5 5 true] ; ['moveTable 80 5 5
      true] ; ['moveTV 50 5 5 true] ; ['doorbell 1] ; ['sayDB 1 3 3 false] ; ['
      sayFrid 1 3 3 false] ; ['fridgeAlert 29] | -1 .
  eq init2 = < table , yellow , empty , straight , noMsg > | null | noOrd | ['
      moveUser 2 5 5 true] ; ['moveTable 80 5 5 true] ; ['sayDB 1 3 3 false] ;
      ['fridgeAlert 29] ; ['LWhite 25 1 1 true] ; ['trayRai 33 2 2 false] ; ['
      sayTV 17 3 3 false] ; ['torsoL 40 2 2 false] | -1 .

  eq < kitchen , LC , TRAY , TORSO , MSSG > | ENV | AORD | SCH | T |= kitchen =
       true .

  ceq BOT | ENV | AORD | [ID PR D INS B] ; SCH | T |= interrupted = true if D >
       INS .
  eq BOT | ENV | AORD | [ID PR D INS true] ; SCH | T |= interruptible = true .
```

```
  eq BOT | ENV | AORD | [ID PR D INS false] ; SCH | T |= interruptible = false
      .

  ceq BOT | ENV | [ID PR 3 INS false] | SCH | T |= say = true if INS < 3 .
  ceq < PLACE , LC , TRAY , TORSO , MSSG > | ENV | AORD | SCH | T |= message =
      true if (MSSG =/= noMsg) .

  ceq BOT | ENV | AORD | SCH | T |= running = true if (AORD =/= noOrd) .
  ceq BOT | ENV | AORD | SCH | T |= preconditions = true if prec(AORD, ENV, T)
      .

  ceq BOT | ENV | AORD | SCH | T |= 5pmOrder = true if comp5pm(AORD, SCH) .
  ceq BOT | ENV | AORD | SCH | T |= time17 = true if (T >= 61200) .

  eq BOT | (ENV {'dbellring T2} ENV2) | AORD | SCH | T |= doorbell = true .
  eq BOT | ENV | ['sayDB PR 3 INS false] | SCH | T |= saydb = true .

  ceq BOT | ENV | AORD | SCH | T |= pendingOrders = true if (AORD =/= noOrd) .
  ceq BOT | ENV | AORD | ORD ; SCH | T |= pendingOrders = true if prec(ORD, ENV
      , T) .
  ceq BOT | ENV | AORD | ORD ; SCH | T |= pendingOrders = false if not(prec(ORD
      , ENV, T)) .

  ceq BOT | ENV | AORD | SCH | T |= 5pm = true if (T == 61200) .

endm
```

APÉNDICE B
Código fuente - fichero
model-checker.maude

```
***(

    This file is part of the Maude 2 interpreter.

    Copyright 1997-2006 SRI International, Menlo Park, CA 94025, USA.

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

)

***
*** Maude LTL satisfiability solver and model checker.
*** Version 2.3.
***

fmod LTL is
  protecting BOOL .
  sort Formula .

  *** primitive LTL operators
  ops True False : -> Formula [ctor format (g o)] .
  op ~_ : Formula -> Formula [ctor prec 53 format (r o d)] .
  op _/\_ : Formula Formula -> Formula [comm ctor gather (E e) prec 55 format (
      d r o d)] .
  op _\/_ : Formula Formula -> Formula [comm ctor gather (E e) prec 59 format (
      d r o d)] .
  op O_ : Formula -> Formula [ctor prec 53 format (r o d)] .
  op _U_ : Formula Formula -> Formula [ctor prec 63 format (d r o d)] .
  op _R_ : Formula Formula -> Formula [ctor prec 63 format (d r o d)] .

  *** defined LTL operators
  op _->_ : Formula Formula -> Formula [gather (e E) prec 65 format (d r o d)]
      .
```

```
  op _<->_ : Formula Formula -> Formula [prec 65 format (d r o d)] .
  op <>_ : Formula -> Formula [prec 53 format (r o d)] .
  op []_ : Formula -> Formula [prec 53 format (r d o d)] .
  op _W_ : Formula Formula -> Formula [prec 63 format (d r o d)] .
  op _|->_ : Formula Formula -> Formula [prec 63 format (d r o d)] . *** leads-
      to
  op _=>_ : Formula Formula -> Formula [gather (e E) prec 65 format (d r o d)]
      .
  op _<=>_ : Formula Formula -> Formula [prec 65 format (d r o d)] .

  vars f g : Formula .

  eq f -> g = ~ f \/ g .
  eq f <-> g = (f -> g) /\ (g -> f) .
  eq <> f = True U f .
  eq [] f = False R f .
  eq f W g = (f U g) \/ [] f .
  eq f |-> g = [](f -> (<> g)) .
  eq f => g = [] (f -> g) .
  eq f <=> g = [] (f <-> g) .

  *** negative normal form
  eq ~ True = False .
  eq ~ False = True .
  eq ~ ~ f = f .
  eq ~ (f \/ g) = ~ f /\ ~ g .
  eq ~ (f /\ g) = ~ f \/ ~ g .
  eq ~ O f = O ~ f .
  eq ~(f U g) = (~ f) R (~ g) .
  eq ~(f R g) = (~ f) U (~ g) .
endfm

fmod LTL-SIMPLIFIER is
  including LTL .

  *** The simplifier is based on:
  ***   Kousha Etessami and Gerard J. Holzman,
  ***   "Optimizing Buchi Automata", p153-167, CONCUR 2000, LNCS 1877.
  *** We use the Maude sort system to do much of the work.

  sorts TrueFormula FalseFormula PureFormula PE-Formula PU-Formula .
  subsort TrueFormula FalseFormula < PureFormula <
    PE-Formula PU-Formula < Formula .

  op True : -> TrueFormula [ctor ditto] .
  op False : -> FalseFormula [ctor ditto] .
  op _/\_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
  op _/\_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
  op _/\_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
  op _\/_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
  op _\/_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
  op _\/_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
  op O_ : PE-Formula -> PE-Formula [ctor ditto] .
  op O_ : PU-Formula -> PU-Formula [ctor ditto] .
  op O_ : PureFormula -> PureFormula [ctor ditto] .
  op _U_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
  op _U_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
  op _U_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
  op _U_ : TrueFormula Formula -> PE-Formula [ctor ditto] .
  op _U_ : TrueFormula PU-Formula -> PureFormula [ctor ditto] .
  op _R_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
  op _R_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
  op _R_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
  op _R_ : FalseFormula Formula -> PU-Formula [ctor ditto] .
```

```
op _R_ : FalseFormula PE-Formula -> PureFormula [ctor ditto] .

vars p q r s : Formula .
var pe : PE-Formula .
var pu : PU-Formula .
var pr : PureFormula .

*** Rules 1, 2 and 3; each with its dual.
eq (p U r) /\ (q U r) = (p /\ q) U r .
eq (p R r) \/ (q R r) = (p \/ q) R r .
eq (p U q) \/ (p U r) = p U (q \/ r) .
eq (p R q) /\ (p R r) = p R (q /\ r) .
eq True U (p U q) = True U q .
eq False R (p R q) = False R q .

*** Rules 4 and 5 do most of the work.
eq p U pe = pe .
eq p R pu = pu .

*** An extra rule in the same style.
eq O pr = pr .

*** We also use the rules from:
***   Fabio Somenzi and Roderick Bloem,
*** "Efficient Buchi Automata from LTL Formulae",
***   p247-263, CAV 2000, LNCS 1633.
*** that are not subsumed by the previous system.

*** Four pairs of duals.
eq O p /\ O q = O (p /\ q) .
eq O p \/ O q = O (p \/ q) .
eq O p U O q = O (p U q) .
eq O p R O q = O (p R q) .
eq True U O p = O (True U p) .
eq False R O p = O (False R p) .
eq (False R (True U p)) \/ (False R (True U q)) = False R (True U (p \/ q)) .
eq (True U (False R p)) /\ (True U (False R q)) = True U (False R (p /\ q)) .

*** <= relation on formula
op _<=_ : Formula Formula -> Bool [prec 75] .

eq p <= p = true .
eq False <= p  = true .
eq p <= True = true .

ceq p <= (q /\ r) = true if (p <= q) /\ (p <= r) .
ceq p <= (q \/ r) = true if p <= q .
ceq (p /\ q) <= r = true if p <= r .
ceq (p \/ q) <= r = true if (p <= r) /\ (q <= r) .

ceq p <= (q U r) = true if p <= r .
ceq (p R q) <= r = true if q <= r .
ceq (p U q) <= r = true if (p <= r) /\ (q <= r) .
ceq p <= (q R r) = true if (p <= q) /\ (p <= r) .
ceq (p U q) <= (r U s) = true if (p <= r) /\ (q <= s) .
ceq (p R q) <= (r R s) = true if (p <= r) /\ (q <= s) .

*** condition rules depending on <= relation
ceq p /\ q = p if p <= q .
ceq p \/ q = q if p <= q .
ceq p /\ q = False if p <= ~ q .
ceq p \/ q = True if ~ p <= q .
ceq p U q = q if p <= q .
ceq p R q = q if q <= p .
```

```
  ceq p U q = True U q if p =/= True /\ ~ q <= p .
  ceq p R q = False R q if p =/= False /\ q <= ~ p .
  ceq p U (q U r) = q U r if p <= q .
  ceq p R (q R r) = q R r if q <= p .
endfm

fmod SAT-SOLVER is
  protecting LTL .

  *** formula lists and results
  sorts FormulaList SatSolveResult TautCheckResult .
  subsort Formula < FormulaList .
  subsort Bool < SatSolveResult TautCheckResult .
  op nil : -> FormulaList [ctor] .
  op _;_ : FormulaList FormulaList -> FormulaList [ctor assoc id: nil] .
  op model : FormulaList FormulaList -> SatSolveResult [ctor] .

  op satSolve : Formula ~> SatSolveResult
  [special (
     id-hook SatSolverSymbol
     op-hook trueSymbol           (True : ~> Formula)
     op-hook falseSymbol    (False : ~> Formula)
     op-hook notSymbol    (~_ : Formula ~> Formula)
     op-hook nextSymbol   (O_ : Formula ~> Formula)
     op-hook andSymbol    (_/\_ : Formula Formula ~> Formula)
     op-hook orSymbol   (_\/_ : Formula Formula ~> Formula)
     op-hook untilSymbol    (_U_ : Formula Formula ~> Formula)
     op-hook releaseSymbol  (_R_ : Formula Formula ~> Formula)
     op-hook formulaListSymbol
       (_;_ : FormulaList FormulaList ~> FormulaList)
     op-hook nilFormulaListSymbol (nil : ~> FormulaList)
     op-hook modelSymbol
       (model : FormulaList FormulaList ~> SatSolveResult)
     term-hook falseTerm    (false)
   )] .

  op counterexample : FormulaList FormulaList -> TautCheckResult [ctor] .
  op tautCheck : Formula ~> TautCheckResult .
  op $invert : SatSolveResult -> TautCheckResult .

  var F : Formula .
  vars L C : FormulaList .
  eq tautCheck(F) = $invert(satSolve(~ F)) .
  eq $invert(false) = true .
  eq $invert(model(L, C)) = counterexample(L, C) .
endfm

fmod SATISFACTION is
  protecting BOOL .
  sorts State Prop .
  op _|=_ : State Prop -> Bool [frozen] .
endfm

fmod MODEL-CHECKER is
  protecting QID .
  including SATISFACTION .
  including LTL .
  subsort Prop < Formula .

  *** transitions and results
  sorts RuleName Transition TransitionList ModelCheckResult .
  subsort Qid < RuleName .
  subsort Transition < TransitionList .
  subsort Bool < ModelCheckResult .
```

```
  ops unlabeled deadlock : -> RuleName .
  op {_,_} : State RuleName -> Transition [ctor] .
  op nil : -> TransitionList [ctor] .
  op __ : TransitionList TransitionList -> TransitionList [ctor assoc id: nil]
      .
  op counterexample : TransitionList TransitionList -> ModelCheckResult [ctor]
      .

  op modelCheck : State Formula ~> ModelCheckResult
  [special (
     id-hook ModelCheckerSymbol
     op-hook trueSymbol            (True : ~> Formula)
     op-hook falseSymbol     (False : ~> Formula)
     op-hook notSymbol     (~_ : Formula ~> Formula)
     op-hook nextSymbol    (O_ : Formula ~> Formula)
     op-hook andSymbol     (_/\_ : Formula Formula ~> Formula)
     op-hook orSymbol    (_\/_ : Formula Formula ~> Formula)
     op-hook untilSymbol     (_U_ : Formula Formula ~> Formula)
     op-hook releaseSymbol  (_R_ : Formula Formula ~> Formula)
           op-hook satisfiesSymbol       (_|=_ : State Formula ~> Bool)
     op-hook qidSymbol     (<Qids> : ~> Qid)
     op-hook unlabeledSymbol  (unlabeled : ~> RuleName)
     op-hook deadlockSymbol (deadlock : ~> RuleName)
     op-hook transitionSymbol ({_,_} : State RuleName ~> Transition)
     op-hook transitionListSymbol
       (__ : TransitionList TransitionList ~> TransitionList)
     op-hook nilTransitionListSymbol  (nil : ~> TransitionList)
     op-hook counterexampleSymbol
       (counterexample : TransitionList TransitionList ~> ModelCheckResult)
     term-hook trueTerm   (true)
   )] .
endfm
```