Formalización de Lógica Temporal Alternante en el Cálculo de Construcciones Coinductivas

Tesina de grado presentada
por

Dante Zanarini
Z-0467/7

al
Departamento de Ciencias de la Computación
en cumplimiento parcial de los requerimientos
para la obtención del grado de

Licenciado en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Av. Pellegrini 250, Rosario, República Argentina

Noviembre 2008

Supervisores

Carlos D. Luna Luis Sierra

Instituto de Computación, Facultad de Ingeniería

Universidad de la República

Julio Herrera y Reissig 565, Montevideo, Uruguay

Resumen

La lógica ATL (*Alternating time temporal logic*) [2, 4] es una generalización de CTL (*Computation Tree Logic*) [9] para la especificación de propiedades temporales en sistemas multiagentes abiertos. Mientras que CTL permite cuantificación existencial y universal sobre las posibles trazas de ejecución, ATL incluye un mecanismo de cuantificación *selectiva* sobre aquellas trazas del sistema que son posibles resultados de un juego, en el cual un conjunto de componentes *controlables* y el entorno alternan movimientos o acciones.

En este trabajo presentamos una formalización en el cálculo de construcciones coinductivas (CCI) [10, 15, 28] de ATL, junto con su implementación en el asistente de pruebas *Coq* [7, 13].

La formalización puede dividirse en dos partes. En primer lugar, representamos en CCI el modelo semántico de ATL, las *estructuras de juegos concurrentes* (CGS), que generalizan las estructuras de Kripke a sistemas multiagentes. La segunda parte define la semántica de los operadores de ATL, basándose en los conceptos de estrategias y coaliciones formalizados en la primera, junto con las propiedades de punto fijo de los operadores temporales.

El uso de la formalización como un sistema deductivo para ATL se ilustran mediante la demostración de una serie de resultados generales sobre dicha lógica.

Finalmente, la representación de CGS y ATL es utilizada para la especificación y verificación de dos casos de estudio. El primero de ellos, tomado de [2], describe un protocolo de control de un paso a nivel, en el cual un controlador y un tren interactúan mediante pedidos de ingreso y otorgamiento de permisos para ingresar a una compuerta. El segundo sistema es una generalización del anterior, en el cual existe un número arbitrariamente grande de trenes que compiten por ingresar a la compuerta. La especificación y verificación de ambos sistemas nos permitió evaluar y validar la formalización propuesta, así como también comparar y relacionar diferentes aspectos de la verificación de sistemas reactivos, en particular respecto de aquellos que no poseen un espacio de estados finito, o están definidos en función de parámetros de especificación que dificultan el uso de herramientas de verificación automáticas.

Agradecimientos

Muchas personas han contribuido, de una u otra forma, a la elaboración de este trabajo. Es para mi un gusto poder expresarles aquí mi eterna gratitud.

A Ceci, que todo el tiempo estuvo a mi lado; sin su constante apoyo este trabajo no hubiese sido posible.

A mi familia, por demostrarme que dar todo por los demás vale la pena.

A Carlos y Luis, mis supervisores, por sus aportes, críticas, sugerencias, comentarios y, sobre todo, por su infinita paciencia a lo largo de este proceso.

A los miembros de la OTCB, que están desde el comienzo.

A los docentes de la licenciatura de Ciencias de la Computación, por hacerme sentir en familia. Especialmente a Raúl, Ana, Guido, Gabriela, Graciela, Maxi, Fede S., Fede O., Pablo (todos ellos), Juanito y Gerardo, por su permanente disposición y desinteresada ayuda.

A los jóvenes estudiantes de LCC, con los que en este último tiempo aprendí tanto como durante mi paso por las aulas de la facultad.

A Pao, Fredy y Mati, que nos hicieron sentir como en casa a miles de kilómetros de distancia.

A los barrieros, los chicos del superior, a los compañeros de fútbol, y a todos los que me han acompañado durante estos años, gracias... totales.

Índice general

1

Introducción

La verificación de programas es hoy una de las áreas de mayor crecimiento en ciencias de la computación. Si nos enfocamos en la verificación de sistemas reactivos y concurrentes, tales como sistemas operativos, sistemas embebidos, hardware para dispositivos electrónicos, etc., la lógica temporal ha resultado un modelo formal ampliamente utilizado para la verificación de tales sistemas. Lógicas temporales tales como LTL [29] y CTL [9] combinan suficiente poder expresivo con propiedades computacionales tales como decidibilidad, permitiendo construir algoritmos que, dado un modelo del sistema y una fórmula temporal, deciden la validez de dicha fórmula en el modelo.

La posibilidad de verificar automáticamente que un sistema satisface su especificación (expresada como una fórmula temporal) resulta de especial interés, y ha sido tema de constante investigación en las últimas décadas. Herramientas de verificación automática tales como SMV [27] y Spin [19] son ampliamente utilizadas a nivel industrial, y se encuentran en constante evolución. Sin embargo, la verificación automática es posible sólo si el modelo a verificar es finito, y el número de estados no lo convierte en computacionalmente intratable. Cuando estas propiedades no se cumplen, es necesario atacar el problema con herramientas más expresivas, tales como un sistema deductivo.

En este trabajo presentamos una formalización en el cálculo de construcciones coinductivas [10, 15, 28] de la *lógica temporal alternante* [2, 4] (ATL, por sus siglas en inglés), junto con su implementación en el asistente de pruebas *Coq* [7, 13]. ATL es una lógica temporal para la especificación de sistemas abiertos basada en teoría de juegos. ATL incluye un mecanismo de cuantificación *selectiva* sobre aquellas trazas del sistema que son posibles resultados de un juego, en el cual un conjunto de componentes *controlables* y el entorno alternan movimientos o acciones.

La formalización implementada en *Coq* será utilizada a lo largo del trabajo como sistema deductivo de carácter general para ATL, así como para la verificación de sistemas reactivos concretos. En particular, presentamos un caso de estudio donde el número de estados y componentes del sistema son no acotados, y por lo tanto no es posible verificar directamente tal sistema mediante algoritmos de verificación automática.

El resto del trabajo se organiza como sigue. En el capítulo 1 describimos brevemente algunos aspectos de la especificación de sistemas mediante lógica temporal, el cálculo de construcciones coinductivas y el asistente de pruebas *Coq*. El capítulo 2 describe en detalle la lógica ATL, presentando su modelo semántico (sección 2.1), sintaxis (2.2) y semántica (2.3). En la sección 2.4 presentamos una axiomatización completa y consistente de ATL, tomada de [17], que será de especial interés a la hora de formalizar los operadores temporales; la sección 2.5 discute algunos aspectos relacionados con la verificación automática de ATL. La formalización de las estructuras de juegos concurrentes y de la lógica ATL se describe en el capítulo 3. En el capítulo 4 se demuestran propiedades relativas a la formalización, tanto para estructuras de juegos concurrentes (4.1) como para fórmulas de ATL (4.2). El capítulo 5 presenta los casos de estudio descriptos anteriormente, tanto para el modelo finito (5.1) como para la generalización propuesta en este trabajo (5.2). Por último, en el capítulo 6 presentamos algunas conclusiones y trabajos futuros.

1.1. Especificación de Sistemas mediante Lógica Temporal

En 1977. Amir Pnueli propone el uso de la lógica temporal para la especificación y verificación de programas, particularmente aquellos que operan de manera continua y en forma concurrente, tales como sistemas operativos y protocolos de comunicación en redes.

En un modelo secuencial típico, e.g. un programa que ordena una lista de números, la correctitud puede formalizarse mediante un par precondición/ postcondición en un formalismo tal como la lógica de Hoare. Esto sucede porque que el programa puede verse como una *transformación* del estado inicial al estado final. Sin embargo, para un programa *reactivo*, que mantiene una continua interacción con el entorno, las nociones de estado final y postcondición no resultan adecuadas para razonar sobre el sistema, dado que el comportamiento esperado es una continua y (potencialmente) infinita interacción con el entorno (es decir, no hay *estado final*). Operadores temporales tales como *eventualmente* e *invaria-*

blemente resultan más apropiados para describir el comportamiento de tales sistemas.

Desde la publicación del trabajo de Pnueli [29], diferentes formalismos basados en lógica temporal se han propuesto para la especificación y verificación de sistemas reactivos y concurrentes. Cada uno de estos intenta encontrar un balance adecuado entre expresividad y complejidad computacional (decidibilidad, verificación automática, etc.). Presentamos a continuación una breve clasificación de de acuerdo a diferentes características de los formalismos.

1.1.1. Clasificación de Lógicas Temporales

En [14], E. Emerson clasifica las diferentes lógicas temporales atendiendo a la naturaleza del tiempo en consideración, el modelo temporal y la expresividad de la lógica. Citamos algunas de estas clasificaciones, que resultan de particular interés en este trabajo.

Proposicional vs. Primer Orden

En una lógica temporal proposicional, la parte de las fórmulas que no corresponde al comportamiento temporal es simplemente una lógica proposicional clásica. Las fórmulas se construyen a partir de proposiciones atómicas (intuitivamente, dichas proposiciones expresan hechos respecto de los estados del sistema subyacente, tales como "la variable x es positiva"), conectivos lógicos tales como $\wedge$, $\vee$, $\neg$, y los operadores temporales.

Si refinamos las proposiciones atómicas de una lógica temporal proposicional en expresiones construidas a partir de variables, constantes, funciones, predicados y cuantificadores, obtenemos una lógica temporal de primer orden. Si asumimos que cada proposición atómica es decidible, entonces la lógica temporal de primer orden es decidible, es decir, la indecidibilidad del sistema se encuentra en las fórmulas atómicas y no en la parte temporal de la lógica.

Por otro lado, aquellas lógicas de primer orden que permiten introducir operadores temporales dentro del alcance de cuantificadores, por ejemplo, mediante fórmulas tales como $\mathit{eventualmente}(\forall x, P\ x) \rightarrow \forall x, \mathit{eventualmente}(P\ x)$ resultan indecidibles tanto a nivel de proposiciones atómicas como de fórmulas temporales.

Lineales vs. Arborescentes

Si consideramos que en cualquier estado de nuestro sistema existe sólo un posible estado futuro, diremos que la naturaleza del tiempo en el modelo es *lineal*. Si, en cambio, suponemos que en cada momento el tiempo puede bifurcarse para representar diferentes estados futuros posibles, entonces estamos en presencia de un modelo de tiempo arborescente.

Dependiendo entonces de la naturaleza del transcurso del tiempo que se considere, nuestro sistema será una lógica temporal lineal, o una lógica temporal arborescente. Los operadores temporales varían de uno a otro modelo, proveyendo en el primer caso modalidades para razonar sobre una línea temporal; mientras que en el segundo caso se permite cuantificación sobre los posibles estados futuros del sistema.

Puntos vs. Intervalos

En general, los formalismos de lógica temporal desarrollados para verificación de programas se basan en operadores temporales que se evalúan como verdaderos o falsos en determinados *puntos* del tiempo. Sin embargo, algunos formalismos tales como *duration calculus*, poseen operadores temporales que son evaluados sobre intervalos temporales.

Relacionado a este aspecto, consideramos ahora la siguiente clasificación

Discreto vs. Continuo

En la mayor parte de las lógicas temporales, el tiempo es *discreto*; el momento presente corresponde al estado actual del sistema, y el próximo instante de tiempo corresponde al inmediato sucesor del sistema. De esta forma, la estructura temporal subyacente se compone de los enteros no negativos, que se corresponden con una secuencia de estados. Sin embargo, algunas lógicas cuyo modelo subyacente son los números reales o los racionales, han sido investigadas y han encotrado aplicación a la verificación de programas en el ámbito de sistemas de tiempo real, donde un modelo cuantitativo del tiempo de respuesta es requerido.

ATL es una lógica proposicional, con un modelo arborescente y discreto.

1.1.2. Verificación Automática y Sistemas Deductivos

La verificación formal de sistemas comprende, en general, las siguientes etapas [20]:

- Un lenguaje de *descripción* de sistemas, donde poder definir un modelo del sistema en estudio,
- un lenguaje de *especificación* que permita describir las propiedades a ser verificadas, y
- un *método de verificación*, para establecer cuándo la descripción del sistema satisface su especificación.

En la verificación de sistemas reactivos utlizando lógica temporal, antendiendo al método de verificación utilizado, dos importante enfoques se destacan: la *verificación de modelos* (o *model checking*), y los sistemas deductivos (*proof systems*).

Verificación de Modelos

En este enfoque, el método de verificación es automático. Dado un modelo *finito* $\mathcal{M}$ del sistema y una propiedad φ expresada mediante una lógica temporal L, un *model checker* para L es un programa que decide si la fórmula φ es válida en $\mathcal{M}$.

Desde su aparición hace más de veinte años , un gran número de herramientas de verificación automática han sido desarrolladas y utilizadas en la verificación de sistemas reactivos y concurrentes tales como protocolos de comunicación, sistemas embebidos y circuitos integrados. Algunas de tales herramientas son *SPIN* [19] (LTL), *SMV* [27] (CTL) y *Mocha* [5] (ATL).

La verificación automática es sólo posible cuando el modelo a verificar es finito, y el número de estados no lo convierte en computacionalmente intratable. Cuando estas propiedades no se cumplen, es necesario atacar el problema con herramientas más expresivas, tales como un sistema deductivo.

Sistemas Deductivos

En los métodos de verificación deductivos, se demuestra la validez de una fórmula φ en un modelo $\mathcal{M}$ mediante la *construcción* de una prueba, usando en general un sistema axiomático-deductivo. Si bien la derivación de la prueba puede realizarse en forma manual, la complejidad de los sistemas a verificar y las fórmulas a demostrar hacen necesario el uso de *asistentes de pruebas*, tales como *Isabelle/HOL* [1], *Coq* [13] o *Lego* [25]. Tales herramientas ofrecen un lenguaje de especificación de alto orden en el cual es posible describir el modelo

y las propiedades a demostrar. La demostración de propiedades se realiza en general de forma interactiva mediante el uso de tácticas, que son mecanismos para construir la prueba paso a paso.

Una obvia desventaja de este método es la necesidad de interacción del usuario en el proceso de demostración, proceso que en general requiere un profundo conocimiento tanto del modelo a verificar como de la herramienta en uso. Por otro lado, la ventaja más evidente de este enfoque es la posibilidad de verificar sistemas con un número arbitrariamente grande o infinito de estados.

En [24], se realiza un interesante análisis comparativo de ambos enfoques, junto a una metodología que propone integrarlos. En este trabajo seguimos algunas de las ideas ahí presentadas, y en el capítulo 5 proponemos un nuevo mecanismo de integración entre ambos modelos, donde la verificación automática aplicada a casos particulares de sistemas paramétricos puede ayudar a *guiar* las demostraciones en el caso general.

1.2. El Cálculo de Construcciones Coinductivas

El cálculo de construcciones coinductivas es un sistema formal basado en la teoría de tipos intuicionista. Esta teoría, introducida por Per Martin Löf en 1972 [26], fue desarrollada originalmente como una formalización constructiva de la matemática. A diferencia de otros formalismos, no se basa en lógica de primer orden, sino que dicha lógica es interpretada mediante la correspondencia entre proposiciones y tipos que brinda el isomorfismo de Curry-Howard, [1] conocido también como *propositions as types* (proposiciones como tipos) y *proofs as programs* (pruebas como programas).

La teoría de tipos intuicionista introduce la noción de tipos dependientes (tipos que contienen valores), lo cual permite extender el isomorfismo de Curry-Howard a la lógica de primer orden. Un predicado se convierte en un tipo (dependiente), y probar su validez se reduce a encontrar un habitante del tipo correspondiente.

El cálculo de construcciones (CoC), desarrollado por Coquand y Huet [10], presenta una síntesis de los conceptos de tipos dependientes y polimorfismo que hace posible adaptar la teoría de tipos intuicionista a un asistente de pruebas. A los efectos de diferenciar objetos computacionales de información lógica, CoC distingue entre dos clases de tipos (cada una

[1] Este isomorfismo, en su versión más simple, asocia a cada término del cálculo lambda simplemente tipado una demostración en lógica minimal intuicionista

de estas clases de tipos se denomina *sort*).

- *Prop*: Los habitantes de este *sort* representan proposiciones. Un habitante $P : Prop$ es una proposición, y un término $x : P$ una *prueba* de P.
- *Set*: Este *sort* agrupa los tipos que contienen información computacional. Ejemplos de tipos que habitan en *Set* son los tipos de datos presentes en cualquier lenguaje de programación funcional.

Al incluir tipos dependientes, CoC resulta sumamente expresivo, constituyéndose en un modelo adecuado para desarrollar programas certificados, permitiendo especificar, programar y demostrar la validez de un programa dentro del mismo formalismo.

El Cálculo de Construcciones Inductivas [28] extiende CoC con la noción primitiva de definición inductiva. Posteriormente, este formalismo fue extendido para formalizar tipos coinductivos [15].

1.3. Coq

Coq es un asistente de pruebas basado en el Cálculo de Construcciones inductivas, diseñado para el desarrollo de pruebas matemáticas y la verificación formal de programas. El lenguaje de especificación *Gallina* provisto por *Coq* es un lenguaje funcional con tipos dependientes que extiende CIC proporcionando un mecanismo de módulos y secciones que permite estructurar convenientemente las teorías en estudio. El sistema viene acompañado de una biblioteca estándar de teorías entre las que se destacan teorías sobre el cálculo de predicados, aritmética natural y entera, listas, streams, y una axiomatización de los números reales.

No pretendemos en esta sección describir exhaustivamente las características de *Coq*. Para una descripción precisa y detallada del sistema, se puede consultar [7], que introduce progresivamente al lector en el uso del sistema y presenta de forma amena el cálculo de construcciones inductivas. Los tutoriales [6, 16] permiten comenzar a interactuar rápidamente con el sistema.

Presentamos brevemente algunas características de *Coq* que consideramos relevantes para la formalización presentada en este trabajo.

Construcción de pruebas *Coq* permite la construcción de demostraciones formales de manera interactiva a través de *tácticas*. Una táctica puede verse como la aplicación de una regla al estilo de deducción natural. Algunas tácticas permiten construir pruebas de forma automática, en casos donde el problema es decidible (por ejemplo, lógica proposicional). Adicionalmente, el usuario puede definir sus propias tácticas, a partir de un lenguaje diseñado para tal fin.

Extracción de Programas La distinción entre programas y pruebas a nivel de tipos de CIC, en base a los sorts *Set* y *Prop*, hace posible la extracción del componente computacional de los programas especificados y verificados en Coq. El mecanismo de extracción permite obtener programas funcionales en los lenguajes OCaml, Haskell y Scheme.

Tipos coinductivos Estos son tipos recursivos que pueden contener objetos infinitos, no bien fundados. En *Coq* es posible definir y demostrar propiedades sobre esta clase de objetos. En este trabajo utilizaremos esta clase de tipos para la formalización de trazas de ejecución y en la definición de la semática de operadores temporales.

2

Alternating Time Temporal Logic (ATL)

La lógica ATL [2, 4] es una extensión de CTL adecuada para la modelización de sistemas multiagentes abiertos. Mientras que CTL permite cuantificación existencial y universal sobre las posibles trazas de ejecución del sistema, ATL introduce una nueva clase de lógica temporal, que permite realizar cuantificaciones *selectivas* sobre aquellas trazas del sistema que son posibles resultados de un juego, en el cual el sistema y el entorno alternan movimientos o acciones. Por ejemplo, precediendo la fórmula temporal *"eventualmente φ"* por un cuantificador selectivo, podemos especificar que en el juego entre el sistema y el entorno, el sistema tiene una estrategia para alcanzar un estado en el cual φ es válida, *independientemente* de cómo se comporte el entorno.

En la especificación de sistemas abiertos, habitualmente se distingue entre el *no determinismo interno*, elecciones tomadas por el sistema; y *no determinismo externo*, decisiones tomadas por el entorno. De esta manera, además de las clásicas preguntas que uno puede formularse en CTL (¿Satisfacen todas las trazas del sistema φ?, ¿Existe alguna traza del sistema que satisfaga φ?), surge naturalmente una nueva pregunta *¿Puede el sistema resolver su no determinismo interno, de forma tal que la propiedad φ sea válida, independientemente de cómo el entorno resuelva el no determinismo externo?* Esta pregunta puede verse como una condición de triunfo sobre un juego entre el sistema y el entorno.

El modelo semántico de ATL, las estructuras de juegos concurrentes [2], provee una semántica unificada para la especificación de sistemas abiertos.

El resto del capítulo se organiza como sigue. En la sección 2.1 presentamos las estructuras de juego concurrentes. Las secciones 2.2 y 2.3 describen la sintaxis y semántica de la lógica. La sección 2.4 presenta una axiomatización completa y consistente de ATL, que será de especial interés en nuestra formalización. Finalmente, en la sección 2.5 se discuten

algunos aspectos relacionados con la verificación autómatica de ATL.

2.1. Estructuras de Juegos Concurrentes

Originalmente, las fórmulas de ATL eran interpretadas sobres *Alternating Transitions Systems* (ATS) [4]; luego generalizadas en [2] a Estructuras de Juegos Concurrentes. Este último será el modelo semántico que utilizaremos en el resto del trabajo.

Un juego concurrente se desarrolla sobre un espacio de estados. En cada paso del juego, cada uno de los componente escoge uno de sus movimientos permitidos, lo cual determina unívocamente el estado siguiente. Clases particulares de estructuras de juegos concurrentes son

Síncronos, basados en turnos: En cada estado del sistema, sólo un jugador tiene más de una alternativa, y este jugador es determinado por el estado actual del sistema

Asíncronos, basados en turnos: En cada estado del sistema, sólo un jugador tiene más de una alternativa, y este jugador es determinado por un scheduler (que, generalmente, satiface ciertas restricciones de equidad)

Sobre cada juego concurrente, podemos definir un conjunto de fórmulas proposicionales Π. Dichas fórmulas se llaman *observables*, y serán las proposiciones atómicas sobre las cuales se interpretarán las fórmulas de ATL. Asumiremos que en cada estado se puede decidir la validez de cada una de estas proposiciones atómicas.

Formalmente, una estructura de juegos concurrente se define como sigue:

Definición 2.1.1 (Estructura de Juego Concurrente) *Una estructura de juego concurrente (CGS) es una tupla* $S = \langle k, Q, \Pi, \pi, d, \delta \rangle$ *con las siguientes componentes:*

- *un número natural* k*, que representa el conjunto de jugadores (componentes, agentes) del sistema. Identificaremos cada jugador con los números* $1, \ldots, k$
- *Un conjunto finito de estados* Q
- *Un conjunto finito de proposiciones* Π*, también llamadas observables*
- *Para cada estado* q*, un conjunto* $\pi(q) \subseteq \Pi$ *de proposiciones atómicas verdaderas en el estado* q*. La función* π *es llamada función de etiquetado (u observación).*

- *Para cada jugador a y cada estado $q \in Q$, un número natural $d_a(q) \geq 1$ de movimientos permitidos en el estado q para el jugador a. Si $q \in Q$, un* vector de movimientos permitidos en q *es una tupla de la forma $\langle j_1, \ldots, j_k \rangle$ tal que $1 \leq j_a \leq d_a(q)$ para cada jugador a. Llamamos $D(q)$ al conjunto $\{1, \ldots, d_1(q)\} \times \cdots \times \{1, \ldots, d_k(q)\}$ de vectores de movimientos permitidos en q. La función D es llamada* función de movimientos*, y describe todas las posibles "jugadas" que pueden realizarse en q*

- *Para cada estado $q \in Q$ y cada vector $\langle j_1, \ldots, j_k \rangle \in D(q)$, un estado $\delta(q, j_1, \ldots, j_k) \in Q$, que es el estado* sucesor *de q si cada jugador a elige el movimiento j_a. La función δ es llamada* función de transición

Sean $q, q' \in Q$, decimos que q' es un *estado sucesor* de q (o un *q-sucesor*) si siempre que el juego se encuentra en el estado q, las componentes del sistema pueden elegir movimientos tales que q' sea el próximo estado. Más formalmente, q' es sucesor de q si existe un vector de movimientos $\langle j_1, \ldots, j_k \rangle \in D(q)$ tal que $q' = \delta(q, j_1, \ldots, j_k)$.

Una *traza* de S es una secuencia infinita $\lambda = q_0, q_1, q_2, \ldots$ de estados tal que para toda posición $i \geq 0$ el estado q_{i+1} es sucesor del estado q_i. A las trazas cuyo estado inicial es q las llamaremos *q-trazas*. Además, si λ es una traza e $i \geq 0$, notamos $\lambda[i]$, $\lambda[0, i]$, $\lambda[i, \infty]$ a la i-ésima posición de λ, el prefijo $q_0, \ldots, q_i$ de λ y el sufijo $q_i, q_{i+1}, \ldots$ de λ, respectivamente.

Presentamos ahora un ejemplo simple de estructura de juego concurrente.

Ejemplo 2.1.2 *Consideremos un sistema con dos procesos, a y b; y dos variables booleanas x e y. El proceso a controla el valor de x. Cuando $x = false$, a puede mantener su valor o cambiarlo a* true*. Si $x = true$, entonces a no puede cambiar su valor. El proceso b controla exactamente de la misma manera el valor de y. Definimos una estructura de juego concurrente $S = \langle k, Q, \Pi, \pi, d, \delta \rangle$ como modelo del sistema propuesto, donde:*

- *$k = 2$. El jugador 1 representa el proceso a, y el jugador 2 el proceso b.*

- *$Q = \{q, q_x, q_y, q_{xy}\}$. El estado q corresponde a $x = y = false$; q_x refiere al estado del sistema donde $x = true$ e $y = false$; q_y se corresponde con $y = true$ y $x = false$; y finalmente q_{xy} es un estado donde ambas variables toman el valor true*

- *$\Pi = \{x, y\}$. Ambas variables son* observables

- *De las interpretaciones dadas a cada estado, surge naturalmente la siguiente función de etiquetado: $\pi(q) = \emptyset$, $\pi(q_x) = \{x\}$, $\pi(q_y) = \{y\}$, $\pi(q_{xy}) = \{x, y\}$*

- *Los movimientos permitidos para cada jugador en cada uno de los estados de Q son los siguientes:*
  - $d_1(q) = d_1(q_y) = 2$*; tanto en q como en q_y, el movimiento* 1 *para a representa* mantener el valor de x*, mientras que el movimiento* 2 *significa* cambiar el valor de x*. En cambio, $d_1(q_x) = d_1(q_{xy}) = 1$; tanto en q_x como en q_{xy}, la única opción para a es mantener el valor de x en true.*
  - $d_2(q) = d_2(q_x) = 2$*; $d_2(q_y) = d_2(q_{xy}) = 1$: Análogamente, en los estados q y q_x el proceso b puede mantener (movimiento* 1*) o cambiar (movimiento* 2*) el valor de y, mientras que en q_y y q_{xy} sólo puede mantenerlo.*
- *La función δ de transiciones se define como sigue*
  - *El estado q tiene cuatro sucesores: $\delta(q,1,1) = q$, $\delta(q,1,2) = q_y$, $\delta(q,2,1) = q_x$ y $\delta(q,2,2) = q_{xy}$*
  - *El estado q_x tiene dos sucesores: $\delta(q,1,1) = q_x$ y $\delta(q,1,2) = q_{xy}$.*
  - *El estado q_y tiene dos sucesores: $\delta(q,1,1) = q_y$ y $\delta(q,,1) = q_{xy}$.*
  - *El estado q_{xy} tiene sólo un sucesor: $\delta(q,1,1) = q_{xy}$*

Supongamos ahora que el proceso b puede cambiar el valor de y de false a true sólo cuando el valor de x es true. La estructura de juegos concurrentes S' que resulta de esta modificación difiere de S sólo en la función de movimientos. Más precisamente, $d_2(q) = 1$. Observemos que $q, q_x, q_x, q_x, q_{xy}^{\omega}$ es una posible traza de S y S', pero $\langle q, q_y, q_y, q_{xy}^{\omega}\rangle$ y $\langle q, q_{xy}^{\omega}\rangle$ no son posibles trazas de S', aunque sí de S.

Por último, consideremos el caso en que b puede cambiar el valor de y de false a true cuando x tiene el valor true, o cuando simultáneamente x toma dicho valor. En este caso, la estructura S'' resultante difiere de S sólo en la función de transición, donde $\delta''(q,1,2) = q$. Intuitivamente, en el estado q el primer movimiento del proceso b sigue representando la elección "mantener el valor de y"*, mientras que el segundo pasa a representar la acción* "cambiar y si el proceso a cambia x al mismo tiempo, de otra forma mantener su valor"*.*

Ejemplo 2.1.3 *Tomaremos un ejemplo de [2], que formaliza mediante una estructura de juegos concurrentes un protocolo que controla un paso a nivel. Definimos la estructura $S_T = \langle k, Q, \Pi, \pi, d, \delta\rangle$ tal que:*

- $k = 2$, *El jugador* 1 *será el tren, mientras que el* 2 *representará al controlador de la compuerta*
- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Pi = \{Out\_of\_gate, In\_gate, Requested, Granted\}$
- - $\pi(q_0) = \{Out\_of\_gate\}$
  - $\pi(q_1) = \{Out\_of\_gate, Requested\}$
  - $\pi(q_2) = \{Out\_of\_gate, Granted\}$
  - $\pi(q_3) = \{In\_gate\}$
- *En cada estado, sólo un jugador tiene más de un movimiento permitido (a esta clase de estructuras se las denomina* turn-based synchronous game structures*). La especificación de la función d es la siguiente:*
  - $d_1(q_0) = 2$ *y* $d_2(q_0) = 1$. *En el estado* q_0 *el tren tiene dos opciones: mantenerse fuera de la compuerta (movimiento 1), o solicitar la entrada al paso a nivel (movimiento 2).*
  - $d_1(q_1) = 1$ *y* $d_2(q_1 = 3)$. *En* q_1 *es el turno del controlador, que puede elegir entre otorgar el permiso para entrar (movimiento 1), rechazarlo (movimiento 2), o retrasar el manejo del pedido (movimiento 3)*
  - $d_1(q_2) = 2$ *y* $d_2(q_2) = 1$. *En el estado* q_2 *es turno del tren, que tiene la opción de: entrar a la compuerta (movimiento 1), o renunciar al permiso otorgado (movimiento 2).*
  - $d_1(q_3) = 1$ *y* $d_2(q_3 = 2)$. *Nuevamente es el turno del controlador, que puede: mantener la compuerta cerrada (movimiento 1) o reabrirla para recibir nuevos pedidos (movimiento 2).*

- $\delta(q_0, 1, 1) = q_0$ *y* $\delta(q_0, 2, 1) = q_1$
- $\delta(q_1, 1, 1) = q_2$, $\delta(q_1, 1, 2) = q_0$ *y* $\delta(q_1, 1, 3) = q_1$
- $\delta(q_2, 1, 1) = q_3$ *y* $\delta(q_2, 2, 1) = q_0$
- $\delta(q_3, 1, 1) = q_3$ *y* $\delta(q_3, 1, 2) = q_0$

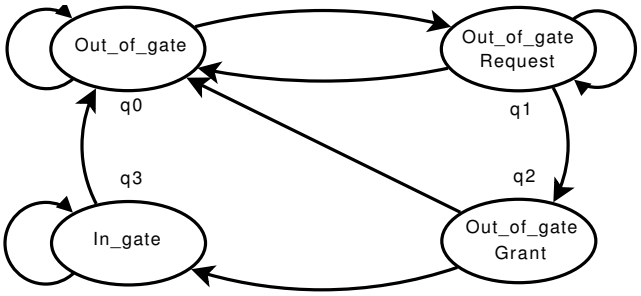

Figura 2.1: *Representación gráfica de la estructura de juego concurrente correspondiente al ejemplo 2.1.3*

2.2. Sintaxis

La sintaxis de ATL se define respecto a un conjunto de proposiciones y un conjunto de jugadores como sigue

Definición 2.2.1 (ATL) *Sea* Π *un conjunto finito de proposiciones atómicas, y* Σ *un conjunto finito de jugadores. El conjunto de fórmulas de la lógica ATL se define inductivamente mediante las siguientes reglas*

(S1) *p, para cualquier proposición* $p \in \Pi$,

(S2) $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \rightarrow \psi$, *donde* φ, ψ *son fórmulas de ATL*

(S3) $\langle\langle A\rangle\rangle \bigcirc \varphi$, $\langle\langle A\rangle\rangle \Box \varphi$ $\langle\langle A\rangle\rangle \varphi \mathcal{U} \psi$, *donde* φ, ψ *son fórmulas de ATL, y* $A \subseteq \Sigma$,

El operador $\langle\langle\ \rangle\rangle$ es un *cuantificador de trazas*; $\bigcirc$ (*next*), $\Box$ (*always*) y $\mathcal{U}$ (*until*) son *operadores temporales.*

2.3. Semántica

La fórmulas de la lógica ATL se interpretan sobre estados de estructuras de juegos concurrentes con las mismas proposiciones atómicas Π y jugadores $\Sigma = \{1, \ldots, k\}$.

La función de etiquetado nos provee una interpretación para las proposiciones atómicas. Los conectivos lógicos tienen su significado usual.

Para evaluar una fórmula de la forma $\langle\langle A\rangle\rangle\varphi$ en un estado q de una CGS, consideremos el siguiente juego entre un protagonista y su oponente. El juego comienza en

q y posee una secuencia infinita de rondas. Sea s una posición arbitraria del juego. Para calcular la posición siguiente, el protagonista elige para cada uno de los jugadores $a \in A$ un movimiento $j_a \in \{1, \ldots, d_a(s)\}$. Luego, el oponente elige para cada jugador $b \in \Sigma \setminus A$ un movimiento $j_b \in \{1, \ldots, d_b(s)\}$. La nueva posición resulta $\delta(s, j_1, \ldots, j_k)$. De esta forma, el juego continua y produce una traza λ. El protagonista gana el juego si λ satisface la fórmula φ, leída como una fórmula temporal lineal cuyo operador más externo es $\bigcirc$, $\mathcal{U}$ o $\Box$. Si existe una estrategia ganadora para el protagonista, decimos que la fórmula $\langle\langle A \rangle\rangle \varphi$ es válida.

Con el objetivo de definir formalmente la semántica de la lógica ATL, necesitamos más precisión sobre el concepto de estrategia en estructuras de juegos concurrentes. Sea $S = \langle k, Q, \Pi, \pi, d, \delta \rangle$ una CGS, una *estrategia* para un jugador $a \in \Sigma$ es una función f_a que mapea cada secuencia no vacía de estados $\alpha \in Q^+$ a un número natural $f_a(\alpha)$, tal que $f_a(\alpha) \leq d_a(q)$, donde q es el último estado de α, i.e., $\alpha = \alpha' \cdot q$. Es decir, una estrategia f_a decide el próximo movimiento para a, teniendo en cuenta la secuencia de estados por las que el sistema ha pasado.

Una estrategia f_a para $a \in \Sigma$ induce un conjunto de trazas que a puede forzar siguiendo los movimientos dados por la estrategia. Sean $q \in Q$, $A \subseteq \Sigma$ y un conjunto $F_A = \{f_a \mid a \in A\}$, definimos el conjunto $out(q, F_A)$ de q-trazas que los jugadores en A pueden forzar siguiendo las estrategias en F_A de la siguiente manera: $\lambda = q_0, q_1, q_2, \ldots \in out(q, F_A)$ si $q_0 = q$ y para todas las posiciones i existe un vector de movimientos $\langle j_1, \ldots, j_k \rangle \in D(q_i)$ tal que $(\forall a \in A, j_a = f_a(\lambda[0..i])) \wedge \delta(q_i, j_1, \ldots, j_k) = q_{i+1}$.

A partir de estas definiciones, podemos dar precisión a lo expresado anteriormente sobre la validez de una fórmula ATL en un estado de una estructura de juegos concurrentes, lo cual haremos en la siguiente definición.

Definición 2.3.1 (Semántica de ATL) *Sea $S = \langle k, Q, \Pi, \pi, d, \delta \rangle$ una estructura de juego concurrente, $q \in Q$ y φ una fórmula de la lógica ATL. Decimos que* φ es válida en el estado q de la estructura S *si $S, q \models \varphi$, donde la relación $\models$ está definida por inducción en la estructura de φ de la siguiente manera:*

- $S, q \models p$, *para* $p \in \Pi$ *sii* $p \in \pi(q)$
- $S, q \models \neg\varphi$ *sii* $S, q \not\models \varphi$
- $S, q \models \varphi_1 \vee \varphi_2$ *sii* $S, q \models \varphi_1$ *o* $S, q \models \varphi_2$

- $S, q \models \varphi_1 \wedge \varphi_2$ *sii* $S, q \models \varphi_1$ *y* $S, q \models \varphi_2$
- $S, q \models \varphi_1 \Rightarrow \varphi_2$ *sii* $S, q \models \varphi_2$ *siempre que* $S, q \models \varphi_1$
- $S, q \models \langle\langle A \rangle\rangle \bigcirc \varphi$ *sii existe un conjunto* $F_A = \{f_a \mid a \in A\}$ *de estrategias, tal que para toda traza* $\lambda \in out(q, F_A)$ *se cumple* $S, \lambda[1] \models \varphi$
- $S, q \models \langle\langle A \rangle\rangle \Box \varphi$ *sii existe un conjunto* $F_A = \{f_a \mid a \in A\}$ *de estrategias, tal que para toda traza* $\lambda \in out(q, F_A)$ *y todas las posiciones* $i \geq 0$ *se cumple* $S, \lambda[i] \models \varphi$
- $S, q \models \langle\langle A \rangle\rangle \varphi_1 \,\mathcal{U}\, \varphi_2$ *sii existe un conjunto* $F_A = \{f_a \mid a \in A\}$ *de estrategias, tal que para toda traza* $\lambda \in out(q, F_A)$ *existe una posición* $i \geq 0$ *tal que* $S, \lambda[i] \models \varphi_2$ *y para todas las posiciones* $0 \leq j < i$ *se cumple* $S, \lambda[j] \models \varphi_1$

Cuando la estructura S *se deduce del contexto, escribiremos directamente* $q \models \varphi$ *. Decimos que* φ *es* válida en S *si, para todo estado* $q \in Q$ *se cumple* $S, q \models \varphi$ *. Además, decimos que* φ *es* lógicamente válida*, o simplemente* válida *si para cualquier estructura de juego concurrente* S *y cualquier estado* q *de* S*,* $S, q \models \varphi$ *.*

Del mismo modo, decimos que φ *es* satisfactible *si existen una estructura de juegos concurrentes* S *y un estado* q *de* S *tales que* $S, q \models \varphi$ *.*

Ejemplo 2.3.2 *Consideremos las estructuras* S*,* S' *y* S'' *definidas en el ejemplo 2.1.2. La fórmula* $S, q \models \langle\langle 2 \rangle\rangle \bigcirc y$ *es válida, dado que el jugador* 2 *puede elegir su segundo movimiento para asignar el valor true a* y*. En oposición,* $S', q \not\models \langle\langle 2 \rangle\rangle \bigcirc y$ *, pues dicho movimiento no está permitido para* 2 *en* S'*. Observemos que* $S, q \not\models \langle\langle 2 \rangle\rangle \bigcirc (x = y)$ *, pues si el jugador* 2 *elige en* q *el movimiento* 1*, puede darse el caso que el jugador* 1 *elija el movimiento* 2*, y en el estado resultante valga* $x \neq y$*. Sin embargo,* $S'', q \models \langle\langle 2 \rangle\rangle \bigcirc (x = y)$ *, debido a que el jugador* 2 *puede elegir su el movimiento* 2 *en* q*, asegurando que* x *e* y *tendrán el mismo valor en el siguiente estado, que puede ser* q *o* q_{xy}*.*

2.4. Axiomatización

Cuando el número de estados del sistema no es acotado, o la especificación es paramétrica en alguno de sus atributos (por ejemplo, no sabemos *a priori* el número de jugadores), la verificación automática del sistema no es posible. La definición de un sistema

deductivo para ATL nos permite demostrar propiedades generales de la lógica, que luego pueden ser instanciadas a modelos particulares.

Contar con una sistema formal para ATL permite abordar cuestiones lógicas fundamentales sobre ATL, tales como completitud, consistencia y decidibilidad.

En esta sección incluimos una axiomatización completa y consistente para ATL, tomada de [17]. Presentamos únicamente los axiomas y reglas de inferencia; el lector interesado en las demostraciones de completitud, consistencia y decidibilidad de ATL puede consultar dicho trabajo.

Definición 2.4.1 *El sistema axiomático para ATL consiste en los siguientes axiomas y reglas de inferecia, donde* $A, A_1, A_2 \subseteq \Sigma$*:*

Axiomas

(TP) *Un conjunto adecuado de tautologías proposicionales*

(⊥) $\neg\langle\langle A\rangle\rangle\bigcirc\bot$

(⊤) $\langle\langle A\rangle\rangle\bigcirc\top$

(Σ) $\neg\langle\langle\varnothing\rangle\rangle\bigcirc\neg\varphi \rightarrow \langle\langle\Sigma\rangle\rangle\bigcirc\varphi$

(S) $\langle\langle A_1\rangle\rangle\bigcirc\varphi_1 \wedge \langle\langle A_2\rangle\rangle\bigcirc\varphi_2 \rightarrow \langle\langle A_1 \cup A_2\rangle\rangle\bigcirc\varphi_1 \wedge \varphi_2,$ *si* $A_1 \cap A_2 = \varnothing$

(FP$_\Box$) $\langle\langle A\rangle\rangle\Box\varphi \leftrightarrow \varphi \wedge \langle\langle A\rangle\rangle\bigcirc\langle\langle A\rangle\rangle\Box\varphi$

(GFP$_\Box$) $\langle\langle\varnothing\rangle\rangle\Box\,(\theta \rightarrow (\varphi \wedge \langle\langle A\rangle\rangle\bigcirc\theta)) \rightarrow \langle\langle\varnothing\rangle\rangle\Box\,(\theta \rightarrow \langle\langle A\rangle\rangle\Box\varphi)$

(FP$_\mathcal{U}$) $\langle\langle A\rangle\rangle\varphi_1\,\mathcal{U}\,\varphi_2 \leftrightarrow \varphi_2 \vee (\varphi_1 \wedge \langle\langle A\rangle\rangle\bigcirc\langle\langle A\rangle\rangle\varphi_1\,\mathcal{U}\,\varphi_2)$

(LFP$_\mathcal{U}$) $\langle\langle\varnothing\rangle\rangle\Box\,((\varphi_2 \vee (\varphi_1 \wedge \langle\langle A\rangle\rangle\bigcirc\theta)) \rightarrow \theta) \rightarrow \langle\langle\varnothing\rangle\rangle\Box\,(\langle\langle A\rangle\rangle\varphi_1\,\mathcal{U}\,\varphi_2 \rightarrow \theta)$

Reglas de inferencia

(Modus Ponens) $$\frac{\varphi_1 \qquad \varphi_1 \rightarrow \varphi_2}{\varphi_2}$$

(Monotonía de $\langle\langle A\rangle\rangle\bigcirc$) $$\frac{\varphi_1 \rightarrow \varphi_2}{\langle\langle A\rangle\rangle\bigcirc\varphi_1 \rightarrow \langle\langle A\rangle\rangle\bigcirc\varphi_2}$$

($\langle\langle\varnothing\rangle\rangle\Box$ - Inevitable) $$\frac{\varphi}{\langle\langle\varnothing\rangle\rangle\Box\varphi}$$

EL axioma ($\bot$) establece que ninguna coalición puede cooperar para hacer cierta la fórmula $\bot$. Similarmente, ($\top$) asegura la validez de la fórmula $\top$ en cualquier estado alcanzable en un paso. El axioma (Σ) establece que todos los componentes del sistema pueden colaborar para que en el próximo estado φ se cumpla, siempre y cuando $\neg\varphi$ no sea inevitable. (S) puede verse como un axioma de cooperación. En efecto, si A_1 puede asegurar la validez de φ_1 en el próximo estado, y A_2 puede hacer lo propio con φ_2, entonces los agentes en ambas coaliciones pueden cooperar para asegurar la validez de $\varphi_1 \wedge \varphi_2$.

El operador $\langle\langle\rangle\rangle\Box$ se introduce mediante dos axiomas. La fórmula ($\mathrm{PF}_\Box$) lo define como un punto fijo de la ecuación

$$X \leftrightarrow \varphi \wedge \langle\langle A\rangle\rangle\bigcirc X,$$

mientras que el axioma ($\mathrm{GFP}_\Box$) establece que $\langle\langle\rangle\rangle\Box$ es el mayor punto fijo de dicha ecuación.

El operador $\langle\langle\rangle\rangle\mathcal{U}$ se introduce de forma similar. El axioma ($\mathrm{FP}_\mathcal{U}$) define dicho operador como un punto fijo de la ecuación

$$X \leftrightarrow \varphi_2 \vee (\varphi_1 \wedge \langle\langle A\rangle\rangle\bigcirc X)$$

A diferencia del operador *always*, $\langle\langle\rangle\rangle\mathcal{U}$ es el *menor* punto fijo de esta ecuación, lo cual se asegura mediante el axioma ($\mathrm{LFP}_\mathcal{U}$).

En cuanto a las reglas de inferencia, la monotonía del operador $\langle\langle\rangle\rangle\bigcirc$ permite demostrar la monotonía del resto de los operadores temporales, dado que éstos se encuentran definidos mediante ecuaciones en función de *next*. La regla ($\langle\langle\varnothing\rangle\rangle\Box$ - *Inevitable*) asegura que, si una fórmula es válida en todos los estados, entonces será válida a lo largo de cualquier traza de ejecución.

2.5. Verificación automática

Sea S una estructura de juegos concurrente finita, y φ una fórmula de ATL. Un *model checker* para ATL determina de manera automática todos los estados q de S que satisfacen la relación $S, q \models \varphi$. Para ATL, este problema es decidible en tiempo polinomial. En efecto, en [2] se presenta un algoritmo de model checking simbólico que resuelve dicho

problema en un tiempo proporcional a $m \times l$, donde m es el número de transiciones de S, y l la longitud de la fórmula a verificar.

La herramienta MOCHA [3, 5] implementa un algoritmo de model checking basado en OBDD (diagramas de decisión binaria ordenados) para ATL, que representa estructuras de juegos concurrentes mediante *Módulos Reactivos*, un lenguaje de especificación para sistemas concurrentes. MOCHA ha sido utilizado con éxito en la verificación de protocolos criptográficos [8, 22], así como en la verificación del chip paralelo **VSI** [18], que contiene un total de 96 procesadores y alrededor de 6 millones de transistores.

Aunque en ciertas ocasiones estamos interesados sólo en determinar la validez de una fórmula, puede darse el caso en que sea necesario saber por qué la fórmula es válida, y qué construcciones han sido necesarias para verificarlo. Por ejemplo, al verificar $S, q \models \langle\langle A \rangle\rangle \Box \varphi$, nos interesaría conocer qué estrategia es la que les permite a los componentes en A asegurar la validez de φ. En [21] se propone una extensión al algoritmo de model checking para ATL que permite extraer estrategias a partir de la verificación de una fórmula.

Si bien el problema de model checking para ATL pertenece a la clase de comlejidad P, debe remarcarse que, al igual que para los modelos de Kripke usados en la verificación de CTL, las estructuras de juego concurrentes padecen del problema de la explosión de estados, es decir, la cantidad de estados crece exponencialmente respecto del número de variables proposicionales atómicas, y por lo tanto la exploración del espacio de estados puede convertirse en un problema intratable.

3

Formalización

En este capítulo presentamos una formalización en el cálculo de construcciones coinductivas de la lógica ATL. La sección 3.1 introduce algunas definiciones y tipos de datos que se utilizarán en el resto de los capítulos, así como la notación que utilizaremos con el fin de facilitar la lectura del trabajo.

En la sección 3.2, formalizamos el modelo semántico de ATL, las estructuras de juegos concurrentes. Nuestra formalización introduce un modelo más general que el presentado en [2]. Mientras que en la definición original el conjunto de estados y jugadores es finito, en este trabajo intruducimos dichos conjuntos mediante tipos en el sort *Set*, sin restringir la cardinalidad de estos conjuntos. Consideramos que tal restricción no es necesaria al razonar en un sistema deductivo. Esta generalización nos permitirá razonar sobre sistemas no acotados o paramétricos, lo cual no es posible (sin reducciones o simplificaciones previas) siguiendo la definición original de CGS.

En la sección 3.3 se formalizan las fórmulas de ATL. Al igual que para las estructuras de juegos concurrentes, generalizaremos la semántica original en ciertos aspectos. Las proposiciones atómicas se introducen mediante relaciones unarias sobre estados, permitiendo que las fórmulas de estados sean proposiciones de alto orden. Esta generalización le otorga mayor expresividad a la lógica, ampliando la clase de propiedades que pueden ser demostradas.

Los conectivos lógicos se formalizan de manera usual, utilizando los operadores inductivos predefinidos en Coq. El operador *next* se introduce como un operador *local*, empleando las nociones de estrategias y sucesor definidas en la sección 3.2 . La formalización de los operadores *until* y *always* se basan en las propiedades de punto fijo que poseen (ver sección 2.4, [17]). El operador *until*, al ser el *menor* punto fijo de la ecuación que lo

caracteriza, se formaliza mediante un tipo inductivo. El operador *always* está definido como el *mayor* punto fijo de una ecuación, y por lo tanto consideramos que su formalización natural es mediante la utilización de tipos coinductivos. Finalmente, introducimos algunos operadores derivados de ATL.

3.1. Tipos básicos y notación utilizada

La formalización presentada en este trabajo ha sido desarrollada por completo en *Gallina*, el lenguaje de especificación de *Coq*. Por razones de claridad, haremos uso de una notación simplificada para introducir los términos de la formalización.

Conectivos lógicos y Cuantificadores Se utiliza la notación estándar para los conectivos del cálculo de predicados: conjunción ($\wedge$), disyunción ($\vee$), negación ($\neg$), implicación ($\rightarrow$), cuantificación universal ($\forall$) y cuantificación existencial ($\exists$).

Funciones y Predicados Un predicado es una función cuyo tipo final es *Prop*. La notación para funciones que utilizamos en este trabajo es similar a la de *Gallina*. A una función (o predicado) de tipo $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow A$ la notaremos

$$\mathit{fun}\ (x_1 : T_1)(x_2 : T_2) \ldots (x_n : T_n) \Rightarrow e$$

Por ejemplo, el predicado *pos*: $\mathbb{N} \rightarrow \mathit{Prop}$, que es válido cuando su argumento es positivo, se define de la siguiente manera: $(\mathit{fun}\ (n : \mathbb{N}) \Rightarrow n > 0)$.

Omitiremos los tipos de los argumentos cuando estos resulten evidentes en el contexto de la definición.

Tipos Inductivos Introduciremos tipos inductivos declarando

- su aridad (una aridad es un tipo convertible a un tipo producto cuya conclusión es un sort), y
- el tipo de sus constructores

Por ejemplo, el tipo de las listas parametrizadas por el tipo de sus elementos puede definirse inductivamente dando su aridad, $\mathit{list} : \mathit{Set} \rightarrow \mathit{Set}$, y el tipo de sus constructores $\mathit{nil} : \forall A : \mathit{Set}, \mathit{list}A$, y $\mathit{cons} : \forall A : \mathit{Set}, A \rightarrow \mathit{list}A \rightarrow \mathit{list}A$.

Sea I un tipo inductivo con aridad

$$\forall(p_1 : A_1) \dots (p_m : A_m)(x_1 : T_1) \dots (x_n : T_n), s$$

donde los primeros m productos corresponden a sus parámetros. Cuando sea posible, declararemos un constructor c de I con tipo

$$\forall(p_1 : A_1) \dots (p_m : A_m)(x_1 : T_1) \dots (x_j : T_j), P_1 \to \dots \to P_k \to I\ p_1\ \dots p_m\ t_1 \dots t_n$$

utilizando una regla de introducción de la forma

$$\frac{P_1, \dots, P_k}{I\ p_1 \dots p_m\ t_1 \dots t_n}\ \text{(c)}$$

donde las ocurrencias libres de variables (correspondientes a $x_1 \dots x_j$) se consideran cuantificadas universalmente, y su tipo puede inferirse del contexto.

Secuencias El tipo de las secuencias finitas parametrizadas por el tipo de sus elementos se define mediante el tipo inductivo $\text{seq} : \mathit{Set} \to \mathit{Set}$, con constructores $\mathit{nil} : \forall A : \mathit{Set}, \text{seq}\ A$ y $\mathit{add} : \forall A : \mathit{Set}, \text{seq}\ A \to A \to \text{seq}\ A$. Utilizaremos la notación $\langle\rangle$ para *nil* y $(s \frown a)$ para $(\mathit{add}\ s\ a)$. Si s_1, s_2 son secuencias de un tipo A, notaremos $s_1 \ominus s_2$ a la concatenación de s_1 con s_2.

Valores certificados El tipo *sig* permite combinar un tipo de datos y un predicado sobre este tipo, creando el tipo de los *valores que satisfacen el predicado*. Formalmente, $\mathit{sig} : \forall(A : \mathit{Set})(P : A \to \mathit{Prop}) \to \mathit{Set}$ se intruduce a partir del constructor $\mathit{exist} : \forall(x : A), P\ x \to \mathit{sig}\ A\ P$.

Un elemento del tipo $\mathit{sig}\ A\ P$ se forma a partir de un valor $x : A$ y una prueba de $P\ x$. Emplearemos la notación $\{x : A \mid P\}$ para denotar la expresión $(\mathit{sig}\ A\ ([x : A]P))$. Diremos que $\{x : A \mid P\}$ es el *subconjunto* de los elementos de A que satisfacen la propiedad P.

Suma disjunta El tipo *sum* es la contraparte en *Set* de la disyunción en *Prop*; formalmente, $\mathit{sum} : \mathit{Set} \to \mathit{Set} \to \mathit{Set}$ se introduce mediante los constructores $\mathit{inl} : \forall(A\ B : \mathit{Set}), A \to \mathit{sum}\ A\ B$ y $\mathit{inr} : \forall(A\ B : \mathit{Set}), B \to \mathit{sum}\ A\ B$. Utilizaremos la notación $A + B$ para el tipo $\mathit{sum}\ A\ B$.

Los tipos unit y Emtpy_set El tipo *unit* se introduce mediante el constructor $tt : unit$. Por lo tanto, es siempre posible construir elementos en dicho conjunto (este tipo puede verse como la contraparte de *True* en *Prop*). El tipo *Empty_set* no posee constructores, y por lo tanto no es posible generar elementos en él. Utilizaremos la notación $\{1\}$ y $\{\ \}$ para referirnos a *unit* y *Empty_set* respectivamente.

Secuencias Infinitas El tipo coinductivo $Stream : Set \to Set$ se introduce con un único constructor: $Cons : \forall(A : Set), A \to Stream\ A \to Stream\ A$. Si $A : Set$, un elemento del tipo $(Stream\ A)$ será una secuencia infinita de elementos de A. Utilizaremos la notación infija $x \lhd \lambda$ para referirnos a la secuencia $(Cons\ x\ \lambda)$, donde $x : A$ y $\lambda : Stream\, A$.

3.2. Formalizando CGS

3.2.1. Definiciones Básicas

Nuestra especificación asume la presencia de tres tipos básicos en el sort *Set*:

- *State*: El conjunto de estados de la CGS
- *Player*: Conjunto de componentes del sistema
- *Move*: Posibles movimientos para los jugadores

Un vector de movimientos es una función cuyo dominio es el conjunto de componentes del sistema, y codominio el conjunto de movimientos:

$$MoveVector := Player \to Move \tag{3.1}$$

Transiciones

La función de transición es representada mediante una relación con el siguiente tipo

$$trans : State \to MoveVector \to State \to Prop. \tag{3.2}$$

Incorporamos a la teoría dos hipótesis que cualquier relación de transición debe cumplir:

$$\begin{aligned} trans\_f : \forall(q, q', q'' : State)(mv : MoveVector), \\ trans\ q\ mv\ q' \to trans\ q\ mv\ q'' \to q' = q'' \end{aligned} \tag{3.3}$$

que especifica la condición de función de *trans*, y

$$\begin{aligned} &trans\_d : \forall(q : State), \\ &\quad \{(mv, q') : MoveVector \times State \mid trans\ q\ mv\ q'\} \end{aligned} \tag{3.4}$$

que asegura la pertenecia de todos los estados al dominio de la relación de transición.

La relación inductiva $suc : Player \to Player \to Prop$, que especifica cuándo un estado es sucesor de otro, se define con un constructor,

$$\frac{mv : MoveVector \qquad trans\ q\ mv\ q'}{suc\ q\ q'}\ (\text{suc\_}intro) \tag{3.5}$$

El conjunto de movimientos permitidos para un jugador en un determinado estado, que en la definición 2.1.1 se representa mediante la función d, en nuestra formalización puede representarse a partir de las definiciones hasta ahora presentadas, mediante la relación inductiva $moveAv : State \to Player \to Move \to Prop$:

$$\frac{\begin{array}{c} q' : State \qquad mv : MoveVector \\ mv\ a = m \qquad trans\ q\ mv\ q' \end{array}}{moveAv\ q\ a\ m}\ (\text{moveAv\_}intro) \tag{3.6}$$

Una prueba de $(moveAv\ q\ a\ m)$ nos indica que el jugador a puede optar por el movimiento m en el estado q.

3.2.2. Trazas de ejecución, necesidad de tipos coinductivos

Utilizaremos la abreviación

$$Trace := Stream\ State \tag{3.7}$$

para referirnos a secuencias infinitas de estados.

Sea $\lambda = q_0, q_1, q_2, \ldots$ una secuencia infinita de estados ¿Qué condición debe cumplir λ para ser una posible ejecución del sistema? Podemos responder a esta cuestión de al menos dos maneras:

1. Utilizando la indexación implícita que de las trazas puede hacerse, i.e, identificar cada estado con su posición en la traza. De esta manera, la propiedad resultante es:

$$\forall n : \mathbb{N}, suc\ q_n\ q_{n+1}$$

2. Definiendo una propiedad coinductiva, basada en la forma de construcción de λ. (ver (3.1)). Siguiendo esta estrategia, la propiedad resulta:

$$(suc\ q_0\ q_1) \wedge (\langle q_1, q_2, \ldots \rangle\ \textit{es una traza del sistema})$$

Si bien ambas formalizaciones capturan la propiedad en la que estamos interesados, no son equivalentes: la primera es una consecuencia de la segunda, pero la recíproca no es cierta. Es decir, utilizando coinducción obtenemos una propiedad más fuerte.

Adicionalmente, observamos que los tipos coinductivos son utilizados para la formalización de lógicas temporales en el cálculo de construcciones. En [11, 12] se emplean dichas construcciones para representar la noción de trazas de ejecución y ciertos operadores temporales de la lógica LTL. La formalización de CTL presentada en [23, 24] utiliza tipos coinductivos para formalizar los operadores $\forall\Diamond$ y $\exists\Box$, que en ATL resultan equivalentes a $\langle\langle\varnothing\rangle\rangle\Diamond$ y $\langle\langle\Sigma\rangle\rangle\Box$.

Finalmente, durante el proceso de desarrollo de la formalización, hemos trabajado con ambas opciones, construyendo objetos y probando propiedades desde ambos enfoques. Como resultado observamos que los tipos coinductivos capturan mejor las propiedades involucradas en la presente formalización, por lo cual preferiremos especificar propiedades de forma coinductiva cuando resulte conveniente.

El predicado $isTrace : Trace \to Prop$ determina si una traza es una posible ejecución del sistema.

$$\frac{suc\ q\ (hd\ x) \qquad isTrace\ x}{isTrace\ (q \lhd x)}\ (isTrace\_intro) \tag{3.8}$$

A partir de esta definición, el predicado $isTraceFrom : State \to Trace \to Prop$, nos permite capturar la relación *"x es una posible traza de ejecución del sistema si el estado inicial es q"*

$$isTraceFrom(q : State)(x : Trace) := (hd\ x = q) \wedge isTrace\ x \tag{3.9}$$

3.2.3. Coaliciones

Una coalición es un subconjunto de jugadores. La forma usual de representar un conjunto A de elementos en un tipo U es mediante una función cuyo tipo es $U \to Prop$. Decimos que un elemento $u : U$ *pertenece* al conjunto A si podemos exhibir una prueba de

la proposición $A\ a$. De esta manera, el complemento se representa mediante el operador $\neg$ y la unión de dos conjuntos A y B se define mediante el tipo $(fun\ (x : U) \Rightarrow Aa \vee Ba)$. La librería standard de Coq utiliza esta formalización.

Sin embargo, para nuestro propósito esta formalización no es satisfactoria. Para entender por qué, supongamos que queremos probar la propiedad

$$\langle\langle A\rangle\rangle\bigcirc\varphi \to \langle\langle B\rangle\rangle\bigcirc\psi \to \langle\langle A \cup B\rangle\rangle\bigcirc(\varphi \wedge \psi),$$

que es lógicamente válida si $A \cap B = \varnothing$ y forma parte de la axiomatización presentada en [17]. Para ello, deberíamos ser capaces de costruir una estrategia para $A \cup B$, a partir de las estrategias para A y B que nos brindan las premisas. Para cada jugador a perteneciente a $A \cup B$, la estrategia será la siguiente:

- La que nos provee la primer premisa, si $a \in A$
- La que nos provee la segunda premisa, si $a \in B$

  Dado que una estrategia es un objeto que habita el sort *Set*, no es posible hacer un análisis por casos sobre una hipótesis de la forma $(A\ a \vee B\ a)$, pues esta habita habita en *Prop*.

  Teniendo en cuenta que nos interesa definir objetos en *Set* dependiendo del *certificado* de pertenencia a una coalición, dicho certificado debe ser un objeto computacional. Definimos entonces una coalición como un sinónimo del tipo $Player \to Set$:

$$Coalition := Player \to Set \tag{3.10}$$

  Diremos que un jugador a *pertenece* a la coalición A, y lo notaremos $a \in A$ si podemos construir un elemento en $A\ a$.

  Las coaliciones universal y vacía

  La coalición formada por todos los jugadores se define como un tipo inductivo *AllC* con un solo constructor, que para cada jugador $a : A$ construye un *certificado* de $(AllC\ a)$

$$\frac{a : Player}{AllC\ a}\ (AllC\_intro) \tag{3.11}$$

Para definir la coalición vacía *EmptyC*, utilizamos un tipo inductivo sin constructores, de esta manera nunca podremos construir un elemento del tipo (*EmptyC a*) para ningún elemento $a : Player$.

$$EmptyC := . \tag{3.12}$$

En lo que resta del trabajo, utilizaremos los símbolos Σ y $\varnothing$ para referirnos a *AllC* y *EmptyC* respectivamente.

Operaciones y Relaciones entre coaliciones

La operación de *complemento* de una coalición viene dada por la definición

$$Oponent(A : Coalition) : Coalition := fun\ (a : Player) \Rightarrow (A\ a \to \{\}) \tag{3.13}$$

Notaremos a la coalición (*Oponent A*) como $\Sigma \backslash A$.

La *unión* de dos coaliciones se define utilizando el tipo *sum* presentado en la sección 3.1:

$$A \uplus B := fun\ (a : Player) \Rightarrow (A\ a) + (B\ a) \tag{3.14}$$

Definimos una relación de contención entre coaliciones:

$$SubCoalition(A\ B : Coalition) := \forall (a : Player), A\ a \to B\ a. \tag{3.15}$$

Es decir, una coalición A está contenida en B si para cada jugador $a \in A$ podemos demostrar que $a \in B$. Cuando resulte conveniente, utilizaremos la notación $A \subseteq B$ para *SubCoalition A B*

Procedemos ahora a definir la relación *same_coalition* y el axioma *eq_coal*, que introduce la noción de extensionalidad para coaliciones.

$$\frac{SubCoalition\ A\ B \qquad SubCoalition\ B\ A}{same\_coalition\ A\ B}\ (\text{same\_intro})$$
$$eq\_coal : \forall (A\ B : Coalition), same\_coalition\ A\ B \to A = B \tag{3.16}$$

Asumiremos que la pertenencia a una coalición es una propiedad decidible, mediante el siguiente axioma:

$$coal\_dec : \forall(A : Coalition)(a : Player), (A\ a) + (\Sigma \backslash A\ a) \tag{3.17}$$

Observemos que este axioma es análogo al axioma del tercero excluido en *Prop*, y no puede ser demostrado en un sistema constructivo.

3.2.4. Estrategias

Una estrategia es una función que toma la historia del juego, el estado actual y elije un movimiento:

$$Strategy := \text{seq}\ State \to State \to Move \tag{3.18}$$

Consideremos un jugador a siguiendo la estrategia f. Sea q el estado actual del sistema, y qs la secuecia de estados por la que el sistema ha pasado. Nos interesa definir los posibles sucesores de a en q al seguir f, lo cual haremos mediante la relación inductiva

$$sucSt : Player \to Strategy \to \text{seq}\ State \to State \to State \to Prop$$

$$\frac{mv : MoveVector \qquad mv\ a = f\ qs\ q \qquad trans\ q\ m\ q'}{sucSt\ a\ f\ qs\ q\ q'}\ (sucSt\_intro) \tag{3.19}$$

Estrategias para una coalición y el conjunto out$(\mathbf{q}, \mathbf{F_A})$

Estamos ahora interesados en definir la noción de conjunto de estrategias para una coalición $A \subseteq \Sigma$, tal como se describió en la sección 2.3:

$$StrategySet(A : Coalition) := \forall a : Player, A\ a \to Strategy \tag{3.20}$$

Dada una coalición A, un término $F : StrategySet\ A$ es una función que toma un jugador a y un certificado de la pertenencia de a a la coalición A y construye una estrategia para a.

Definiremos el conjunto de posibles q-sucesores de un conjunto de estrategias F_A, habiendo el sistema pasado por la secuencia de estados qs. La relación *"q' es un*

posible sucesor de la secuencia de estados $(qs \frown q)$ *si los jugadores en* A *siguen el conjunto de estrategias* F_A*"*, se formaliza mediante la relación inductiva

$$sucStSet : \forall A : Coalition, StrategySet\ A \to \text{seq}\ State \to State \to State \to Prop,$$

cuyo único constructor es

$$\frac{mv : MoveVector \qquad trans\ q\ mv\ q' \qquad \forall (a : Player)(H : A\ a), F_A\ a\ H\ qs\ q = mv\ a}{sucStSet\ A\ F_A\ qs\ q\ q'}\ (sucStSet\_intro) \tag{3.21}$$

Para que un estado q' cumpla con tal propiedad, debe existir un vector de movimientos mv tal que

- Existe una transición del estado actual a q',
- si $a \in A$ y H es un certificado de esta relación de pertenencia, entonces la estrategia $F\ a\ H$ aplicada a la historia de ejecución (qs) y al estado actual (q) debe coincidir con el movimiento $mv\ a$.

Dado que el primer argumento de *sucStSet*, A, puede deducirse del segundo (F_A), en lo que sigue lo omitiremos. Si tenemos una prueba de $sucStSet\ F_A\ qs\ q\ q'$, diremos que q' pertenece al conjunto de posibles sucesores de $qs \frown q$ si los jugadores en A siguen las estrategias F_A, y lo notaremos $q' \in sucSt(qs, q, F_A)$

Procedemos a formalizar el conjunto de trazas que una coalición A puede forzar si sigue el conjunto de estrategias F_A, definido en la sección 2.3 como $out(q, F_A)$. Las condiciones para que una traza $\lambda = q_0, q_1, q_2, \ldots$ pertenezca a dicho conjunto se interpretan de la siguiente manera:

1. $q_0 = q$,
2. para todas las posiciones i existe un vector de movimientos mv tal que
   - $\forall a \in A, mv\ a\ =\ F\ a\ (\lambda[0..i-1])\ q_i$
   - $trans\ q_i\ mv\ q_{i+1}$

donde, para el caso $i = 0$, $\lambda[0..-1]$ denota la secuencia vacía.

La primer condición se formaliza fácilmente. Para la segunda, y al igual que para la relación *isTrace*, utilizaremos una definición coinductiva. Definimos la relación

$$isOut : \forall A : Coalition, StrategySet\ A \to \text{seq}\ State \to Trace \to Prop$$

tal que una prueba de $isOut\ A\ F_A\ qs\ \lambda$ se traducirá en:

> *"Habiendo el sistema pasado por la secuencia de estados qs, si los jugadores en A siguen las estrategias en F_A, entonces λ es una posible traza de ejecución a partir de qs"*

isOut tendrá un solo constructor:

$$\frac{sucStSet\ F_A\ qs\ q\ q' \qquad isOut\ A\ F\ (qs \frown q)\ (q' \lhd \lambda)}{isOut\ A\ F_A\ qs\ (q \lhd q' \lhd \lambda)}\ (isOut\_intro) \tag{3.22}$$

Para que la traza $q \lhd q' \lhd \lambda$ cumpla con la propiedad, necesitamos que

- q' sea un sucesor de q al seguir las estrategias en F, asumiendo que qs es la historia de ejecución del sistema, y
- $q' \lhd \lambda$ sea una traza de ejecución al seguir las estrategias en F, siendo $qs \frown q$ la nueva historia de ejecución.

Observemos que es necesario "arrastrar" la secuencia de estados por la que ha pasado el sistema, dado que las estrategias toman sus decisiones en función de la historia completa de ejecución.

A partir de la relación *isOut*, resulta inmediato definir la relación de pertenencia a $out(q, F_A)$:

$$\begin{aligned} isOutFrom(A : Coalition)(F_A : StrategySet\ A)(q : State)(\lambda : Trace) := \\ (hd\ \lambda = q) \wedge isOut\ A\ F_A\ \langle\rangle\ \lambda \end{aligned} \tag{3.23}$$

Tanto para *isOut* como para *isOutFrom*, omitiremos el argumento correspondiente a la coalición, dado que es inferible a partir del conjunto de estrategias en consideración. Utilizaremos la notación $\lambda \in out(q, F_A)$ para referirnos a una prueba de $isOutFrom\ F_A\ q\ \lambda$.

3.3. Formalizando ATL

Nos proponemos en esta sección formalizar la sintaxis y semántica del conjunto de fórmulas de la lógica ATL presentado en la sección 2.2. Asumiremos que S es una estructura de juegos concurrentes tal como se formalizó en la sección 3.2.

Cualquier fórmula de ATL es una fórmula de estados, es decir, su validez depende del estado en el cual se evalúe. La siguiente definición captura este concepto:

$$StateForm := State \to Prop \tag{3.24}$$

Si $q : State$ y $\varphi : StateForm$, una prueba de $(\varphi\ q)$ se interpreta como $S, q \models \varphi$, donde $\models$ es la relación introducida en la sección 2.3.

3.3.1. Constantes y conectivos lógicos

Definimos la fórmula $\top$, que será válida en todos los estados

$$\top : StateForm := fun\ q : State \Rightarrow True \tag{3.25}$$

La fórmula $\bot$ no es válida en ningún estado

$$\bot : StateForm := fun\ q : State \Rightarrow False \tag{3.26}$$

La negación, implicación, conjunción y disyunción se definen empleando los conectivos lógicos presentados en la sección 3.1,

$$\begin{aligned} Neg(\varphi : StateForm) : StateForm &:= fun\ q \Rightarrow \sim \varphi\ q \\ Imp(\varphi, \psi : StateForm) : StateForm &:= fun\ q \Rightarrow \varphi\ q \to \psi\ q \\ And(\varphi, \psi : StateForm) : StateForm &:= fun\ q \Rightarrow \varphi\ q \wedge \psi\ q \\ Or(\varphi, \psi : StateForm) : StateForm &:= fun\ q \Rightarrow \varphi\ q \vee \psi\ q \end{aligned} \tag{3.27}$$

Sobrecargaremos los operadores $\to$, $\wedge$, $\vee$ y $\neg$, y escribiremos $\varphi \to \psi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\neg\varphi$, para denotar las fórmulas $(Imp\ \varphi\ \psi)$, $(And\ \varphi\ \psi)$, $(Or\ \varphi\ \psi)$ y $(Neg\ \varphi)$ respectivamente, para cualesquiera $\varphi, \psi : StateForm$.

3.3.2. El operador Next

En la semántica de ATL presentada en la sección 2.3.1, la fórmula $\langle\langle A\rangle\rangle\bigcirc\varphi$ es válida en un estado q de una estructura de juegos concurrente S si existe un conjunto de estrategias $F_A = \{f_a \mid a \in A\}$, tal que si $\lambda \in out(F_A, q)$, se cumple $S, \lambda[1] \models \varphi$.

Observemos que esta definición es equivalente a la siguiente, en la cual no es necesario utilizar la noción de trazas:

- $S, q \models \langle\langle A\rangle\rangle\bigcirc\varphi$ si existe un conjunto de estrategias $F_A = \{f_a \mid a \in A\}$, tal que para cualquier estado $q' \in sucStSet(\langle\rangle, q, F_A)$, se cumple $S, q' \models \varphi$

En este trabajo utilizaremos esta última propiedad, la cual consideramos captura mejor la noción del operardor $\bigcirc$.

Definición 3.3.1 *Sea $A \subseteq \Sigma$ una coalición, φ : StateForm y q el estado actual del sistema. Definimos la relación inductiva*

$$Next : Coalition \to StateForm \to State \to Prop,$$

mediante el constructor

$$\frac{F : StrategySet\ A \qquad \forall q' : State, q' \in sucStSet(\langle\rangle, F, q) \to \varphi\ q'}{Next\ A\ \varphi\ q}\ (\text{next})$$

El constructor *next* nos permite probar $q \models \langle\langle A\rangle\rangle\bigcirc\varphi$ si puede encontrarse un conjunto de estrategias F para los jugadores en A tal que, para cualquier estado q' que sea un posible sucesor de q, si los jugadores en A siguen las estrategias en F, se cumple $q' \models \varphi$.

Observemos que, si A : *Coalition* y φ : *StateForm*, *Next* $A\ \varphi$: *StateForm*. Utilizaremos la notación $\langle\langle A\rangle\rangle\bigcirc\varphi$ para referirnos a *Next* $A\ \varphi$ cuando resulte conveniente.

3.3.3. El operador Always

Recordemos que la semántica de ATL [2] define la validez de la fórmula $\langle\langle A\rangle\rangle\Box\varphi$ en un estado q de la siguiente manera:

- $q \models \langle\langle A\rangle\rangle\Box\varphi$ sii existe un conjunto $F_A = \{f_a \mid a \in A\}$ de estrategias, tal que para toda traza $\lambda \in out(q, F_A)$ y todas las posiciones $i \geq 0$ se cumple $S, \lambda[i] \models \varphi$.

Por otro lado, la axiomatización descripta en la sección 2.4 presenta una interpretación del operador $\langle\langle A\rangle\rangle\Box$ basándose en la propiedad de punto fijo que posee. El axioma ($\mathrm{FP}_\Box$) asegura que dicho operador es un punto fijo de la ecuación

$$X \leftrightarrow \varphi \wedge \langle\langle A\rangle\rangle \bigcirc X.$$

Adicionalmente, el axioma $\mathrm{GFP}_\Box$ establece que $\langle\langle\rangle\rangle\Box$ es el *mayor* punto fijo de la ecuación. A partir de esta última propiedad, observamos que la semántica del operador puede basarse en una definición coinductiva.

Definición 3.3.2 *Sea $A \subseteq \Sigma$, φ : StateForm, y q : State. La relación coinductiva*

$$Box : Coalition \to StateForm \to State \to Prop$$

se introduce mediante el constructor

$$\frac{F : StrategySet\ A \qquad \varphi\ q \qquad \forall q' : State, q' \in sucStSet(\langle\rangle, F, q) \to Box\ A\ \varphi\ q'}{Box\ A\ \varphi\ q}\ (\text{box})$$

Utilizaremos la notación $\langle\langle A\rangle\rangle\Box\varphi$ para la fórmula de estados ($Box\ A\ \varphi$), y $q \models \langle\langle A\rangle\rangle\Box\varphi$ cuando querramos referirnos a una prueba de $Box\ A\ \varphi\ q$

El constructor *box* construye una prueba de $q \models \langle\langle A\rangle\rangle\Box\varphi$ si

- $\varphi\ q$, y
- Podemos encontrar un conjunto de estrategias F_A tal que, para cualquier posible sucesor q' de F_A, se cumple $q' \models \langle\langle A\rangle\rangle\Box\varphi$.

Observemos que, aplicando nuevamente el costructor *box*, esta última condición requiere demostrar ($\varphi\ q'$) y obtener un nuevo conjunto de estrategias F'_A, tal que, para cualquier F'_A-sucesor q'' de q', se cumple $q'' \models \langle\langle A\rangle\rangle\Box\varphi$. De esta manera, el proceso de demostración es infinito, y es por esto que se introduce mediante un tipo coinductivo.

3.3.4. El operador Until

Para la formalización de este operador, procederemos en forma similar al operador $\langle\langle A\rangle\rangle\Box$, basándonos en la definición por punto fijo introducida en el axioma ($\mathrm{FP}_\mathcal{U}$)

presentado en la sección 2.4:

$$\langle\langle A\rangle\rangle \varphi \,\mathcal{U}\, \psi \leftrightarrow \psi \vee (\varphi \wedge \langle\langle A\rangle\rangle \bigcirc \langle\langle A\rangle\rangle \varphi \,\mathcal{U}\, \psi)$$

Dado que el axioma ($\mathrm{LFP}_{\mathcal{U}}$) esablece que $\langle\langle\rangle\rangle\,\mathcal{U}$ es el *menor* punto fijo de la ecuación, lo formalizaremos empleando un tipo inductivo.

La siguiente definición introduce en nuestra formalización la noción de validez de la fórmula de estados $\langle\langle A\rangle\rangle \varphi \,\mathcal{U}\, \psi$ basándose en las observaciones recién presentadas.

Definición 3.3.3 *Sea $A \subseteq \Sigma$, $\varphi, \psi : StateForm$, y $q : State$. La relación inductiva*

$$Until : Coalition \to StateForm \to StateForm \to State \to Prop$$

se introduce mediante los siguientes constructores:

$$\frac{\psi\ q}{Until\ A\ \varphi\ \psi\ q}\ (\text{until\_now})$$

$$\frac{\begin{array}{c} F : StrategySet\ A \qquad \varphi\ q \\ \forall q' : State, q' \in sucStSet(\langle\rangle, F, q) \to Until\ A\ \varphi\ \psi\ q' \end{array}}{Until\ A\ \varphi\ \psi\ q}\ (\text{until\_later})$$

Utilizaremos la notación $\langle\langle A\rangle\rangle \varphi \,\mathcal{U}\, \psi$ para la fórmula de estados ($Until\ A\ \varphi\ \psi$), y $q \models \langle\langle A\rangle\rangle \varphi \,\mathcal{U}\, \psi$ cuando querramos referirnos a una prueba de $Until\ A\ \varphi\ \psi\ q$

La demostración de la validez de una fórmula del tipo $q \models \langle\langle A\rangle\rangle \varphi \,\mathcal{U}\, \psi$ se logra mediante alguna de las siguientes situaciones

- Encontrando una prueba de $q \models \psi$ (aplicando el constructor *until_now*), o bien
- construyendo un conjunto de estrategias F_A tal que, si los jugadores en A siguen los movimientos devueltos por F_A, para cualquier estado q' que sea F_A-sucesor de q, se cumple $q' \models \langle\langle A\rangle\rangle \varphi \,\mathcal{U}\, \psi$ (aplicando el constructor *until_later*).

Es decir, controlando los jugadores en A, podemos *guiar* al sistema a través de una secuencia de estados $\langle q_0, q_1, \ldots, q_n\rangle$ tal que

- $\forall i \in \{0, \ldots, n-1\}, q_i \models \varphi$,
- $q_n \models \psi$

3.3.5. Operadores Derivados

A partir de los operadores *Next*, *Always* y *Until*, podemos definir operadores derivados utilizados habitualmente en la especificación de sistemas reactivos. Algunos de ellos son:

Definición 3.3.4 (*Eventualmente*) *Sea q el estado actual del sistema. Si los jugadores en una coalición A pueden forzar un estado futuro en el cual valga una cierta fórmula de estados φ, diremos que la fórmula $\langle\langle A\rangle\rangle\Diamond\varphi$ es válida en q. Formalmente:*

$$q \models \langle\langle A\rangle\rangle\Diamond\varphi \leftrightarrow q \models \langle\langle A\rangle\rangle\top\,\mathcal{U}\,\varphi$$

Definición 3.3.5 (*Eventualmente siempre*) *Sea q el estado actual del sistema. Si los jugadores en A pueden forzar una traza de ejecución en la cual eventualmente "siempre φ" sea válida , decimos que A puede asegurar* en todos los posibles estados futuros salvo un número finito *la validez de φ, y lo notaremos $\langle\langle A\rangle\rangle\overset{\infty}{G}\varphi$.*

$$q \models \langle\langle A\rangle\rangle\overset{\infty}{G}\varphi \leftrightarrow q \models \langle\langle A\rangle\rangle\Diamond(\langle\langle\varnothing\rangle\rangle\Box\varphi)$$

4

Propiedades

La formalización presentada en el capítulo 4 permite razonar sobre la lógica ATL y su modelo semántico, CGS. En este capítulo presentamos algunas propiedades sobre ambos modelos.

En la sección 4.1 trabajamos sobre la formalización de estructuras de juegos concurrentes, demostrando teoremas relativos a la relación de transición, la construcción de estrategias, y las posibles trazas de ejecución del sistema que, al seguir determinadas estrategias, una coalición puede forzar. Las propiedades sobre CGS serán de particular interés en al menos dos aspectos. Primero, al formalizar la semántica de los operadores temporales respecto a una estructura de juegos concurrente S, cuando nos enfrentemos a demostraciones de fórmulas ATL será necesario en ocasiones razonar sobre las características de S. Por otro lado, al trabajar con estructuras de juegos concurrentes concretas, como es el caso del capítulo 5, las propiedades ya demostradas a nivel general podrán ser instanciadas y reutilizadas, facilitando la demostración de propiedades específicas.

En la sección 4.2 demostramos la validez de algunas fórmulas ATL, que resultan de especial interés desde el punto de vista teórico y práctico. En el primer caso, varias de las fórmulas demostradas se corresponden con los axiomas y teoremas de la axiomatización de ATL presentada en [17], lo cual pone en evidencia que nuestra formalización puede resultar adecuada para razonar sobre propiedades generales de ATL. Desde el punto de vista de la verificación de sistemas concretos, algunos de los teoremas aquí presentados han sido utilizados para simplificar las fórmulas a demostrar en el caso de estudio presentado en el capítulo 5.

Todas los resultados presentados en este capítulo han sido enunciados y demostrados en *Coq*. Por razones de claridad, aquí en ocasiones se omiten pasos, y solo se ofrece una justificación informal de su validez. Las demostraciones completas (y los términos que las representan) pueden encontrarse en la formalización desarrollada en Coq.

4.1. Razonando en CGS

4.1.1. Propiedades sobre la relación de transición

La hipótesis *trans_d* (3.4) nos permite probar importantes propiedades sobre la relación *"es sucesor de"* (3.5), el conjunto de movimientos permitidos en un estado, *moveAv* (3.6) y el conjunto de posibles trazas de ejecución desde un estado, *isTraceFrom* (3.9).

La primera de estas propiedades nos asegura que en ningún estado el sistema entrará en *deadlock*.

Lema 4.1.1 *Todos los estados tienen al menos un sucesor*

$$always\_suc : \forall\, q : State, \{q' : State \mid suc\ q\ q'\}$$

Demostración *Aplicando trans_d al estado q obtenemos un par (mv, q') tales que se cumple la relación*

$$trans\ q\ mv\ q'$$

El estado q' cumple con las premisas del constructor suc_intro, por lo tanto podemos usarlo como testigo en nuestra demostración. □

Una consecuencia de este resultado es la siguiente propiedad sobre la relación *moveAv*.

Lema 4.1.2 *En cada estado, cada jugador tiene al menos un movimiento permitido*

$$always\_choice : \forall (a : Player)(q : State), \{m : Move \mid moveAv\ a\ q\ m\}$$

Demostración *Aplicando la hipótesis trans_d al estado q, obtenemos un vector de movimientos mv y un estado q' tales que*

$$trans\ q\ mv\ q' \tag{H}$$

Si a : Player, entonces resulta inmediato que (mv a) es un movimiento que cumple con la relación (moveAv q a (mv a)). □

El lema 4.1.1 puede generalizarse para construir una traza de ejecución del sistema. El siguiente resultado realiza tal construcción.

Lema 4.1.3 *Partiendo de cualquier estado q, siempre existe una traza que es una posible ejecución del sistema desde q*

$$always\_trace\_from : \forall\, q : State, \{\lambda : Trace \mid isTraceFrom\ q\ \lambda\}$$

Demostración *Observemos que, a partir de la hipótesis trans_d, es posible construir un sucesor para q. Sea q' tal estado. Aplicando la misma hipótesis a q', obtenemos un estado q'' tal que (suc q' q''). Repitiendo este proceso indefinidamente, obtenemos una traza $\lambda = q, q', q'', \ldots$ que cumplirá con la relación (isTraceFrom q λ).*

La definición formal de una traza que sea una posible continuación desde un cierto estado q se obtiene a partir de una función co-recursiva, como sigue

$$\begin{aligned} \Lambda := fun\ q \Rightarrow \quad & match\ (trans\_d\ q)\ with \\ & \quad \mid (mv, q')\ h \Rightarrow q' \lhd (\Lambda\ q') \\ & end \end{aligned}$$

Recordemos que un término del tipo (trans_d q) es un par formado por una tupla $(mv, q') : MoveVector \times State$ y una prueba $h : (trans\ q\ mv\ q')$. La demostración en Coq de este lema se realiza por coinducción, y utiliza las ideas explicadas más arriba. □

4.1.2. Algunas Propiedades sobre Coaliciones

Probaremos que toda coalición está contenida en Σ, mediante el siguiente lema

Lema 4.1.4 *SubAll* : $\forall(A : Coalition), SubCoalition\ A\ \Sigma$

Demostración *Trivial, dado que para cualquier jugador a siempre podremos encontrar una prueba de* $(\Sigma\ a)$*. El término de prueba es el siguiente:*

$$fun\ (A : Coalition)(a : Player)(\_ : A\ a) \Rightarrow AllC\_intro\ a$$

□

A partir del axioma *coal_dec* (3.17), podemos probar el siguiente lema:

Lema 4.1.5 *AllSub* : $\forall(A : Coalition), SubCoalition\ \Sigma\ (A \uplus \Sigma \backslash A)$

Demostración *A partir de (3.17), el término de prueba resulta*

$$fun\ (A : Coalition)(a : Player)(\_ : \Sigma\ a) \Rightarrow coal\_dec\ A\ a$$

□

El siguiente teorema es consecuencia de los resultados recién expuestos

Teorema 4.1.6 *La unión de una coalición con su complemento resulta en la coalición universal,*

$$eq\_All\_A\_OpA : \forall(A : Coalition), \Sigma = (A \uplus \Sigma \backslash A)$$

Demostración *Trivial, a partir de la definición de la relación same_coalition* (3.16) *y los lemas 4.1.4 y 4.1.5* □

4.1.3. Un conjunto de estrategias para cada coalición

Consideremos el axioma $(\top)$ declarado en la sección 2.4. Si queremos demostrar que es válido en cualquier estructura de juegos concurrentes, debemos encontrar un conjunto de estrategias F tal que, para todos los estados q' que son posibles sucesores del estado actual si los jugadores en la coalición siguen las estrategias en F, se cumple $q' \models \top$;

esta última proposición es trivial, y se cumple para cualquier estado. Sin embargo, aun es necesario construir el conjunto F.

La siguiente definición construye un conjunto de estrategias para una coalición haciendo uso del lema 4.1.2, que nos asegura la existencia de un movimiento permitido para cada jugador en cada estado:

Definición 4.1.7 *El conjunto de estrategias* trivial *para una coalición se define como*

$$\begin{aligned} &\mathit{trivialStSet}(A : \mathit{Coalition}) := \\ &\quad \mathit{fun}\ (a : \mathit{Player})(H : A\ a)(qs : \mathrm{seq}\ \mathit{State})(q : \mathit{State}) \Rightarrow \\ &\quad\quad (\mathit{match}\ \mathit{always\_choice}\ a\ q\ \mathit{with}\ \mathit{exist}\ m\ \_ \Rightarrow m) \end{aligned}$$

El conjunto de estrategias *trivialStSet* A elige algún movimiento permitido para cada jugador en cada estado. Puede pensarse como un conjunto de estrategias que aleatoriamente elige movimientos para cada jugador. Ninguna propiedad particular puede asumirse sobre los movimientos que elige, a excepción de que son permitidos en el estado actual.

4.1.4. La unión hace la fuerza

La axiomatización de ATL presentada en [17] incluye formas de razonar sobre la unión de coaliciones. En la definición (3.14) introdujimos la noción de unión de dos coaliciones A y B. Definiremos ahora un mecanismo para construir un conjunto de estrategias F para $A \uplus B$, a partir de conjuntos de estrategias F_A y F_B para A y B respectivamente. Recordemos que un conjunto de estrategias para $A \uplus B$ es una función cuyo tipo es $\forall(a : \mathit{Player}), A \uplus B\ a \rightarrow \mathit{Strategy}$

Definición 4.1.8 *Un (posible) conjunto de estrategias para la coalición $A \uplus B$, a partir de los conjuntos de estrategias $F_A : \mathit{StrategySet}\, A$ y $F_B : \mathit{StrategySet}\, B$ se construye mediante la siguiente definición.*

$$\begin{aligned} &\mathit{unionStSet}\ (A\ B\ F_A\ F_B) : \mathit{StrategySet}(A \uplus B) := \\ &\quad \mathit{fun}\ (a : \mathit{Player})(H : A \uplus B) \Rightarrow \\ &\quad\quad \mathit{match}\ H\ \mathit{with} \\ &\quad\quad\quad \mid \mathit{inl}\ H_A \Rightarrow F_A\ a\ H_A \\ &\quad\quad\quad \mid \mathit{inr}\ H_B \Rightarrow F_B\ a\ H_B \\ &\quad\quad \mathit{end} \end{aligned}$$

Al igual que para coaliciones, emplearemos la notación $F_A \uplus F_B$ para denotar la estrategia *unionStSet* F_A F_B.

Demostraremos ahora algunas propiedades que la unión de dos conjuntos de estrategias F_A, F_B cumplen respecto a una estructura de juegos concurrente. Estas propiedades serán de gran utilidad a la hora de demostrar la validez de fórmulas de ATL en la sección 4.2.

Lema 4.1.9 *Sean* A, B : *Coalition, y* F_A : *StrategySet* A, F_B : *StrategySet* B. *El conjunto de estados posibles estados sucesores al seguir la estrategia* $F = F_A \uplus F_B$ *para* $A \uplus B$ *está contenido en el conjunto de posibles sucesores para* F_A*:*

$$\mathit{UnionLeftSuc} : q' \in \mathit{sucSt}(qs, q, F_A \uplus F_B) \rightarrow q' \in \mathit{sucSt}(qs, q, F_A)$$

Demostración *Sean* qs : seq *State*, q, q' : *State, y* H *una prueba de*

$$q' \in \mathit{sucSt}(qs, q, F_A \uplus F_B) \tag{H}$$

Debemos demostrar que $q' \in \mathit{sucStSet}(qs, q, F_A)$. *Para esto, es necesario construir un vector de movimientos* m *tal que*

(1) $\forall(a : \mathit{Player})(H_A : A), F_A\ a\ qs\ q = m\ a$, *y*

(2) $\mathit{trans}\ q\ m\ q'$

Por la definición de sucStSet, a partir de (H) *podemos asegurar la existencia de un vector de movimientos* mv *tal que:*

$$\forall(a : \mathit{Player})(H : A \uplus B\ a), F_A \uplus F_B\ a\ H\ qs\ q = mv\ a \tag{Hmv}$$

$$\mathit{trans}\ q\ mv\ q' \tag{Htrans}$$

Proponemos mv *como argumento del constructor sucStSet_intro para la demostración de (1) y (2). La prueba de (2) es trivial a partir de* (Htrans). *Para demostrar (1), sean* a : *Player y* H_A : A a, *nuestro objetivo es*

$$F_A\ a\ H_A\ qs\ q\ = mv\ a$$

La hipótesis (Hmv) *es una función que, dados* a : *Player y* $H : A \uplus B$, *nos construye una prueba de la igualdad*

$$F_A \uplus F_B\ a\ H\ qs\ q = mv\ a$$

Por lo tanto, reescribiendo en nuestro objetivo (Hmv a (inl H_A)), obtenemos la igualdad

$$F_A\ a\ H_A\ qs\ q = F_A \uplus F_B\ a\ (inl\ H_A)\ qs\ q$$

A partir de la definición de unionStSet (4.1.8), el término derecho se reduce al izquierdo, con lo cual la igualdad resulta trivial. □

Un resultado similar puede demostrarse para la coalición B:

Lema 4.1.10 *El conjunto de estados posibles estados sucesores al seguir la estrategia $F = F_A \uplus F_B$ para $A \uplus B$ está contenido en el conjunto de posibles sucesores para F_B:*

$$UnionRightSuc : q' \in sucSt(qs, q, F_A \uplus F_B) \rightarrow q' \in sucSt(qs, q, F_B)$$

Demostración *Análoga a la del lema 4.1.9* □

Los lemas recién demostrados resultan de particular importancia para verificar que el axioma (S) es lógicamente válido. A partir de estos lemas, y utilizando la técnica de coinducción, podemos demostrar los siguientes resultados, que muestran que las propiedades anteriores son válidas a lo largo de cualquier traza de ejecución del sistema:

Lema 4.1.11 *Sea qs la historia de ejecución de una estructura de juegos concurrentes S, y q el estado actual. Si λ es una posible continuación del sistema para la coalición $A \uplus B$, al seguir el conjunto de estrategias $F_A \uplus F_B$, entonces λ es una posible continuación del sistema para la coalición A, al seguir el conjunto de estrategias F_A,*

$$UnionLeftTrace : isOut\ F_A \uplus F_B\ qs\ \lambda \rightarrow isOut\ F_A\ qs\ \lambda$$

Demostración *Sean A, B : Coalition, y F_A, F_B conjuntos de estrategias para A y B respectivamente. Dado que el predicado isOut es coinductivo, la demostración será por coinducción. Por lo tanto, asumimos la hipótesis*

$$\forall(\lambda,\ qs), isOut\ F_A \uplus F_B\ qs\ \lambda \rightarrow isOut\ F_A\ qs\ \lambda \qquad \text{(Hcoind)}$$

que emplearemos únicamente dentro de un constructor del tipo isOut ([13, 16]).

Sean λ : *Trace*, qs : seq *State*, *y H una prueba de*

$$isOut\ F_A \uplus F_B\ qs\ \lambda \tag{H}$$

Expresando a λ *como* $q \triangleleft q' \triangleleft \lambda'$, *a partir de* (H) *y aplicando la definición coinductiva de isOut, podemos asegurar los siguientes resultados:*

$$sucStSet\ (F_A \uplus F_B)\ qs\ q\ q' \tag{Hsuc}$$

$$isOut\ (F_A \uplus F_B)\ (qs \frown q)\ (q' \triangleleft \lambda') \tag{Hout}$$

Nuestro objetivo se reescribe como:

$$isOut\ F_A\ qs\ (q' \triangleleft \lambda')$$

Aplicando el constructor de isOut_intro, debemos probar

(1) $sucStSet\ F_A\ qs\ q\ q'$

(2) $isOut\ F_A\ (qs \frown q)\ (q' \triangleleft \lambda')$

La prueba de (1) se obtiene aplicando el lema UnionLeftSuc a la hipótesis Hsuc. La prueba de (2) se obtiene aplicando la hipótesis (Hcoind) *con los argumentos* $(q' \triangleleft \lambda')$, $(qs \frown q)$ *y* (H). □

Lema 4.1.12 *Sea qs la historia de ejecución de una estructura de juegos concurrentes S, y q el estado actual. Si* λ *es una posible continuación del sistema para la coalición* $A \uplus B$, *al seguir el conjunto de estrategias* $F_A \uplus F_B$, *entonces* λ *es una posible continuación del sistema para la coalición A, al seguir el conjunto de estrategias* F_A,

$$UnionRightTrace : isOut\ F_A \uplus F_B\ qs\ x \rightarrow isOut\ F_B\ qs\ x$$

Demostración *Análoga a la del lema 4.1.11* □

De estos resultados se derivan los siguientes lemas, que relacionan los conjuntos $out(q, F_A)$ y $out(q, F_B)$ con el conjunto $out(q, F_A \uplus F_B)$

Teorema 4.1.13 *El conjunto de posibles trazas de ejecución del sistema a partir del estado inicial q para $A \uplus B$, al seguir el conjunto de estrategias $F_A \uplus F_B$, está contenido en $out(q, F_A)$*

$$UnionRightFrom : \lambda \in out(q, F_A \uplus F_B) \rightarrow \lambda \in out(q, F_A)$$

Demostración *Trivial, a partir del lema 4.1.11* □

Teorema 4.1.14 *El conjunto de posibles trazas de ejecución del sistema a partir del estado inicial q para $A \uplus B$, al seguir el conjunto de estrategias $F_A \uplus F_B$, está contenido en $out(q, F_B)$*

$$UnionRightFrom : \lambda \in out(q, F_A \uplus F_B) \rightarrow \lambda \in out(q, F_B)$$

Demostración *Trivial, a partir del lema 4.1.12* □

4.2. Razonando en ATL

Recordemos que una fórmula de ATL φ es *válida* en una estructura de juegos concurrentes S si para cualquier estado q de S se cumple $q \models \varphi$. En lo que sigue, asumiremos que S es una estructura de juegos concurrente tal como se formalizó en la sección 3.2.

4.2.1. Propiedades del Operador Next

Teorema 4.2.1 *Sean $A_1, A_2, \subseteq \Sigma$ dos coaliciones disjuntas, y $\varphi, \psi : StateForm$. Entonces*

$$\langle\langle A_1 \rangle\rangle \bigcirc \varphi \wedge \langle\langle A_2 \rangle\rangle \bigcirc \psi \rightarrow \langle\langle A_1 \uplus A_2 \rangle\rangle \bigcirc (\varphi \wedge \psi)$$

Demostración *Sea $q : State$. Nuestro objetivo es construir un conjunto de estrategias F para $A_1 \uplus A_2$ tal que, si $q' \in sucStSet(\langle\rangle, q, F)$, entonces se cumple $(\varphi \wedge \psi)\ q'$.*

A partir de la primer premisa ($\langle\langle A_1 \rangle\rangle \bigcirc \varphi$), podemos concluir que existe un conjunto de estrategias F_{A_1} tal que,

$$\forall q' : State, q' \in sucStSet(\langle\rangle, q, F_{A_1}) \rightarrow \varphi\ q' \qquad (\mathrm{H}\varphi)$$

Del mismo modo, podemos asegurar la existencia de un conjunto de estrategias F_{A_2} *tal que*

$$\forall q' : State, q' \in sucStSet(\langle\rangle, q, F_{A_2}) \rightarrow \psi\ q' \qquad (\mathrm{H}\psi)$$

Proponemos entonces como conjunto de estrategias $F = F_{A_1} \uplus F_{A_2}$*, definido en la sección 4.1.4. Debemos demostrar que*

$$\forall q' : State, q' \in sucStSet(\langle\rangle, q, F) \rightarrow (\varphi\ q' \wedge \psi\ q')$$

Consideremos un estado $q' \in sucStSet(\langle\rangle, q, F)$*. Es decir,* q' *es un posible sucesor de* q *si los jugadores en* $A_1 \uplus A_2$ *siguen el conjunto de estrategias* F*.*

A partir del lema UnionLeftSuc (4.1.9) concluimos que $q' \in sucStSet(\langle\rangle, q, F_{A_1})$*. Usando este hecho y* $(\mathrm{H}\varphi)$ *tenemos una prueba de* $(\varphi\ q')$*.*

Con un razonamiento análogo, y empleando el lema UnionRightSuc (4.1.10) y la hipótesis $(\mathrm{H}\psi)$ *concluimos* $(\psi\ q')$*.* □

Teorema 4.2.2 (Monotonía de $\langle\langle\rangle\rangle\bigcirc$ sobre $\subseteq$ entre coaliciones)
Sean $A_1, A_2 : Coalition$*, y* $\varphi : StateForm$*. Entonces*

$$\langle\langle A_1\rangle\rangle\bigcirc\varphi \rightarrow \langle\langle A_1 \uplus A_2\rangle\rangle\bigcirc\varphi$$

Demostración *Sea* $q : State$*, probaremos que a partir de*

$$q \models \langle\langle A_1\rangle\rangle\bigcirc\varphi \qquad (\mathrm{H})$$

podemos deducir $q \models \langle\langle A_1 \uplus A_2\rangle\rangle\bigcirc\varphi$ *.*

Observemos que (H) *nos asegura la existencia de un conjunto de estrategias* $F_{A_1} : StrategySet\ A_1$ *tal que*

$$\forall q' : State, q' \in sucStSet(\langle\rangle, q, F_{A_1}) \rightarrow \varphi\ q' \qquad (H')$$

Debemos encontrar un conjunto de estrategias $F : StrategySet(A_1 \uplus A_2)$ *tal que,*

$$\forall q' : State, q' \in sucStSet(\langle\rangle, q, F) \rightarrow (\varphi\ q')$$

.

Con la intención de emplear la hipótesis (H')*, buscaremos un conjunto de estrategias tal que* $sucStSet(\langle\rangle, q, F) \subseteq sucStSet(\langle\rangle, q, F_{A_1})$*. Como hemos visto en el lema UnionLeftSuc (4.1.9), si* $F = F_{A_1} \uplus F_{A_2}$*, donde* $F_{A_2} : StrategySet\ A_2$*, entonces la propiedad anterior es válida.*

Por lo tanto, sólo debemos encontrar un conjunto de estrategias cualquiera para A_2*. Para esto, utilizaremos el conjunto de estrategias "trivial" definido en la sección 4.1.3, es decir* $F_{A_2} := trivialStSet\ A_2$

De esta manera, aplicando el constructor next con $F := F_{A_1} \uplus F_{A_2}$*, nuestro objetivo resulta*

$$\forall q' : State, q' \in sucStSet(\langle\rangle, q, F_{A_1} \uplus F_{A_2}) \to \varphi\ q'$$

A partir de (H') *y el lema UnionLeftSuc (4.1.9), la demostración queda completa.* □

Otras propiedades demostradas

Los siguientes teoremas enuncian propiedades relativas al operador *Next* que han sido demostradas en nuestra formalización:

Teorema 4.2.3 (⊥) *Sea* $A : Coalition$*, entonces la siguiente fórmula es válida en cualquier estructura de juegos concurrente*

$$\langle\langle A\rangle\rangle \bigcirc \bot \to \bot$$

□

Teorema 4.2.4 (○-Regularidad) *Sean* $A : Coalition$*,* $\varphi : StateForm$*, entonces la siguiente fórmula es válida en cualquier estructura de juegos concurrentes*

$$\langle\langle A\rangle\rangle \bigcirc \varphi \wedge \langle\langle Oponent\ A\rangle\rangle \bigcirc \neg\varphi \to \langle\langle \Sigma\rangle\rangle \bigcirc \bot$$

□

Teorema 4.2.5 (○-Monotonía) *Sean* $A : Coalition$*,* $\varphi, \psi : StateForm$*, entonces*

$$(\forall s : State, \varphi\ s \to \psi\ s) \to \langle\langle A\rangle\rangle \bigcirc \varphi \to \langle\langle A\rangle\rangle \bigcirc \psi$$

es una fórmula válida de ATL. □

Las demostraciones resultan similares a las presentadas en lo teoremas 4.2.1 y 4.2.2, y el lector interesado puede consultar la formalización *Coq* para más detalles.

4.2.2. Propiedades del Operador Always

Teorema 4.2.6 (□-Inducción) *Sean* $A \subseteq \Sigma$, $\varphi, \psi : StateForm$, *entonces*

$$(\forall s : State, \psi\, s \to (\varphi\, s \wedge s \models \langle\langle A \rangle\rangle \bigcirc \psi\,)) \to (\forall q : State, \psi\, q \to q \models \langle\langle A \rangle\rangle \Box \varphi\,)$$

Demostración *Asumamos*

$$(\forall s : State, \psi\, s \to (\varphi\, s \wedge s \models \langle\langle A \rangle\rangle \bigcirc \psi\,)) \tag{H}$$

Es decir, para cualquier estado s, *una prueba de* $\psi\, s$ *nos permite asegurar* $\varphi\, s$ *y que los jugadores en* A *tienen un conjunto de estrategias* F *que aseguran que en cualquier estado* $s' \in sucStSet(\langle\rangle, s, F)$ *se cumple* $\psi\, s'$.

La prueba de

$$\forall q : State, \psi\, q \to q \models \langle\langle A \rangle\rangle \Box \varphi$$

será por coinducción, construyendo, para un estado q *que cumpla* $\psi\, q$, *una prueba de* $q \models \langle\langle A \rangle\rangle \Box \varphi$.

Por lo tanto, podemos asumir como hipótesis

$$\forall q : State, \psi\, q \to q \models \langle\langle A \rangle\rangle \Box \varphi \tag{Hco-ind}$$

siempre y cuando sea utilizada dentro de un constructor del tipo coinductivo Box ([13, 16]).

Sea entonces $q : State$, *y asumamos*

$$\psi\ q \tag{H}\psi)$$

Debemos entonces probar $q \models \langle\langle A \rangle\rangle \Box \varphi$. *Para esto, es necesario ver que* $(\varphi\ q)$ *es válida, y que existe un conjunto de estrategias* $F : StrategySet\, A$ *tal que* $\forall q' : State, q' \in sucStSet(\langle\rangle, q, F) \to q' \models \langle\langle A \rangle\rangle \Box \varphi$.

Si aplicamos (H) *al estado* q *y la hipótesis* (Hψ) *obtenemos*

$$\varphi\ q \wedge q \models \langle\langle A\rangle\rangle \bigcirc \psi \tag{H'}$$

Por lo tanto, φ *es válida en* q*. y existe un conjunto de estrategias* $F_A : StrategySet\ A$ *tal que*

$$\forall q' : State, q' \in sucStSet(\langle\rangle, q, F_A) \rightarrow \psi\ q' \tag{HSuc}$$

Proponemos entonces a este conjunto F_A *como el conjunto que nos asegurará la invarianza de* φ *a lo largo de cualquier traza* $\lambda \in out(q, F)$*.*

Apliquemos entonces el constructor box con F_A *como argumento.*

Necesitamos una prueba de

(1) $\varphi\ q$

(2) $\forall q' : State, q' \in sucStSet(\langle\rangle, q, F_A) \rightarrow q' \models \langle\langle A\rangle\rangle \Box \varphi$

La prueba de (1) es directa a partir de (H$'$) *Para probar (2), tomamos* q' : *State tal que* $q' \in sucStSet(\langle\rangle, q, F_A)$*. A partir de* (HSuc) *concluimos* $(\psi\ q')$*. Usando este hecho y la hipótesis de coinducción* (Hco-ind)*, obtenemos un término de prueba de* $q' \models \langle\langle A\rangle\rangle \Box \varphi$ *.*

□

Teorema 4.2.7 (□ - Inevitable) *Sean* A : *Coalition y* P : *StateForm. Entonces*

$$(\forall q : State, \varphi\ q) \rightarrow (\forall q : State, q \models \langle\langle A\rangle\rangle \Box \varphi\ )$$

Demostración *Sea* H *la hipótesis que nos asegura la validez de* φ *en cada estado:*

$$\forall q : State, \varphi\ q \tag{H}$$

Demostraremos que para cualquier estado q *se cumple* $q \models \langle\langle A\rangle\rangle \Box \varphi$ *por coinducción, siguiendo la definición del operador* $\langle\langle\rangle\rangle \Box$*. La hipótesis de coinducción resulta*

$$\forall q : State, q \models \langle\langle A\rangle\rangle \Box P \tag{Hco-ind}$$

Observemos que la hipótesis (H) *nos provee suficiente información para demostrar la propiedad, dado que φ será necesariamente válida a lo largo de cualquier traza de ejecución (dado que es un invariante del sistema), por lo tanto cualquier conjunto de estrategias para A será suficiente. Proponemos entonces el conjunto trivialStSet A como conjunto de estrategias para A. Aplicando el constructor box con* $(F := \mathit{trivialStSet}\ A)$ *como argumento, nuestros objetivos resultan*

(1) $\varphi\ q$

(2) $\forall q' : \mathit{State}, q' \in \mathit{sucStSet}(\langle\rangle, q, F) \rightarrow q' \models \langle\langle A \rangle\rangle \Box \varphi$

Ambas pruebas son triviales: La prueba de (1) es directa a partir de (H)*, y la de (2) a partir de* (Hco-ind). □

Otras propiedades demostradas

Los siguientes teoremas han sido demostrados en la formalización *Coq*. Por razones de espacio no presentamos su demostración.

Teorema 4.2.8 (□-Distribución) *Sea* $A : \mathit{Coalition}$*, y* $\varphi, \psi : \mathit{StateForm}$*. Entonces*

$$\langle\langle A \rangle\rangle \Box(\varphi \rightarrow \psi) \wedge \langle\langle A \rangle\rangle \Box \varphi \rightarrow \langle\langle A \rangle\rangle \Box \psi$$

es un teorema de la lógica ATL. □

Teorema 4.2.9 (□-Monotonía) *Sea* $A : \mathit{Coalition}$*, y* $\varphi, \psi : \mathit{StateForm}$*. Entonces*

$$(\forall s : \mathit{State}, \varphi\ s \rightarrow \psi\ s) \rightarrow \langle\langle A \rangle\rangle \Box \varphi \rightarrow \langle\langle A \rangle\rangle \Box \psi$$

es un teorema de la lógica ATL □

4.2.3. Propiedades del Operador Until

Teorema 4.2.10 (Until-inducción) *Sean* $A \subseteq \Sigma$*,* $\varphi, \psi, \phi : \mathit{StateForm}$*, entonces*

$$(\forall s : \mathit{State}, \psi\ s \vee (\varphi\ s \wedge s \models \langle\langle A \rangle\rangle \bigcirc \phi\ ) \rightarrow \phi\ s) \rightarrow \forall q : \mathit{State}, q \models (\langle\langle A \rangle\rangle \varphi\, \mathcal{U}\, \psi\ \rightarrow \phi)$$

Demostración *Asumamos la hipótesis*

$$\forall s : State, \psi\, s \vee (\varphi\, s \wedge s \models \langle\langle A \rangle\rangle \bigcirc \phi\,) \rightarrow \phi\ s \tag{H}$$

la cual asegura que en cualquier estado s para probar la validez de $(\phi\ s)$ es suficiente con demostrar $\psi\ s$, o bien $(\varphi\ s \wedge s \models \langle\langle A \rangle\rangle \bigcirc \phi\,)$.

Debemos probar la proposición

$$\forall q : State, q \models (\langle\langle A \rangle\rangle \varphi\, \mathcal{U}\, \psi\ \rightarrow \phi)$$

Sea $q : StateForm$ y la hipótesis

$$q \models \langle\langle A \rangle\rangle \varphi\, \mathcal{U}\, \psi \tag{H$\mathcal{U}$}$$

La demostración de $\phi\ q$ será por inducción en (H$\mathcal{U}$)*. El principio de inducción asociado al operador Until nos genera dos obligaciones de prueba, una por cada constructor del operador.*

Caso base: *until_here . En este caso asumimos que* (H$\mathcal{U}$) *se construye a partir del primer constructor del operador Until, por lo que contamos con la siguiente hipótesis:*

$$\psi\ q \tag{H$\mathcal{U}$ *_here*}$$

La prueba de $\phi\ q$ es directa aplicando H al estado q, y proveyendo una demostración de $\psi\ q \vee (\varphi\ q \wedge q \models \langle\langle A \rangle\rangle \bigcirc \phi\,)$ a partir de (H$\mathcal{U}$ *_here*)

Paso Inductivo: *until_there . En este caso, el principio de inducción para $\mathcal{U}$ nos asegura la existencia de un conjunto de estrategias F_A tal que*

$$\varphi\ q \tag{Hφ}$$

$$\forall q' : State, q' \in sucStSet(\langle\rangle, q, F_A) \rightarrow q' \models \langle\langle A \rangle\rangle \varphi\, \mathcal{U}\, \psi \tag{Hind}$$

$$\forall q' : State, q' \in sucStSet(\langle\rangle, q, F_A) \rightarrow q' \models \phi \tag{H$\bigcirc\mathcal{U}$}$$

La demostración de $\phi\ q$ se obtiene aplicando la hipótesis (H) *al estado q. Ésta nos generará la siguiente obligación de prueba:*

$$\psi\ q \vee (\varphi\ q \wedge q \models \langle\langle A \rangle\rangle \bigcirc \phi\,)$$

Demostraremos esta disyunción proveyendo una prueba de su lado derecho. La demostración de φ q es trivial a partir de (Hφ). *Para demostrar $q \models \langle\langle A\rangle\rangle \bigcirc \phi$, aplicamos el constructor next con F_A como conjunto de estrategias. La hipótesis* (Hind) *provee la información necesaria para concluir la prueba.*

□

Teorema 4.2.11 ($\mathcal{U}$-Monotonía) *Sea A : Coalition, y $\varphi, \varphi', \psi, \psi'$: StateForm. Entonces la siguiente es una fórmula válida en ATL*

$$\big(\forall s : State, \varphi\ s \to \varphi'\ s\big) \wedge \big(\forall s : State, \psi\ s \to \psi'\ s\big) \to \langle\langle A\rangle\rangle \varphi\, \mathcal{U}\, \psi \to \langle\langle A\rangle\rangle \varphi'\, \mathcal{U}\, \psi'$$

Demostración *La demostración es por inducción en la prueba de $\langle\langle A\rangle\rangle \varphi\, \mathcal{U}\, \psi$. Las dos obligaciones de pruebas se demuestran de manera directa a partir de las hipótesis generadas.* □

5

Caso de Estudio

En el capítulo previo nuestra formalización de ATL ha sido utilizada para razonar sobre propiedades generales de dicha lógica. Hemos demostrado una gran cantidad de teoremas que pueden resultar de utilidad al momento de verificar propiedades sobre sistemas concretos. En este capítulo pretendemos analizar la posibilidad de utilizar la formalización de la lógica ATL para la verificación de sistemas reactivos y concurrentes. Para ello, proponemos dos casos de estudio relacionados.

El primero formaliza la estructura de juegos concurrentes presentada en el ejemplo 2.1.3, especificando un protocolo de control de un paso a nivel, cuyos componentes son un controlador y un tren que realiza pedidos para ingresar a la compuerta del paso a nivel. En la sección 5.1 formalizamos la estructura de juegos concurrentes siguiendo las definiciones dadas en la sección 3.2; y especificamos en ATL propiedades que el sistema debe verificar. La sección 5.2 presenta una generalización del sistema anterior, en la cual un número no acotado de trenes compiten por el ingreso a la compuerta, y el controlador debe asegurar diferentes propiedades de seguridad y vitalidad. Algunas de estas propiedades son enunciadas y verificadas para tal sistema.

Observemos que, si bien en el primer caso de estudio puede utilizarse una herramienta de verificación automática, el segundo problema no puede ser verificado por un *model checker*, dado que posee un número infinito de estados.

5.1. Controlando un paso a nivel

En esta sección presentaremos una formalización en teoría de tipos del ejemplo 2.1.3, y demostraremos diferentes propiedades que el sistema debe cumplir.

5.1.1. El modelo

Estados Los estados del sistema serán introducidos mediante un tipo inductivo como sigue:

$$State : Set := q_{out} \mid q_{req} \mid q_{gran} \mid q_{in}$$

Jugadores El tipo que representa a los dos jugadores del sistema es

$$Player : Set := Train \mid Controller$$

Movimientos Los posibles movimientos del sistema son los siguientes

$$\begin{aligned} Move : Set \quad := \quad & \mid stayOut \mid request \mid grant \mid delay \mid deny \mid enter \\ & \mid relinquish \mid keepClosed \mid reopen \mid idle \end{aligned}$$

- *stayOut* y *request* representan los dos movimientos permitidos para el tren en el estado q_{out};
- *grant*, *delay* y *deny* son los permitidos para el controlador en el estado q_{req}
- *enter* y *relinquish* son las dos opciones del tren en el estado q_{gran}
- *reopen* y *keepClosed* son los movimientos permitidos para el controlador en el estado q_{in}
- *idle* representa el único movimiento permitido para los jugadores en los estados en que no es "su turno"

Vectores de Movimientos Un vector de movimientos es una función de tipo $Player \to Move$. Utilizaremos la notación $\langle mt, mc \rangle$, donde $mt, mc : Move$ para representar el vector de movimientos

$$fun\ p : Player \Rightarrow (match\ p\ with Train \Rightarrow mt \mid Controller \Rightarrow mc)$$

Transiciones Las transiciones del sistema también se introducen mediante un tipo inductivo, con un constructor por cada posible transición, como sigue

$$
\begin{array}{l}
trans : State \to MoveVector \to State \to Prop := \\
\quad \begin{array}{llllll}
\mid transOut1 & : & trans & q_{out} & \langle stayOut, idle\rangle & q_{out} \\
\mid transOut2 & : & trans & q_{out} & \langle request, idle\rangle & q_{req} \\
\mid transReq1 & : & trans & q_{req} & \langle idle, grant\rangle & q_{gran} \\
\mid transReq2 & : & trans & q_{req} & \langle idle, delay\rangle & q_{req} \\
\mid transReq3 & : & trans & q_{req} & \langle idle, deny\rangle & q_{out} \\
\mid transGra1 & : & trans & q_{gran} & \langle enter, idle\rangle & q_{in} \\
\mid transGra2 & : & trans & q_{gran} & \langle relinquish, idle\rangle & q_{out} \\
\mid transIn1 & : & trans & q_{in} & \langle idle, keepClosed\rangle & q_{in} \\
\mid transIn2 & : & trans & q_{in} & \langle idle, reopen\rangle & q_{out}
\end{array}
\end{array}
\tag{5.1}
$$

Emplearemos la notación $(q, \langle mt, mc\rangle) \rightsquigarrow q'$ para representar una prueba de $(trans\ q\ \langle mt, mc\rangle\ q')$.

Observemos que hemos definido la función de transición como una relación. Es posible demostrar que la relación *trans* así definida cumple con las hipótesis *trans_f* (3.3) y *trans_d* (3.4). Por lo tanto, todos los lemas que utilizan dichas hipótesis son válidos en nuestro modelo.

Coaliciones Además de las coaliciones Σ y $\varnothing$, definimos una coalición para cada uno de los jugadores del sistema, mediante las siguientes definiciones

$$
\begin{array}{lll}
T : Coalition & := & fun\ p : Player \Rightarrow \\
 & & match\ p\ with \\
 & & \quad Train \Rightarrow \{1\} \\
 & & \quad \mid Controller \Rightarrow \{\ \} \\
 & & end \\
C : Coalition & := & fun\ p : Player \Rightarrow \\
 & & match\ p\ with \\
 & & \quad Train \Rightarrow \{\ \} \\
 & & \quad \mid Controller \Rightarrow \{1\} \\
 & & end
\end{array}
$$

donde los tipos $\{1\}$ y $\{\ \}$ son los definidos en la sección 3.1. Con estas definiciones, siempre podremos encontrar un término del tipo $(T\ \ Train)$, pero no será posible

construir un término en el tipo (T *Controller*). Observaciones análogas pueden hacerse sobre la coalición C.

Fórmulas de estados atómicas Las fórmulas de estados atómicas representan las variables observables del sistema (cada uno de los elementos del conjunto Π). Para el caso del ejemplo 2.1.3, existen cuatro variables observables:

$$\Pi = \{Out\_of\_gate, Requested, Granted, In\_gate\}$$

Representaremos cada una de estas fórmulas como un elemento del tipo *StateForm*:

$$Out\_of\_gate : StateForm :=$$
$$\quad fun\ q : State \Rightarrow match\ q\ with\ q_{in} \Rightarrow False \mid \_ \Rightarrow True\ end$$

$$Requested : StateForm :=$$
$$\quad fun\ q : State \Rightarrow match\ q\ with\ q_{req} \Rightarrow True \mid \_ \Rightarrow False\ end$$

$$Granted : StateForm :=$$
$$\quad fun\ q : State \Rightarrow match\ q\ with\ q_{gran} \Rightarrow True \mid \_ \Rightarrow False\ end$$

$$In\_gate : StateForm :=$$
$$\quad fun\ q : State \Rightarrow match\ q\ with\ q_{in} \Rightarrow True \mid \_ \Rightarrow False\ end$$

5.1.2. Propiedades

La formalización presentada en la sección precedente nos permite demostrar importantes propiedades que el sistema debe cumplir. Algunas de ellas son:

- Siempre que el tren esté fuera de la compuerta, y no se le haya dado permiso para entrar, el controlador puede impedirle ingresar a la compuerta:

$$\langle\langle\varnothing\rangle\rangle\Box\left((Out\_of\_gate \wedge \neg Granted) \rightarrow \langle\langle C\rangle\rangle\Box\, Out\_of\_gate\right) \tag{5.2}$$

- Si el tren se encuentra fuera de la compuerta, el controlador no puede forzarlo a ingresar a ella:

$$\langle\langle\varnothing\rangle\rangle\Box\left(Out\_of\_gate \rightarrow \neg\langle\langle C\rangle\rangle\Diamond In\_gate\right) \tag{5.3}$$

- Si el tren se encuentra fuera de la compuerta, éste y el controlador pueden cooperar de forma tal que el tren ingrese a la compuerta:

$$\langle\langle\varnothing\rangle\rangle\Box\,(Out\_of\_gate \rightarrow \langle\langle\Sigma\rangle\rangle\Diamond In\_gate) \tag{5.4}$$

- Siempre que el tren se encuentre fuera de la compuerta, puede eventualmente solicitar permiso para ingresar, en cuyo caso el controlador decide si otorgar o no el permiso para hacerlo:

$$\begin{aligned}\langle\langle\varnothing\rangle\rangle\Box(&Out\_of\_gate \rightarrow \\ &\langle\langle T\rangle\rangle\Diamond\,(Requested \wedge \langle\langle C\rangle\rangle\Diamond Granted \wedge \langle\langle C\rangle\rangle\Box\neg Granted))\end{aligned} \tag{5.5}$$

- Siempre que el tren se encuentre dentro de la compuerta, el controlador puede forzarlo a salir en el próximo paso:

$$\langle\langle\varnothing\rangle\rangle\Box\,(In\_gate \rightarrow \langle\langle C\rangle\rangle\bigcirc Out\_of\_gate) \tag{5.6}$$

Observación Todas las propiedades presentadas son del tipo $\langle\langle\varnothing\rangle\rangle\Box\varphi$, donde φ : *StateForm*. Sin embargo, cuando demostremos la validez de dichas fórmulas, nos bastará con probar que φ es válida en todos los estados. En efecto, a partir de esta demostración y el lema 4.2.7 puede probarse $\langle\langle\varnothing\rangle\rangle\Box\varphi$

Teorema 5.1.1 *La propiedad 5.4 es válida.*

Demostración *Teniendo en cuenta la observación 5.1.2, es suficiente con demostrar la fórmula*

$$\forall q : State, q \models (Out\_of\_gate \rightarrow \langle\langle\Sigma\rangle\rangle\Diamond In\_gate\,)$$

Sea q : State, y la hipótesis

$$Out\_of\_gate\ q \tag{H}$$

Procederemos por casos en q. Probaremos que, para cada estado q que que satisface ($Out\_of\_gate\ q$), se cumple $q \models \langle\langle\Sigma\rangle\rangle\Diamond In\_gate$. Dado que el operador $\langle\langle\rangle\rangle\Diamond$ está definido en función del operardor $\mathcal{U}$, la proposición a probar es la siguiente:

$$q \models \langle\langle\Sigma\rangle\rangle\top\,\mathcal{U}\ In\_gate$$

Es decir, debemos encontrar un conjunto de estrategias (F : $StrategySet\ \Sigma$) tal que

- $(In\_gate\ q)$, *si aplicamos el contructor until_here, o bien*
- $(\top\ q) \wedge \forall q' : State, q' \in sucStSet(\langle\rangle, q, F) \rightarrow \langle\langle\Sigma\rangle\rangle \top\, \mathcal{U}\, In\_gate$, *si aplicamos en constructor until_later*

Proponemos las siguientes estrategias para cada uno de los jugadores:

$$
\begin{aligned}
f_t : Strategy := \quad & fun\ (qs : \mathrm{seq}\ State)(q : State) \Rightarrow \\
& match\ q\ with \\
& \mid q_{out} \Rightarrow request \\
& \mid q_{req} \Rightarrow idle \\
& \mid q_{gran} \Rightarrow enter \\
& \mid q_{in} \Rightarrow idle \\
& end
\end{aligned}
$$

$$
\begin{aligned}
f_c : Strategy := \quad & fun\ (qs : \mathrm{seq}\ State)(q : State) \Rightarrow \\
& match\ q\ with \\
& \mid q_{out} \Rightarrow idle \\
& \mid q_{req} \Rightarrow grant \\
& \mid q_{gran} \Rightarrow idle \\
& \mid q_{in} \Rightarrow reopen \\
& end
\end{aligned}
$$

De donde F resulta

$$
\begin{aligned}
F : StrategySet\ \Sigma := & fun\ (p : Player)(H : \Sigma\ q) \Rightarrow \\
& (match\ p\ with\ Train \Rightarrow f_t \mid Controller \Rightarrow f_c)
\end{aligned}
$$

A partir de la definición de la fórmula de estados Out_of_gate, observamos que sólo hay tres posibles valores para q que cumplen con la hipótesis H:

- $q = q_{gran}$
- $q = q_{req}$
- $q = q_{out}$

Si $q = q_{in}$, entonces la hipótesis H se reduce a False, y por lo tanto la demostración es trivial, eliminando esta hipótesis.

Demostraremos cada caso por separado, mediante los siguientes lemas:

Lema 5.1.2 *Si los jugadores en* Σ *siguen las estrategias en* F*, entonces*

$$q_{gran} \models \langle\langle \Sigma \rangle\rangle \top \mathcal{U}\ In\_gate$$

Demostración *Dado que* $q_{gran} \not\models In\_gate$*, la aplicación del constructor until_here no es una posibilidad razonable. Aplicaremos el constructor until_later con* F *como argumento, de donde surgen las siguientes obligaciones de prueba:*

(1) $\top\ q_{gran}$

(2) $\forall q' : State, q' \in sucStSet(\langle\rangle, q_{gran}, F) \rightarrow q' \models \langle\langle \Sigma \rangle\rangle \top \mathcal{U}\ In\_gate$

La prueba de (1) es trivial, a partir de la definición de $\top$*. Para demostrar (2), asumamos un estado* q' *tal que*

$$q' \in sucStSet(\langle\rangle, q, F) \tag{HSuc}$$

Debemos demostrar

$$q' \models \langle\langle \Sigma \rangle\rangle \top \mathcal{U}\ In\_gate$$

La hipótesis (HSuc) *nos asegura la existencia de un vector de movimientos* m *tal que*

(a) $\forall (p : Player)(H : \Sigma\ p), F\ H\ \langle\rangle\ q_{gran} = m\ p$

(b) $(q_{gran}, m) \rightsquigarrow q'$

A partir de la definición de F*, y sabiendo que siempre es posible encontrar un término del tipo* $(\Sigma\ p)$ *para cualquier* $p : Player$*, la parte (a) nos permite demostrar las siguientes hipótesis:*

$$\begin{aligned} enter &= m\ Train \\ idle &= m\ Controller \end{aligned} \tag{H'}$$

Por lo tanto, concluimos que $m = \langle enter, idle \rangle$*. Reescribiendo esta igualdad en (b), obtenemos la hipótesis*

$$(q_{gran}, \langle enter, idle \rangle) \rightsquigarrow q' \tag{Htrans}$$

A partir de esta hipótesis y la definición de la relación trans (5.1), observamos que el único constructor posible de la relación trans que nos provee una prueba de (Htrans) *es transGra1, y por lo tanto el único valor posible para* q' *es* q_{in}*.*

Podemos entonces reescribir nuestro objetivo de la siguiente manera:

$$q_{in} \models \langle\langle \Sigma \rangle\rangle \top \mathcal{U}\ In\_gate$$

Dado que la fórmula In_gate es válida en q_{in}, aplicando el constructor until_later nuestro objetivo se reduce a

$$In\_gate\ q_{in}$$

que se demuestra trivialmente a partir de la definición de In_gate. □

Lema 5.1.3 *Si los jugadores en Σ siguen las estrategias en F, entonces*

$$q_{req} \models \langle\langle \Sigma \rangle\rangle \top \mathcal{U}\ In\_gate$$

Demostración *La demostración es similar a la del lema 5.1.2. Aplicando el constructor until_later con F como argumento, y a partir de la definición de este conjunto de estrategias, se observa que el único sucesor posible de F en q_{req} es q_{gran}. Por lo tanto, el objetivo a probar resulta*

$$q_{gran} \models \langle\langle \Sigma \rangle\rangle \top \mathcal{U}\ In\_gate$$

Aplicando el lema 5.1.2, la demostración queda completa. □

Lema 5.1.4 *Si los jugadores en Σ siguen las estrategias en F, entonces*

$$q_{out} \models \langle\langle \Sigma \rangle\rangle \top \mathcal{U}\ In\_gate$$

Demostración *La demostración es similar a la de los lemas precedentes, y utiliza los lemas 5.1.2 y 5.1.3.* □

Utilizando estos tres lemas, la demostración del teorema queda completa. □

El resto de las propiedades se demuestra de forma similar, y el lector interesado puede consultar la formalización Coq.

5.2. Un problema no acotado

En esta sección formalizaremos un protocolo de control de un paso nivel, en el cual existe un número arbitrariamente grande de trenes que compiten por entrar a una compuerta cuyo controlador debe asegurar determinadas propiedades de seguridad y vitalidad (sólo un tren puede estar atravesando la compuerta, todos los pedidos deben ser procesados, etc.).

Nuestro modelo es una generalización del presentado en la sección 5.1.

5.2.1. Modelizando el sistema mediante (E)CGS

Como modelo del sistema proponemos una representación mediante una estructura de juegos concurrente extendida $S_{T\infty}$. Describimos a continuación cada una de las componentes de dicha estructura.

Jugadores Identificaremos a cada tren del sistema con un identificador, que representaremos mediante un número natural:

$$\mathit{Player} : \mathit{Set} := \mathit{Train} : \mathit{Id} \to \mathit{Player} \mid \mathit{Controller} : \mathit{Player},$$

donde $\mathit{Id} := \mathbb{N}$. Es decir, los componentes del sistema son:

- El controlador: *Controller*
- Los trenes: *Train* 0, *Train* 1, *Train* 2, ...

Si $n : \mathit{Id}$, utilizaremos la notación t_n para referirnos al componente (*Train* n) Observemos que para el conjunto de jugadores estamos haciendo uso de nuestra definición extendida de CGS, dado que en tal modelo el número de jugadores es siempre finito.

Estados Al igual que para el conjunto de jugadores, y a diferencia del sistema presentado en la sección 5.1, el número de estados del sistema no es finito. Sin embargo, podemos representarlo de forma finita. En cada estado debe mantenerse información respecto a qué trenes solicitan el ingreso a la compuerta, qué tren ha sido autorizado a ingresar y, finalmente, qué tren se encuentra dentro de la compuerta. Para identificar

los trenes que han solicitado el ingreso, definimos una *función de solicitud*, de la siguiente manera:

$$Petition := Id \to Bool$$

Para una función $f : Petition$, decimos que el tren t_n solicitó el ingreso a la compuerta si $(f\ n = true)$

El conjunto de estados del sistema se define mediante el siguiente tipo inductivo:

$$\begin{aligned} State : Set \quad := \quad & \mid q_{out} : State \\ & \mid q_{req} : Petition \to State \\ & \mid q_{gran} : Petition \to Id \to State \\ & \mid q_{in} : Petition \to Id \to State \end{aligned}$$

donde cada estado representa las siguientes situaciones:

- q_{out}: Todos los trenes se encuentran fuera de la compuerta, y no hay pedidos de ingreso por atender
- $(q_{req}\ f)$: Se ha recibido al menos una solicitud de ingreso. Los trenes que han requerido ingresar son aquellos que pertenecen al conjunto $\{t_n \mid f\ n = true\}$
- $(q_{gran}\ f\ k)$: El controlador ha autorizado al tren t_k a ingresar a la compuerta. En este estado se mantiene la información relativa a los pedidos que restan por atender mediante la función f.
- $(q_{in}\ f\ k)$: t_k ha ingresado a la compuerta. La información referente a los pedidos que restan por atender se mantiene en f.

Movimientos Los posibles movimientos del sistema son los siguientes

$$\begin{aligned} Move : Set \quad := \quad & | \; stayOut : Move \\ & | \; request : Move \\ & | \; grant : Id \to Move \\ & | \; delay : Move \\ & | \; deny : Id \to Move \\ & | \; denyAll : Move \\ & | \; enter : Move \\ & | \; relinquish : Move \\ & | \; keepClosed : Move \\ & | \; reopen : Move \\ & | \; idle : Move \end{aligned}$$

Las modificaciones respecto del sistema de la sección 5.1 son las siguientes:

- $(deny\ n)$: Representa un movimiento para el controlador en el cual se rechaza el pedido recibido por el tren t_n
- $(denyAll)$: Representa un movimiento que le permite al controlador cancelar todos los pedidos de ingreso
- $(grant\ n)$: El controlador le otorga el permiso de ingreso al tren t_n

Vectores de movimientos Sean $mt : Id \to Move$, $mc : Move$. Notaremos $\langle mt, mc \rangle$ al vector de movimientos dado por:

$$fun\ p : Player \Rightarrow match\ p\ with\ t_n \Rightarrow mt\ n \mid Controller \Rightarrow mc$$

Transiciones La función de transición se introduce mediante la relación inductiva

$$trans : State \to MoveVector \to State \to Prop,$$

definida de la siguiente manera:

$$
\begin{aligned}
trans \;:=\;& \\
&\mid transOut1 : trans\ q_{out}\ \langle fun\ n \Rightarrow stayOut, idle\rangle\ q_{out} \\
&\mid transOut2 : \forall(f : Petition), (\exists n : Id, f\ n = true) \rightarrow \\
&\quad trans\ q_{out}\ \langle fun\ n \Rightarrow if\ f\ n\ then\ request\ else\ stayOut, idle\rangle\ (q_{req}\ f) \\
&\mid transReq1 : \forall(f : Petition)(n : Id), f\ n = true \rightarrow \\
&\quad trans\ (q_{req}\ f)\ \langle fun\ n \Rightarrow idle, grant\ n\rangle\ (q_{gran}\ (f \oplus \{n \leftarrow false\})\ n) \\
&\mid transReq2 : \forall(f : Petition), \\
&\quad trans\ (q_{req}\ f)\ \langle fun\ n \Rightarrow idle, delay\rangle\ (q_{req}\ f) \\
&\mid transReq3 : \forall(f : Petition)(n : Id), (\exists m : Id, m \neq n \wedge f\ m = true) \\
&\quad \rightarrow trans\ (q_{req}\ f)\ \langle fun\ n \Rightarrow idle, deny\ n\rangle\ (q_{req}\ f \oplus \{n \leftarrow false\}) \\
&\mid transReq4 : \forall(f : Petition)(n : Id), (\forall m : Id, m \neq n \rightarrow f\ m = false) \\
&\quad \rightarrow trans\ (q_{req}\ f)\ \langle fun\ n \Rightarrow idle, deny\ n\rangle\ q_{out} \\
&\mid transReq5 : \forall(f : Petition), trans\ (q_{req}\ f)\ \langle fun\ n \Rightarrow idle, denyAll\rangle\ q_{out} \\
&\mid transGra1 : \forall(f : Petition)(n : Id), \\
&\quad trans\ (q_{gran}\ f\ n)\ \langle fun\ m \Rightarrow if\ m =_b n\ then\ enter\ else\ idle, idle\rangle \\
&\qquad (q_{in}\ f\ n) \\
&\mid transGra2 : \forall(f : Petition)(n : Id), (\forall k : Id, k \neq n \rightarrow f\ k = false) \rightarrow \\
&\quad trans\ (q_{gran}\ f\ n)\ \langle fun\ m \Rightarrow if\ m =_b n\ then\ relinquish\ else\ idle, idle\rangle \\
&\qquad q_{out} \\
&\mid transGra3 : \forall(f : Petition)(n : Id), (\exists k : Id, k \neq n \wedge f\ k = true) \rightarrow \\
&\quad trans\ (q_{gran}\ f\ n)\ \langle fun\ m \Rightarrow if\ m =_b n\ then\ relinquish\ else\ idle, idle\rangle \\
&\qquad (q_{req}\ f) \\
&\mid transIn1 : \forall(f : Petition)(n : Id), \\
&\quad trans\ (q_{in}\ f\ n)\ \langle fun\ n \Rightarrow idle, keepClosed\rangle\ (q_{in}\ f\ n) \\
&\mid transIn2 : \forall(f : Petition)(n : Id), (\forall m : Id, f\ m = false) \rightarrow \\
&\quad trans\ (q_{in}\ f\ n)\ \langle fun\ n \Rightarrow idle, reopen\rangle\ q_{out} \\
&\mid transIn3 : \forall(f : Petition)(n : Id), (\exists m : Id, f\ m = true) \rightarrow \\
&\quad trans\ (q_{in}\ f\ n)\ \langle fun\ n \Rightarrow idle, reopen\rangle\ (q_{req}\ f)
\end{aligned}
$$

En la definición anterior, se usaron las siguientes funciones:

- $=_b$: $Id \rightarrow Id \rightarrow Bool$. Esta función *decide* la igualdad sobre el tipo Id.
- $\oplus$: $Petition \rightarrow Id \rightarrow Bool \rightarrow Petition$, es un operador tal que $(f \oplus \{n \leftarrow b\})$ aplicada a un identificador m devuelve b si $m = n$, y $f\ m$ en caso contrario.

La definición de la función de transición se interpreta de la siguiente manera.

En el estado q_{out} existen dos posibilidades:

- Todos los trenes eligen el movimiento *stayOut*, por lo que el sistema se mantiene en el estado q_{out}
- Un subconjunto no vacío de trenes realiza un pedido de ingreso, y el sistema transiciona al estado $(q_{req}\ f)$, donde f mantiene la información sobre los trenes que solicitan ingresar

Si el sistema se encuentra en el estado $q_{req}\ f$, es el turno del controlador, que puede:

- Otorgarle el permiso a alguno de los trenes que lo había solicitado, eligiendo el movimiento $(grant\ n)$
- Elegir el movimiento *delay*, y mantener el sistema en el mismo estado
- Denegarle el permiso a alguno de los trenes que lo había solicitado, mediante el movimiento $(deny\ n)$. En este caso, el sistema transiciona a
  - El estado q_{out} si el pedido de t_n era el único por ser atendido
  - El estado $(q_{req}\ f \oplus \{n \leftarrow false\})$ si todavía restan permisos por ser atendidos
- Denegarle el pedido a todos los trenes, transicionando el sistema al estado q_{out}

En el estado $(q_{gran}\ f\ n)$ es el tren que ha obtenido el permiso quien decide el próximo estado, que puede ser:

- $(q_{in}\ f\ n)$, si t_n elige el movimiento *enter*
- q_{out}, si el tren decide rechazar el permiso y no quedan pedidos por atender
- $(q_{req}\ f)$, si el tren decide rechazar el permiso y restan solicitudes por atender

Finalmente, en el estado $(q_{in}\ f\ n)$, el controlador decide entre los movimientos:

- *keepClosed*, que mantiene el estado actual

- *reopen*, que transiciona al estado
  - q_{out} si el pedido de t_n era el único por ser atendido
  - $q_{req}\ f$, si todavía restan pedidos por atender

Observación Si el estado inicial del sistema es q_{out}, entonces la función de transición nos asegura que nunca se dará una situación en la cual el sistema se encuentre en un estado $(q_{req}\ f)$ tal que $(\forall n : Id, fn = false)$.

Coaliciones A diferencia del modelo acotado, en este sistema existe un número infinito (no numerable) de coaliciones. A modo de ejemplo presentamos algunas de ellas, una de las cuales será de especial interés para la demostración de propiedades de la sección 5.2.2

- *Trenes*:

$$T := fun\ p : Player \Rightarrow match\ p\ with\ Train\ n \Rightarrow \{1\} \mid Controller \Rightarrow \{\ \}$$

- *Controlador*:

$$C := fun\ p : Player \Rightarrow match\ p\ with\ Train\ n \Rightarrow \{\ \} \mid Controller \Rightarrow \{1\}$$

- *Un tren particular:* Si $n : Id$, la coalición $T_n = \{t_n\}$ se define de la siguiente manera

$$\begin{aligned} T_n := fun\ p : Player \Rightarrow \quad & match\ p\ with \\ & Train\ k \Rightarrow if\ n =_b k\ then\ \{1\}\ else\ \{\ \} \\ & \mid Controller \Rightarrow \{\ \} \end{aligned}$$

- Si $n : Id$, la coalición $A = \{t_n, Controller\}$ se define de la siguiente manera

$$A := T_n \uplus C$$

Fórmulas de estados Al igual que para el conjunto de jugadores y estados, haremos uso de nuestra generalización de CGS para definir fórmulas de estados, no limitándonos a un número finito de situaciones observables. Algunas fórmulas de estados que resultarán de particular interés a la hora de enunciar y demostrar propiedades sobre el sistems son las siguientes:

- No hay un tren en la compuerta

$$Out\_of\_gate := fun\ q : State \Rightarrow match\ q\ with\ q_{in}\ f\ m \Rightarrow False \mid \_ \Rightarrow True$$

- El sistema se encuentra en el estado $(q_{req}\ f)$, para algún $f : Petition$:

$$Request := fun\ q : State \Rightarrow match\ q\ with\ q_{req}\ f \Rightarrow True \mid \_ \Rightarrow False$$

  Teniendo en cuenta la observación 5.2.1, la validez de la fórmula *Request* nos asegura que existe al menos una solicitud sin atender.

- El tren t_n tiene una solicitud sin atender

$$\begin{array}{l} HasRequest(n : Id) := \\ \quad fun\ q : State \Rightarrow \\ \qquad match\ q\ with \\ \qquad\quad \mid q_{req}\ f \Rightarrow if\ (f\ n)\ then\ True\ else\ False \\ \qquad\quad \mid q_{gran}\ f\ m \Rightarrow if\ (f\ n)\ then\ True\ else\ False \\ \qquad\quad \mid q_{in}\ f\ m \Rightarrow if\ (f\ n)\ then\ True\ else\ False \\ \qquad\quad \mid q_{out} \Rightarrow False \\ \qquad end \end{array}$$

- *El tren t_n tiene permiso para ingresar a la compuerta*:

$$\begin{array}{l} Granted(n : Id) := \\ \quad fun\ q : State \Rightarrow \\ \qquad match\ q\ with \\ \qquad\quad \mid q_{gran}\ f\ m \Rightarrow if\ f\ n =_b m\ then\ True\ else\ False \\ \qquad\quad \mid \_ \Rightarrow False \\ \qquad end \end{array}$$

- El tren t_n se encuentra dentro de la compuerta:

$$\begin{array}{l} In(n : Id) := \\ \quad fun\ q : State \Rightarrow \\ \qquad match\ q\ with \\ \qquad\quad \mid q_{in}\ f\ m \Rightarrow if\ f\ n =_b m\ then\ True\ else\ False \\ \qquad\quad \mid \_ \Rightarrow False \\ \qquad end \end{array}$$

- Todos los trenes se encuentran fuera de la compuerta, y no hay pedidos sin atender ni permisos otorgados

$$Out\_no\_req := fun\ q : State \Rightarrow match\ q\ with\ q_{out} \Rightarrow True \mid \_ \Rightarrow False$$

- El sistema se encuentra en el estado $(q_{gran}\ f\ m)$, para algún $f : Petition$ y $m : Id$

$$InGranted := fun\ q : State \Rightarrow match\ q\ with\ q_{gran}\ f\ m \Rightarrow True \mid \_ \Rightarrow False$$

5.2.2. Propiedades del Modelo

La lógica ATL nos permite especificar y demostrar importantes propiedades que la estructura $S_{T\infty}$ satisface.

Algunas de ellas son:

- Sea $n : Id$. Si el sistema se encuentra en el estado q_{out}, entonces la coalición $A := \{t_n, Controller\}$ puede asegurar la entrada de t_n a la compuerta:

$$\langle\langle\varnothing\rangle\rangle\Box\,(Out\_no\_req \rightarrow \langle\langle A\rangle\rangle\Diamond In(n)) \tag{5.7}$$

- Si no hay un tren en la compuerta ni permisos de ingreso otorgados, el controlador puede mantener la compuerta vacía para siempre:

$$\langle\langle\varnothing\rangle\rangle\Box\,((Out\_of\_gate \wedge \neg InGranted) \rightarrow \langle\langle C\rangle\rangle\Box\, Out\_of\_gate) \tag{5.8}$$

- Sean $n, m : Id$, $n \neq m$; y $B := \{t_n, t_m, Controller\}$. Si el tren t_n tiene permiso para ingresar a la compuerta, y t_m tiene pedido en ingreso, entonces la coalicón B puede cooperar de forma tal que el sistema alcance un estado en el cual t_m se encuentre dentro de la compuerta

$$\langle\langle\varnothing\rangle\rangle\Box\,(Granted(n) \wedge HasRequest(m) \rightarrow \langle\langle B\rangle\rangle\Diamond In(m)) \tag{5.9}$$

- La coalición T puede mantener indefinidamente al sistema fuera del estado q_{in}

$$\langle\langle\varnothing\rangle\rangle\Box\,(Out\_of\_gate \rightarrow \langle\langle T\rangle\rangle\Box\, Out\_of\_gate) \tag{5.10}$$

Para ilustrar la forma de demostrar estas propiedades en nuestra formalización, demostraremos a continuación la validez de la propiedad 5.7.

Teorema 5.2.1 *Sea* $n : Id$, *y* $A := \{t_n, Controller\}$. *La siguiente fórmula de estados es válida en* $S_{T\infty}$

$$\langle\langle\varnothing\rangle\rangle\Box\,(Out\_no\_req \rightarrow \langle\langle A\rangle\rangle\Diamond In(n))$$

Demostración *A partir de la observación 5.1.2, es suficiente probar la validez de la fórmula:*

$$\forall q : State, q \models (Out\_no\_req \rightarrow \langle\langle A\rangle\rangle\Diamond In(n))$$

Sea $q : State$ *y la hipótesis*

$$q \models Out\_no\_req \tag{H}$$

Debemos encontrar una prueba de

$$q \models \langle\langle A\rangle\rangle\Diamond In(n)$$

que, aplicando la definición del operador $\Diamond$ *resulta*

$$q \models \langle\langle A\rangle\rangle\top\,\mathcal{U}\,In(n)$$

La demostración será por inducción (análisis de casos) en q. *Observemos que si* $q \neq q_{out}$, *la hipótesis H se reduce a False, por lo cual la demostración es trivial eliminando dicha hipótesis.*

En el caso $q = q_{out}$, *debemos encontrar una estrategia para la coalición* A *que asegure la entrada del tren* t_n *a la compuerta. Proponemos la siguiente estrategia para cada uno de los jugadores en* A*:*

$$\begin{aligned} f_{t_n} : Strategy := \;& fun\ (qs : \mathrm{seq}\ State)(q : State) \Rightarrow \\ & match\ q\ with \\ & \quad \mid q_{out} \Rightarrow request \\ & \quad \mid q_{req}\ f \Rightarrow idle \\ & \quad \mid q_{gran}\ f\ m \Rightarrow (if\ n =_b m\ then\ enter\ else\ idle) \\ & \quad \mid q_{in}\ f\ m \Rightarrow idle \\ & end \end{aligned}$$

$$
\begin{aligned}
f_c : \mathit{Strategy} := \quad & \mathit{fun}\ (qs : \mathrm{seq}\ \mathit{State})(q : \mathit{State}) \Rightarrow \\
& \quad \mathit{match}\ q\ \mathit{with} \\
& \quad\ \mid q_{out} \Rightarrow \mathit{idle} \\
& \quad\ \mid q_{req}\ f \Rightarrow (\mathit{if}\ f\ n\ \mathit{then}\ \mathit{grant}\ \mathit{else}\ \mathit{denyAll}) \\
& \quad\ \mid q_{gran}\ f\ m \Rightarrow \mathit{idle} \\
& \quad\ \mid q_{in}\ f\ m \Rightarrow \mathit{reopen} \\
& \quad \mathit{end}
\end{aligned}
$$

A partir de f_{t_n} puede construirse un conjunto de estrategias F_{T_n} para la coalición unitaria T_n. De igual manera, para la coalición C construimos el conjunto de estrategias $F_C = \{f_c\}$. Usando la unión de estrategias definida en la sección 4.1 el conjunto de estrategias F_A resulta

$$F_A := F_{T_n} \uplus F_C$$

Demostraremos entonces que, siguiendo las estrategias en F_A, la coalición A puede asegurar la validez de la fórmula

$$q_{out} \models \langle\langle A \rangle\rangle \top\, \mathcal{U}\, \mathit{In}(n)$$

Dado que el estado q_{out} no satisface la fórmula de estados $\mathit{In}(n)$, aplicaremos el constructor until_there con F_A como argumento, con lo cual nuestro objetivo resulta

$$(\top\ q_{out}) \wedge \big(\forall q' : \mathit{State}, q' \in \mathit{sucStSet}(\langle\rangle, q_{out}, F_A) \rightarrow q' \models \langle\langle A \rangle\rangle \top\, \mathcal{U}\, \mathit{In}(n)\big)$$

La primera parte de esta conjunción es trivial. Para demostrar la segunda, sea q' : State tal que

$$q' \in \mathit{sucStSet}(\langle\rangle, q_{out}, F_A) \qquad \text{(HSuc)}$$

Es fácil ver que el único estado que satisface la hipótesis (HSuc) *es un estado de la forma $(q_{req}\ f)$ donde $f(n) = true$. En efecto, dicha hipótesis nos asegura la existencia de un vector de movimientos mv que satisface las siguientes hipótesis:*

$$\forall (p : \mathit{Player})(H : A\ p), F_A\ p\ H\ \langle\rangle\ q_{out} = mv\ p \qquad \text{(HF-mv)}$$

$$\langle q_{out}, mv \rangle \rightsquigarrow q' \qquad \text{(Htrans)}$$

A partir de la definición de la función de transición, existen sólo dos posibles valores para q' *que satisfacen* (Htrans)*, a saber:*

(1) $q' = q_{out}$*. En este caso, necesariamente debe cumplirse* $\forall(m : Id), mv\ m = idle$*, lo cual, en el caso* $m = n$ *contradice la hipótesis* (HF-mv)*, que asegura* $mv\ n = request$

(2) $q' = q_{req}\ f$*, y* $f\ n = true$*. En efecto, si un conjunto no vacío de jugadores (entre los que se encuentra* t_n*) elige el movimiento request, las hipótesis* (HF-mv) *y* (Htrans) *resultan consistentes.*

Por lo tanto, nuestro objetivo puede reescribirse de la siguiente manera:

$$f\ n = true \rightarrow (q_{req}\ f) \models \langle\langle A \rangle\rangle \top \mathcal{U}\ In(n)$$

Aplicando nuevamente el constructor until_there con F_A *como argumento, y siguiendo un razonamiento similar al anterior, puede probarse que el conjunto de estados* $sucStSet(\langle\rangle, (q_{req}\ f), F_A)$ *está compuesto sólo por estados de la forma* $(q_{gran}\ f\ n)$ [1] *, y por lo tanto la demostración del teorema se reduce a:*

$$(q_{gran}\ f\ n) \models \langle\langle A \rangle\rangle \top \mathcal{U}\ In(n)$$

Un razonamiento análogo al anterior (aplicación de until_there y análisis sobre el conjunto $sucStSet(\langle\rangle, (q_{gran}\ f\ n), F_A)$*), reduce la demostración a*

$$(q_{in}\ f\ n) \models \langle\langle A \rangle\rangle \top \mathcal{U}\ In(n)$$

la cual es trivial aplicando el constructor until_here.

□

[1] Pues, siguiendo F_A, el controlador elegirá otorgarle el permiso de ingreso a t_n

5.2.3. Relación con el modelo finito

El conjunto de estrategias F_A fue definido con la intención de *"guiar"* al sistema a través de la secuencia de estados

$$\langle q_{out}, (q_{req}\ f_1), (q_{gran}\ f_2\ n), (q_{in}\ f_2\ n)\rangle$$

donde f_1 y f_2 son tales que $f_1(n) = true$ y $f_2(n) = false$. El teorema 5.2.1 refleja esta situación en la estructura de la demostración. En cada paso, demostramos que la sucesión de estados al seguir F_A debía ser de esa forma, y que por lo tanto eventualmente t_n estaría dentro de la compuerta.

Diversas estrategias pueden definirse para demostrar la misma propiedad. En tal caso, la demostración del teorema estará siempre guiada por dicha definición.

Si comparamos el conjunto de estrategias F_A con el conjunto F utilizado en el modelo acotado para demostrar una propiedad similar, vemos que existe una correspondencia entre las acciones de cada uno de los jugadores en ambas estrategias. Si bien la correspondencia no será siempre posible, pensamos que razonar sobre modelos finitos es de gran utilidad antes de pasar a casos no acotados donde técnicas de demostración más complejas son necesarias.

Consideramos que una integración posible entre *model checking* y un sistema de verificación como el de este trabajo puede derivarse de estas ideas. En efecto, si utilizamos un algoritmo de *model checking* como el propuesto en [21], donde a partir del proceso de verificación automática es posible extraer estrategias que hacen válidas las propiedades en estudio, esta información puede aprovecharse al momento de derivar nuevas estrategias para el modelo no acotado, generalizando las estrategias del modelo finito.

6

Conclusiones

6.1. Conclusiones

En este trabajo presentamos una formalización de la lógica ATL y su modelo semántico (CGS) en el cálculo de construcciones coinductivas, junto con una implementación en el asistente de pruebas *Coq*.

Hemos generalizado las estructuras de juego concurrentes para poder trabajar con sistemas no acotados o paramétricos, mediante la abstracción del conjunto de estados y jugadores del sistema en tipos no necesariamente finitos. Las nociones de coalición y estrategias fueron formalizadas utilizando tipos de datos con contenido computacional, abriendo la posibilidad de extraer prototipos funcionales a partir del modelo del sistema. Las definiciones relativas a trazas de ejecución se introdujeron mediante tipos coinductivos.Adicionalmente, se han demostrado propiedades generales que luego pueden aplicarse a diferentes modelos concretos. Por ejemplo, en la demostración del teorema 5.2.1 referente a las propiedades del protocolo de control de un paso a nivel, se utilizaron los resultados relativos a la unión de estrategias demostradas en la sección 4.1.

En cuanto a la formalización de ATL, se implementó un sistema deductivo para dicha lógica. Al igual que en CGS, hemos utilizado el poder expresivo del cálculo de construcciones para extender ATL con algunas características que consideramos importantes para la verificación de sistemas reactivos. En primer lugar, representamos las fórmulas de estados como relaciones unarias, lo cual no restringe las fórmulas atómicas a ser

simplemente variables proposicionales. La formalización de los operadores temporales $\langle\langle\rangle\rangle\Box$ y $\langle\langle\rangle\rangle\mathcal{U}$ se basan en las propiedades de punto fijo que poseen. El operador *until*, al ser el *menor* punto fijo de la ecuación que lo caracteriza, se formaliza mediante un tipo inductivo. El operador *always* está definido como el *mayor* punto fijo de una ecuación, y por lo tanto su formalización es mediante el uso de tipos coinductivos. Esta semántica resulta más general que la presentada en [2], y consideramos captura mejor la propiedad en cuestión.

El sistema deductivo ha sido utilizado para la demostración de teoremas ATL. Consideramos que nuestra formalización ha resultado adecuada para la demostración de tales propiedades. Contar con teoremas generales de la lógica resulta de especial interés en la verificación de sistemas concretos. Dichos resultados pueden ayudarnos a simplificar las demostraciones de propiedades sobre modelos particulares, pudiendo transformar las fórmulas a demostrar por otras equivalentes pero más simples. Observemos que esta última ventaja resulta igualmente útil si el modelo concreto se verificará de manera automática o asistida, dado que simplificaciones previas pueden reducir el tiempo de ejecución de algoritmos de *model checking* (si utilizamos verificación automática), o el tamaño de los términos de prueba (en el caso de verificación deductiva).

Finalmente, nuestra formalización fue utilizada para la especificación y verificación de dos casos de estudio. Para ambos modelos demostramos importantes propiedades de seguridad, haciendo uso de teoremas generales ya demostrados en la formalización de ATL. Si bien uno de estos casos de estudio puede ser verificado de forma automática, el otro, al poseer un número no acotado de estados, no puede ser verificado mediante algoritmos de *model checking*, y requiere de técnicas de verificación más expresivas. Consideramos que la posibilidad de razonar sobre sistemas concurrentes paramétricos o con un número de estados no acotados resulta un importante aporte de este trabajo.

La ventaja de utilizar un demostrador de teoremas como *Coq* se ha puesto en evidencia en varios aspectos. Principalmente, la posibilidad de verificar (mediante el *type checker* de Coq) los términos de prueba nos garantiza la corrección de las demostraciones. Entre las otras ventajas podemos citar el uso que hemos hecho de la librería estándar de *Coq*, lo cual nos evitó definir tipos de datos (y demostrar propiedades sobre éstos)

necesarios para nuestra formalización.

6.2. Trabajos Futuros

En esta sección proponemos algunas posibles extensiones a la formalización aquí presentada. Algunas de ellas son:

- Extender la formalización a *Fair-ATL*: Las restricciones de equidad (*fairness*) reducen el número de trazas posibles de ejecución, contemplando sólo aquellas que satisfacen un cierto número de restricciones relacionadas con el comportamiento de los componentes. Por ejemplo, en el caso de estudio presentado en el capítulo 5, una restricción de equidad posible es sobre el controlador, no permitiéndole que siempre que se encuentre en el estado q_{req} escoja rechazar pedidos. La lógica *Fair-ATL* [2] modifica la semántica de ATL para tener en cuenta tales restricciones. Una interesante extensión de nuestra formalización podría dirigirse en este sentido.
- Formalizar ATL*: Mientras que las fórmulas de ATL requiren preceder cada operador temporal por una cuantificación de trazas de la forma $\langle\langle A \rangle\rangle$, ATL* permite anidar operadores temporales de forma arbitraria, extendiendo el poder expresivo de la lógica [1]. Por ejemplo, la fórmula $\langle\langle A \rangle\rangle \Box\Diamond\varphi$ es sintácticamente correcta en ATL*, y es válida si los jugadores en A pueden asegurar en todo estado futuro que *eventualmente* φ será cierta [2]. La complejidad de los algoritmos de *model checking* para ATL* convierte la verificación automática de esta lógica en un problema intratable. Consideramos que es posible una extesión de nuestra formalización para representar la semántica de ATL*.
- Verificación de protocolos de intercambio de firmas digitales. Una interesante aplicación de ATL se presenta en [8] y [22]. El problema en estudio son "protocolos de firma de contratos digitales", los cuales analizan la dificultad de intercambiar firmas digitales sin producir situaciones de ventaja para ningulo de los involucrados (un agente A está en ventaja si ha obtenido un certificado de la firma de otro componente B, puede evitar que B reciba la firma de A). Diversos

[1] En particular, las restricciones de equidad pueden codificarse en ATL*
[2] Puede demostrarse que esta fórmula no es representable en ATL

algoritmos se han propuesto para garantizar la equidad durante todo el proceso de firma. En [8] se presenta un análisis formal de la corrección de algunos de ellos, utilizando ATL y el *model checker* MOCHA.

Sin embargo, para situaciones donde el número de firmantes es mayor que dos, dicho análisis formal se reduce a verificar los algoritmos con tres y cuatro componentes. Consideramos que la verificación de esta clase de protocolos utilizando la formalización aquí presentada puede ser una interesante línea de investigación.

Bibliografía

[1] *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, London, UK, 2002.

[2] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49:672–713, 2002.

[3] Rajeev Alur, Luca de Alfaro, Radu Grosu, Tom Henzinger, Minsu Kang, Christoph Kirsch, Rupak Majumdar, Freddy Mang, and Bow-Yaw Wang. jmocha: A model-checking tool that exploits design structure. In *Proceedings of the 23rd Annual IEEE/ACM International Conference on Software*, pages pp. 835–836. IEEE Computer Society Press, January 2001.

[4] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Lecture Notes in Computer Science*, 1536:23–60, 1998.

[5] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. Mocha: Modularity in model checking. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 521–525, London, UK, 1998. Springer-Verlag.

[6] Yves Bertot. Coq in a hurry. Nov 2008.

[7] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. CoqÁrt: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[8] Rohit Chadha, Steve Kremer, and Andre Scedrov. Formal analysis of multiparty contract signing. *J. Autom. Reason.*, 36(1-2):39–83, 2006.

[9] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[10] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[11] Solange Coupet-Grimal. Ltl in coq. Contributions to the coq system, 2002.

[12] Solange Coupet-Grimal. An axiomatization of linear temporal logic in the calculus of inductive constructions. *Journal of Logic and Computation*, 13(6):801–813, 2003.

[13] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2006. Version 8.1.

[14] E. Allen Emerson. Temporal and modal logic. pages 995–1072, 1990.

[15] E. Giménez. *A Calculus of Infinite Constructions and its application to the verification of communicating systems.* PhD thesis, Ecole Normale Supérieure de Lyon, 1996.

[16] Eduardo Gimnez and Projet Coq. A tutorial on recursive types in coq. Technical report, 1998.

[17] Valentin Goranko and Govert van Drimmelen. Complete axiomatization and decidability of alternating-time temporal logic. *Theoretical Compuer. Sciences*, 353(1):93–117, 2006.

[18] Thomas A. Henzinger, Xiaojun Liu, Shaz Qadeer, and Sriram K. Rajamani. Formal specification and verification of a dataflow processor array. In *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 494–499, Piscataway, NJ, USA, 1999. IEEE Press.

[19] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[20] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about computer systems (first edition).* Cambridge University Press,

2000. ISBN: 0521656028 (paperback), ISBN: 0521652006 (hardback), 387pp. http://www.cs.bham.ac.uk/research/lics/.

[21] W. J. Jamroga. Strategic planning through model checking of atl formulae. In L. Rutkowski, J. Siekmann, R. Tadeusiewicz, and L. A. Zadeh, editors, *Artificial Intelligence and Soft Computing (ICAISC 2004)*, volume 3070 of *Lecture notes in Computer Science*, pages 879–884, Berlin, 2004. Springer Verlag.

[22] Steve Kremer and Jean-François Raskin. A game-based verification of non-repudiation and fair exchange protocols. *J. Comput. Secur.*, 11(3):399–429, 2003.

[23] Carlos Luna. Computation tree logic for reactive systems and timed computation tree logic for real time systems. Contributions to the coq system, 2000.

[24] Carlos Luna. Especificación y análisis de sistemas de tiempo real en teoría de tipos. Master's thesis, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2000.

[25] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, Computer Science Dept., Univ. of Edinburgh, 1992.

[26] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[27] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Pittsburgh, PA, USA, 1992.

[28] Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In *TLCA '93: Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, London, UK, 1993. Springer-Verlag.

[29] Amir Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.