Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Control y exploración de ambientes de sistemas reactivos

Tesis de Licenciatura en Ciencias de la Computación

Maureen Keegan

Director: Sebastián Uchitel

Codirector:

Buenos Aires, 2019

CONTROL Y EXPLORACIÓN DE AMBIENTES DE SISTEMAS REACTIVOS

Los sistemas autoadaptativos necesitan poder comprender autónomamente el ambiente en el que operan y usar este conocimiento para controlarlo de manera tal que los objetivos del sistema se cumplan. ¿Cómo se puede lograr esto si el ambiente es desconocido?

En esta tesis formalizamos el problema de control y exploración para ambientes de sistemas reactivos. En el caso en que las metas son realizables en el ambiente dado, la solución genera una estrategia que controla el ambiente y las cumple, en caso contrario concluye que no es posible cumplirlas. Presentamos una solución restringida a objetivos de tipo GR(1), que usa MTS (*Modal Transition Systems*) para representar el conocimiento parcial del comportamiento del ambiente. Esta solución se basa en la síntesis de controladores para MTS para tomar las decisiones de exploración, de manera tal que cada paso provea información sobre el ambiente o contribuya a cumplir los objetivos.

Palabras claves: Síntesis de controladores, Exploración, MTS, LTS, GR(1), FLTL, Conocimiento parcial.

AGRADECIMIENTOS

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce sapien ipsum, aliquet eget convallis at, adipiscing non odio. Donec porttitor tincidunt cursus. In tellus dui, varius sed scelerisque faucibus, sagittis non magna. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Mauris et luctus justo. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Mauris sit amet purus massa, sed sodales justo. Mauris id mi sed orci porttitor dictum. Donec vitae mi non leo consectetur tempus vel et sapien. Curabitur enim quam, sollicitudin id iaculis id, congue euismod diam. Sed in eros nec urna lacinia porttitor ut vitae nulla. Ut mattis, erat et laoreet feugiat, lacus urna hendrerit nisi, at tincidunt dui justo at felis. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Ut iaculis euismod magna et consequat. Mauris eu augue in ipsum elementum dictum. Sed accumsan, velit vel vehicula dignissim, nibh tellus consequat metus, vel fringilla neque dolor in dolor. Aliquam ac justo ut lectus iaculis pharetra vitae sed turpis. Aliquam pulvinar lorem vel ipsum auctor et hendrerit nisl molestie. Donec id felis nec ante placerat vehicula. Sed lacus risus, aliquet vel facilisis eu, placerat vitae augue.

A mi persona favorita.

Índice general

1. INTRODUCCIÓN

Los sistemas autoadaptativos necesitan poder comprender autónomamente el ambiente en el que operan y usar este conocimiento para controlarlo de manera tal que los objetivos del sistema se cumplan.

Consideremos un componente autoadaptativo que tiene que coordinar servicios cuyos protocolos son completamente o parcialmente desconocidos. ¿Cómo puede un componente ganar suficiente conocimiento sobre el comportamiento de dichos servicios para generar una estrategia que los coordine correctamente de acuerdo a cierto objetivo?

Alternativamente, ¿cómo puede un componente concluir, si fuera el caso, que en el ambiente en que se encuentra las metas son *irrealizables*? El componente debe explorar su ambiente estimulando los servicios y sensando su comportamiento para ganar suficiente conocimiento para poder garantizar las metas o declarar insatisfacibilidad.

Una dificultad clave de controlar un ambiente desconocido es que el descubrimiento es obstaculizado por comportamiento no controlable. Por ejemplo, un servicio puede tener varios valores de retorno que dependen de alguna configuración interna que no puede ser modificada por el componente que lo explora. Es razonable asumir, a su vez, que el ambiente puede comportarse de manera adversaria, es decir deliberadamente no exhibir ciertas respuestas a los estímulos que recibe. Por lo tanto no existe garantía de que durante la exploración todo el comportamiento sea descubierto. Esto hace que los enfoques bifásicos (sean a través de aprendizaje de autómatas [1–3] o a través de exploración aleatoria) en donde primero se explora totalmente el comportamiento del ambiente y luego se computa una estrategia sean inadecuados.

En esta tesis formalizamos el problema de controlar un sistema reactivo con un ambiente desconocido que busca lograr metas especificadas en lógica linear temporal. La noción clave que se explota para evitar una estrategia bifásica es asegurar que cada acción tomada aumente el conocimiento que se tiene del ambiente hasta el momento, o mueva el sistema un paso más cerca de alcanzar sus metas.

Mostramos cómo los *Modal Transitions Systems* (MTS) [4] pueden ser usados para caracterizar tanto el conocimiento parcial del ambiente con el que comienza el proceso de exploración como el conocimiento que se acumula durante el relevamiento.

También presentamos una solución al problema de control y exploración restringido a metas del tipo GR(1) [5]. La solución se basa en síntesis de controladores para MTS [6]. Asumimos que el comportamiento del ambiente se puede caracterizar con un sistema de transición etiquetado o *labelled transition system* (LTS) determinístico. Esto nos permite mostrar que se puede utilizar síntesis para generar estrategias de exploración y control que garanticen las metas o que concluyan que son irrealizables.

El procedimiento que proponemos garantiza que, dado un ambiente en el cual las metas son realizables, incluso si el ambiente deliberadamente intenta evitar la exploración de su comportamiento, es posible controlarlo sin necesariamente conseguir conocimiento total.

El uso de síntesis de MTS nos permite guiar el proceso de control y exploración a través de acciones controlables que logran los objetivos o encuentran comportamiento previamente desconocido. Los controladores extraídos mediante esta síntesis no dependen de la exploración total para concluir si la meta es irrealizable o para efectivamente realizarla. Si se descubre nuevo comportamiento, se integra al conocimiento actual: requerimos que

el ambiente sea capaz de reportar consistentemente su estado actual, para poder permitir la detección de ciclos. Si el nuevo comportamiento nos lleva a un estado en donde las metas son irrealizables, entonces debe ser posible reiniciar el ambiente para poder reintentar controlar y explorar con el nuevo conocimiento adquirido.

En el resto de esta tesis proveemos motivación y un ejemplo en la sección 2, definiciones preliminares en la sección 3 y luego formalizamos el problema de control y exploración y proveemos una solución en la sección 4-5. Reportamos validación en la sección 6 y finalizamos con trabajos asociados y conclusiones.

2. MOTIVACIÓN Y EJEMPLO

Consideremos el siguiente ejemplo tomado de [7]. Queremos un cliente que interactúe con un servicio de préstamos de libros que permita a los usuarios pedir libros y recibirlos en sus hogares. Una vez leídos, los usuarios los devuelven por correo.

Para relevar el servicio de préstamos y su interfaz se puede utilizar un servicio como *Google APIs Discovery Service*. Sin embargo, en la práctica el protocolo (el orden en que se debe llamar a los métodos del servicio) para interactuar correctamente con el servicio está, en el mejor de los casos, parcialmente disponible. El problema consiste en construir un cliente que pueda deducir como usar el servicio y, a su vez, intente safisfacer sus objetivos de ser posible.

Por ejemplo, en la Figura 2.1 mostramos el protocolo de un servicio de préstamos de libros muy simple (la interacción entre el usuario y el cliente no es parte del protocolo). Cuando un usuario quiere pedir prestado un libro, genera el pedido $usrReq$. Luego el cliente busca el libro ($queryBook$). El servicio devuelve una lista de los libros disponibles ($list$) entre los cuales el cliente puede elegir uno ($select$) y recibirlo en su hogar ($deliver$). Si ningún libro está disponible ($unavailable$) el cliente puede reintentar una vez trascurrido cierto tiempo o informar al usuario que el pedido no puede ser satisfecho ($abort$).

Algunos servicios (como por ejemplo el de la Figura 2.2) permiten a los usuarios reservar una copia de un libro actualmente no disponible ($waitAndhold$). Cuando alguien devuelve el libro, se entrega a quien lo reservó. Otros servicios (Figura 2.3) pueden proveer la misma funcionalidad, pero puede tambien ocurrir que finalmente el libro no esté disponible ($unavailable$).

Los objetivos que el cliente quiere lograr son los siguientes:

- Si se realiza un pedido se entrega un libro. Podemos formalizar este objetivo en lógica linear temporal del siguiente modo: $\varphi_1 = \Box(usrReq \implies \Diamond deliver)$
- El cliente no debe utilizar el servicio más de una vez por pedido. Es decir: $\varphi_2 = \Box(queryBook \implies X(\neg queryBook\ W deliver)))$
- Los usuarios no deben recibir libros o mensajes de cancelación si no hicieron un pedido. $(\varphi_4 = \Box((abort \vee deliver) \implies X((\neg abort \wedge \neg deliver)\ W usrReq)))$

Es importante notar que la satisfacibilidad de estos objetivos depende del servicio con el que se esté tratando. Por ejemplo, para el servicio de la Figura 2.1 no hay forma de garantizar los objetivos: para lograr φ_1, si el servicio reporta que el libro no está disponible ($unavailable$), el cliente no puede volver a generar un pedido porque violaría φ_2.

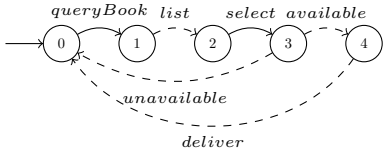

Fig. 2.1: Un servicio de préstamos básico. No existe la posibilidad de esperar a que un libro sea devuelto ($usrReq$ y $abort$ no aparecen ya que son interacciones cliente-usuario).

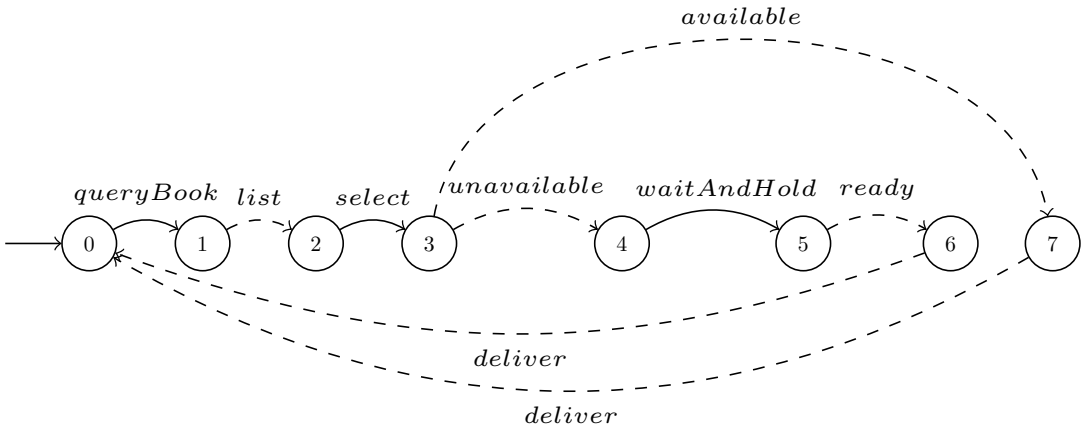

Fig. 2.2: Un servicio de préstamos de libros donde se garantiza que al reservar un libro éste sea entregado.

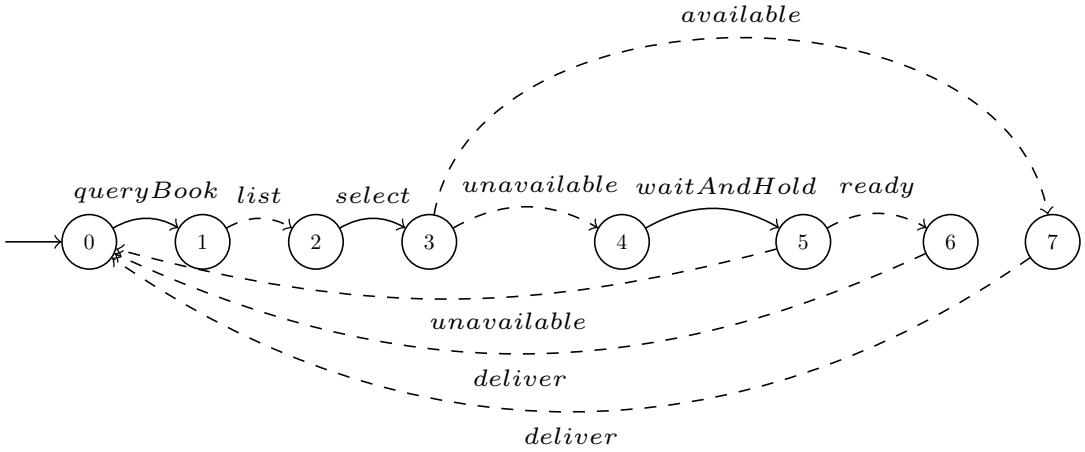

Fig. 2.3: Un servicio de préstamos de libros donde no se garantiza que al reservar un libro éste sea entregado.

Para la Figura 2.2, en cambio, los objetivos son satisfacibles ya que si el libro no se encuentra disponible, se puede utilizar el mecanismo de reserva y espera. Finalmente, en el servicio en la Figura 2.3 no es posible garantizar los objetivos dado que incluso esperando la reserva de un libro es posible que el mismo siga no disponible (*unavailable*).

¿Cómo debería comportarse el cliente para lograr sus objetivos, o concluir que no es posible hacerlo, si conoce la interfaz del servicio al que está conectado pero no su protocolo? El cliente debe interactuar con el servicio y acumular suficiente conocimiento sobre su comportamiento. Durante el proceso de exploración, debido a la falta de conocimiento, es posible que algunas metas sean violadas. En estos casos el cliente tiene que tener la capacidad de reiniciar el servicio para volver a intentar, pero esta vez con más conocimiento. Es por esto que asumimos que existe una acción *reset*! que genera este comportamiento.

Supongamos que el cliente quiere controlar y descubrir el servicio de la Figura 2.1, y que la única información que tiene disponible es la existencia de las siguientes acciones:

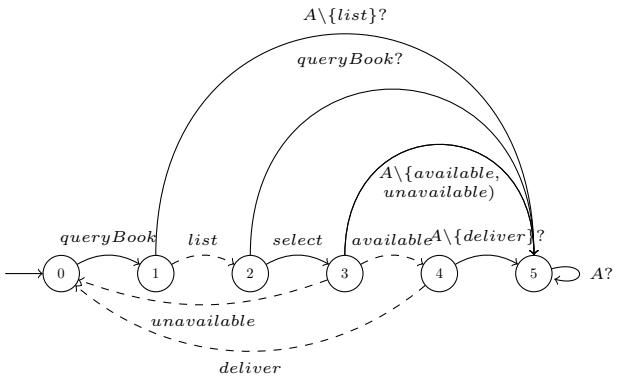

Fig. 2.4: MTS con conocimiento parcial del comportamiento del servicio de préstamo de libros. A es el conjunto de acciones de su interfaz. $A \setminus B$ es resta de conjuntos. Las etiquetas con signos de pregunta son acciones aún no exploradas.

queryBook, *list*, *select*, *available*, *unavailable*, *deliver*, donde *queryBook* y *select* son controlables por el cliente. El procedimiento funciona de la siguiente manera: al recibir el pedido del usuario (*usrReq*) el cliente se encuentra en el estado 0, y debe intentar alguna acción controlable que lo pueda llevar a entregar el libro (*deliver*). Asumamos que intenta seleccionar un libro (*select*) (dado que no conoce el protocolo del servicio, solo las acciones posibles). El servicio rechaza este pedido, porque la acción no está disponible en el estado 0. Si el cliente ahora intenta preguntar por un libro (*queryBook*), el servicio acepta el pedido y se mueve al estado 1. Eventualmente recibe un listado de libros (*list*) que lo lleva al estado 2. Es importante notar que esto puede suceder incluso antes de que el cliente detecte que no es posible ejecutar una acción controlable desde 1. Ahora intenta seleccionar un libro (*select*) y llega al estado 3. En este momento puede suceder que el libro esté disponible (*available*) y entonces sea posible entregarlo (*deliver*), llevando al servicio nuevamente al estado 0. Si ocurre otro pedido del usuario (*usrReq*), el cliente sabe cómo lograr entregar el libro (*deliver*) siempre y cuando observe el mismo comportamiento no controlable (es decir se ejecute la acción *available* en el estado 4). Si en algún momento en el estado 3 se descubre que el libro no está disponible (*unavailable*), entonces el servicio llega al estado 0, donde sabe que es posible hacer otro pedido (*queryBook*) pero no es posible seleccionar un libro (*select*). Es claro que no existe ninguna estrategia desde este punto que garantice φ_1 dado que ninguna acción controlable puede avanzar el servicio y el cliente no tiene garantías de que alguna acción no controlable ocurra. De hecho nosotros sabemos que ninguna acción no controlable ocurrirá (ver Figura 2.1) pero el cliente no lo sabe.

La Figura 2.4 muestra un *Modal Transition System*(MTS) que codifica el conocimiento acumulado por el cliente hasta el momento. Las transiciones con signos de pregunta indican que no se sabe aún si dicho comportamiento está disponible. Los estados 0 a 4 codifican el comportamiento observado hasta el momento y el comportamiento desconocido va al estado 5 que modela la falta total de conocimiento sobre el comportamiento futuro.

Basándose en el MTS, el cliente puede razonar que la razón por la cual llegó a un camino sin salida no significa que es imposible lograr las metas. El camino sin salida puede haber sido producto de ejecutar la acción *select* desde el estado 2. Quizás si hubiera probado otra acción controlable (como *queryBook*) el camino sin salida se habría evitado.

Por lo tanto, el cliente ejecuta *reset*! y comienza de nuevo. Por supuesto, cuando el cliente llega al estado 2 nuevamente descubre que no hay otras acciones controlables aceptadas por el servicio. Más adelante descubre que no hay otras opciones controlables en el estado 0. Concluye entonces que no es posible garantizar los objetivos del sistema y termina, emitiendo un mensaje que denota insatisfacibilidad (al que nos referiremos como *none*!). Notemos que en los estados 1, 3 y 4 saber si hay acciones controlables disponibles es irrelevante porque incluso si las hubiera no hay garantía de que ganarían la condición de carrera contra las acciones no controlables disponibles en esos estados.

Si conectamos el mismo cliente al servicio de la Figura 2.2, éste explora el comportamiento disponible y eventualmente prueba la funcionalidad de reserva. Descubre que esta funcionalidad es crucial para garantizar el envío de los libros y finalmente logra una ejecución que, desde el último *reset*!, garantiza todos los objetivos por siempre. Por otro lado, para el servicio de la Figura 2.3 así como para el más simple, el cliente logra sus metas siempre y cuando no ocurra que un libro no esté disponible (*unavailable*) en los estados 3 o 5. Una vez que ambos ocurren, el cliente reporta que el servicio no garantiza las metas.

Notemos la tensión entre control y relevamiento: si el ambiente deliberadamente o por azar no muestra parte de su comportamiento, no es posible conocerlo completamente. Puede ser que este comportamiento no observado sea necesario para concluir irrealizabilidad de las metas y por ende es posible que permita temporariamente cumplir los objetivos.

¿Cómo se puede automatizar el comportamiento descripto? En esta tesis formalizamos el problema de control y relevamiento descripto arriba y mostramos como resolverlo usando síntesis de controladores para MTS con objetivos SGR(1).

3. PRELIMINARES

Presentamos un breve resumen sobre síntesis de controladores.

Definición 3.0.1. (Labelled Transition Systems) *Un Sistema de transición etiquetado (Labelled Transition System, LTS) es* $W = (S_W, \Sigma_W, \Delta_W, s^0_W)$*, donde* S_W *es un conjunto finito de estados,* $\Sigma_W \subseteq Act$ *es su* alfabeto de comunicación*,* $\Delta_W \subseteq (S_W \times \Sigma_W \times S_W)$ *es una relación de transición, y* $s^0_W \in S_W$ *es el estado inicial. Usamos* $\Delta_W(s)$ *para denotar* $\{s' \mid \exists e.(s, e, s') \in \Delta_W\}$*.*

Una ejecución ε *de* W *es una palabra finita o infinita* $\varepsilon = s^0, e^0, s^1, \ldots \in (S_W \times \Sigma_W)^\omega \cup (S_W \times \Sigma_W)^* \cdot S_W$*, donde* $|\varepsilon|$ *es la cantidad de símbolos de* Σ_W *que aparecen en* ε *y para cada* $i < |\varepsilon|$ *se tiene* $(s^i, e^i, s^{i+1}) \in \Delta_W$*. Una ejecución es* maximal *si es inifinita o termina en un estado* s *tal que* $\Delta_W(s) = \emptyset$*. Una palabra* $\pi \in \Sigma^\omega_W \cup \Sigma^*_W$ *es una* traza *de* W *si hay una ejecución* ε *de* W *tal que* $\varepsilon|_{\Sigma_W} = \pi$*, donde* $\pi|_\Sigma$ *es la proyección de la palabra* π *sobre el alfabeto* Σ*. Una traza es* maximal *si es la proyección de una ejecución maximal.*

Si para algún s*,* s' *y* e *existe* $(s, e, s') \in \Delta_W$ *decimos que* e *está habilitada desde* s*.*

Los sistemas de transición modales (*Modal Transitions Systems (MTS)* [4] son nociones abstractas de LTSs. Extienden a los LTSs mediante la distinción de dos conjuntos de transiciones. Intuitivamente, un MTS describe un conjunto de posibles LTSs definiendo un límite superior y un límite inferior de conjuntos de transiciones que pueden existir en cada estado. Para cada estado el MTS define transiciones requeridas, que tienen que existir, y transiciones posibles, que podrían no existir. Solamente pueden aparecer acciones definidas dentro de estos conjuntos.

Definición 3.0.2. (Sistemas de Transición Modales [4]) *Un* Sistema de transición modal (*Modal Transition System (MTS)) es* $K = (S_K, \Sigma_K, \Delta^r_K, \Delta^p_K, s^0_K)$*, donde* S_K *es un conjunto finito de estados,* Σ_K *es su* alfabeto de comunicación*,* $\Delta^r_K \subseteq \Delta^p_K \subseteq (S_K \times \Sigma_K \times S_K)$ *son las relaciones de transiciones requeridas y posibles respectivamente, y* $s^0_K \in S_K$ *es el estado inicial.*

Notamos con $\Delta^p_K(s)$ el conjunto de acciones posibles permitidas desde s, es decir $\Delta^p_K(s) = \{e \mid \exists s' \cdot (s, e, s') \in \Delta^p_K\}$. Similarmente, $\Delta^r_K(s)$ denota el conjunto de acciones requeridas habilitadas desde s.

Definición 3.0.3. (Refinamiento [4]) *Sean* K *y* M *dos MTS, donde* $K = (S_K, \Sigma_K, \Delta^r_K, \Delta^p_K, s^0_K)$ *y* $M = (S_M, \Sigma_M, \Delta^r_M, \Delta^p_M, s^0_M)$*. La relación* $R \subseteq S_K \times S_M$ *es un* refinamiento *entre* K *y* M *si lo siguiente vale para toda* $e \in \Sigma_K \cup \Sigma_M$ *y todo* $(s, s_M) \in R$*.*

- *Si* $(s_K, e, s'_K) \in \Delta^r_K$ *entonces existe* s'_M *tal que* $(s_M, e, s'_M) \in \Delta^r_M$ *y* $(s'_K, s'_M) \in R$*.*
- *Si* $(s_M, e, s'_M) \in \Delta^p_M$ *entonces existe* s'_K *tal que* $(s_K, e, s'_K) \in \Delta^p_K$ *y* $(s'_K, s'_M) \in R$*.*

Decimos que M *refina* K *si existe una relación de refinamiento* R *entre* K *y* M *tal que* $(s^0_K, s^0_M) \in R$*, y se nota* $K \preceq M$*.*

Intuitivamente, M refina K si cada transición requerida de K existe en M y cada transición posible en M es también posible en K. Un LTS pude verse como un MTS donde $\Delta^p_K = \Delta^r_K$. Por lo tanto, un LTS puede también refinar un MTS. Los LTSs que refinan un MTS K son descripciones completas del comportamiento del sistema y por lo tanto se

llaman *implementaciones* de K. Denotamos con $\mathtt{I}^{det}[K]$ el conjunto de implementaciones determinísticas de K.

Usamos *Fluent Linear Temporal Logic* [8] (FLTL) para definir los objetivos de control. Más específicamente, nos concentramos en fórmulas GR(1) [9], un subconjunto de LTL que admite fórmulas del tipo $\bigwedge_{i=1}^{n} \square\diamondsuit A_i \rightarrow \bigwedge_{j=1}^{m} \square\diamondsuit G_j$ donde A_i y G_j son expresiones *fluent* no temporales. La definición completa se puede leer en [8]. Sea una fórmula FLTL φ y una traza π, π satisface φ se nota $\pi \models \varphi$.

Definimos ahora el problema de síntesis de controladores.

Definición 3.0.4. (Control para LTS [10]) *El problema de control de LTSs es una tupla $\mathcal{K} = \langle W, G, \Sigma^c \rangle$. Donde W es un LTS determinístico que modela el dominio del problema $W = (S_W, \Sigma_W, \Delta_W, s_W^0)$, el alfabeto Σ_W está particionado en acciones controlables y no controlables, es decir $\Sigma = \Sigma^c \uplus \Sigma^u$ y G es una fórmula FLTL.*

Un controlador *es una función $f : \Sigma^* \rightarrow 2^{\Sigma^c}$ que asocia un conjunto de acciones controlables con toda secuencia de acciones.*

Una traza $\sigma \in \Sigma^\omega$ es compatible con el controlador f si para cada $i \geq 0$ se tiene $\sigma_i \in f(\sigma_{0..i-1}) \cup \Sigma^u$.

Una f es una solución para $\mathcal{K}$ si todas las trazas maximales σ de W que son compatibles con f son infinitas y además $\sigma \models G$. Cuando tal controlador existe se dice que el problema es realizable, en caso contrario se dice que es irrealizable.

Determinar si un LTL es realizable es un problema decidible [11] y por lo tanto el control de LTS lo es también. Es posible determinar que un problema de control de LTS es realizable y simultáneamente extraer un controlador C que lo resuelva. Existe síntesis eficiente para diseños reactivos para gran parte de LTL (por ejemplo, GR(1) [5]).

Decidir si un problema de control LTS es realizable se puede traducir a un juego de dos jugadores donde uno de ellos tiene que ganar (el controlador). Usamos el término *estado ganador* para denotar los estados visitados por un controlador que es solución del problema de control. En otras palabras, los estados ganadores son aquellos desde los cuales un controlador puede garantizar los objetivos. Dado que los juegos son determinísticos, todos los estados que no son ganadores son perdedores.

3.1. El problema de control MTS

El problema de síntesis de controladores para MTS consiste en chequear si todas, alguna o ninguna de las implementaciones LTS determinísticas de cierto MTS determinístico pueden ser controladas por un controlador LTS.

Lo definimos formalmente de la siguiente manera.

Definición 3.1.1. (Semánticas del Control de MTS [6]) *Dado un MTS determinístico K, una fórmula FLTL G y un conjunto $\Sigma^c \subseteq \Sigma$ de acciones controlables, resolver el* problema de control para MTS $\mathcal{K} = \langle K, G, \Sigma^c \rangle$ *es responder:*

- ***All**, si para todos los LTSs $W \in \mathcal{I}^{det}[K]$, el problema de control $\langle W, G, \Sigma^c \rangle$ es realizable,*
- ***Some**, si para algún LTS $W \in \mathcal{I}^{det}[K]$, el problema de control $\langle W, G, \Sigma^c \rangle$ es realizable,*
- ***None**, en otro caso.*

Notemos que, como en el caso del problema de control para LTSs, restringimos el problema a modelos determinísticos. Esto surge de que la solución para la realizabilidad

de MTS es una reducción a realizabilidad de LTSs. En esta tesis distinguimos entre dos casos: (a) **None** y (b) **Some** o **All**. Dado que **All** implica **Some** generalmente hablamos de **Some**. La explicación completa respecto de estas tres opciones se puede leer en [6].

El problema de control para MTS se puede resolver reduciéndolo a control para LTSs de la siguiente manera. Construimos un problema de control LTS que codifica si existe una implementación del MTS para la cual hay un controlador. Esto se logra modelando un problema de control LTS en donde el controlador puede elegir la implementación más "fácil" de controlar. Formalmente, definimos el LTS K^I que tiene transiciones adicionales que modelan explícitamente cuáles transiciones del MTS están disponibles.

Definición 3.1.2. *Dado un MTS* $K = (S_K, \Sigma_K, \Delta_K^r, \Delta_K^r, s_K^0)$, *definimos* $K^I = (S_{K^I}, \Sigma_{K^I}, \Delta_{K^I}, s_K^0)$ *de la siguiente manera:*

$$
\begin{aligned}
S_{K^I} &= S \cup \{(s,i) \mid s \in S,\ i \subseteq \Sigma \text{ and } \Delta^r(s) \subseteq i\} \\
\Sigma_{K^I} &= \Sigma \cup \overline{\Sigma}, \text{ donde } \overline{\Sigma} = \{e_i \mid i \subseteq \Sigma\} \\
\Delta_{K^I} &= \{(s, e_i, (s,i)) \mid s \in S \text{ and } \Delta^r(s) \subseteq i \subseteq \Delta^p(s)\} \\
&\cup \{((s,i), e, s') \mid (s, e, s') \in \Delta^p \text{ and } e \in i\}
\end{aligned}
$$

Los estados en K^I son de dos tipos. Aquellos que son de la forma s con $s \in S$ codifican estados donde es posible elegir cuál subconjunto de posibles transiciones se quiere implementar. Elegir un subconjunto $i \subseteq \Sigma$, lleva a un estado (s, i). Los estados de la forma (s, i) tienen todas las transiciones salientes listadas en el conjunto i. Una transición desde (s, i) con la etiqueta $e \in i$ lleva al mismo estado s' en K^I que tomar e desde s en K.

El algoritmo en el Listing 3.1 computa la solución para el problema de control de MTS. Para una descripción más detallada se puede leer [6].

```
1  procedure resolverMTS(K, G, Σ^c)
2     K_S^I = ⟨K^I, X_Σ̄(G), Σ^c ∪ Σ̄⟩
3     if (esLTSRealizable(K_S^I))
4        return Some
5     else
6        return None
```

Listing 3.1: Solución para la Definición 3.1.1

4. EL PROBLEMA DE CONTROL Y EXPLORACIÓN

En esta sección describimos y formalizamos el Problema de Control y Exploración. El Problema de Control y Exploración (C&D por sus siglas en inglés) asume que existe un MTS K que representa el conocimiento parcial inicial del ambiente a controlar. Además, asume que el comportamiento real del ambiente (que en este momento es desconocido para el controlador) puede ser descripto con un LTS W (el mundo) y que el conocimiento inicial disponible es consistente con el ambiente. Formalmente esto significa que tienen el mismo alfabeto ($\Sigma = \Sigma_K = \Sigma_W$) y que el comportamiento del ambiente es un refinamiento del conocimiento inicial: ($K \preceq W$). Es importante notar que K puede modelar el total desconocimiento del comportamiento del ambiente, a excepción de su interfaz (acciones controlables y no controlables). En la Figura 2.4 podemos ver conocimiento parcial, mientras que en la Figura 5.2 podemos ver el conocimiento menos refinado que podemos representar. El problema consiste entonces en encontrar una estrategia que controle el ambiente eligiendo cuáles acciones controlables de K habilitar.

Asumimos que es posible reiniciar el ambiente ($reset!$), es decir retornar el estado actual del W a su estado inicial (s^0_W). Finalmente, asumimos que existe una acción controlable ($none!$) que el controlador usa para reportar que no es posible realizar los objetivos desde el estado actual. Denotamos el conjunto completo de eventos del problema C&D como $\Sigma_! = \Sigma \uplus \{reset!, none!\}$ y sus eventos controlables como $\Sigma^c_! = \Sigma^c \uplus \{reset!, none!\}$.

El problema C&D para un objetivo FLTL G es el de encontrar un controlador (f: $\Sigma^*_! \to \mathcal{P}(\Sigma^c_!)$) que, basándose en el comportamiento observado hasta el momento, habilite las acciones controlables de manera tal que concluya insatisfacibilidad o eventualmente satisfaga el G desde el último $reset!$.

Definición 4.0.1. (Problema de Control y Exploración)) *Sea K un MTS determinístico con su alfabeto particionado en acciones controlables y no controlables, es decir $\Sigma = \Sigma^c \uplus \Sigma^u$, $\Sigma_! = \Sigma \uplus \{reset!, none!\}$, $\Sigma^c_! = \Sigma^c \uplus \{reset!, none!\}$ y G una fórmula FLTL. Definimos el problema de Control y Exploración C&D como una tupla $\mathcal{K} = \langle K, G, \Sigma^c_! \rangle$.*

Un controlador *es una función $f : \Sigma^*_! \to \mathcal{P}(\Sigma^c_!)$ que asocia a toda secuencia de acciones un conjunto de acciones controlables.*

Una traza $\sigma \in \Sigma^\omega_!$ es compatible con un controlador f si para cada $i \geq 0$ se tiene que: si $f(\sigma_{0..i-1}) \in \{reset!, none!\}$ entonces $\sigma_i = f(\sigma_{0..i-1})$, y si no $\sigma_i \in f(\sigma_{0..i-1}) \cup \Sigma^u$.

Para simplificar algunas de las notaciones asumimos que $f(\epsilon) = \{reset!\}$.

Un controlador f es una solución *para $\mathcal{K}$ si para todo $W \in \mathcal{I}^{det}[K]$ y para cada traza $\sigma \in \Sigma^\omega_!$ que es compatible con f tal que cada subsecuencia maximal sin símbolos $reset!$ en σ, proyectada sobre Σ es una (finita o infinita) traza de W, se tiene:*

1. *$\sigma \models \Diamond\, none!$ implica $\langle W, G, \Sigma^c \rangle$ es irrealizable, y*
2. *$\sigma \models \Diamond\, none! \vee \Diamond(reset! \wedge \mathrm{X}(G \wedge \Box\, \neg(reset! \vee none!))$*

De acuerdo a esta definición, utilizando una solución f en un ambiente en el cual es posible garantizar G, eventualmente se descubre suficiente comportamiento del ambiente para garantizar G. Más precisamente, a partir de cierto momento no ocurren acciones $reset!$ y se garantiza G.

Por otra parte, si a la misma solución f se la utiliza en un ambiente en el cual es imposible garantizar G, entonces eventualmente se descubre suficiente comportamiento

para reportar que G no puede ser garantizado (retornando *none*!), o, a partir de cierto momento no ocurren acciones *reset*! y se garantiza G.

Recordemos (c.f., Sección 2) que el último caso corresponde a aquel en el cual el ambiente no ejecuta su mejor estrategia (es decir no selecciona las acciones que forzarían una violación de G) o no muestra suficiente comportamiento para poder concluir insatisfacibilidad.

5. SOLUCIÓN AL PROBLEMA DE CONTROL Y EXPLORACIÓN

En esta sección presentamos un algoritmo que resuelve el problema C&D definido anteriormente. En otras palabras, dado un conocimiento inicial K, W un refinamiento de K y un objetivo G, el algoritmo genera un controlador para el problema C&D.

Restringimos el mundo de objetivos posibles al de objetivos SGR(1) [9]. Para este tipo de objetivos, asumiendo implementaciones determinísticas, existe una solución polinomial al problema de control de MTS [6]. Además, es posible chequear si un objetivo SGR(1) puede ser logrado desde un estado no inicial del MTS sin aumentar la complejidad temporal. Usamos este dato para facilitar la implementación. Estas técnicas pueden generalizarse a FLTL completo teniendo en cuenta la información necesaria para chequear la fórmula entera desde los estados no iniciales.

El algoritmo interactúa con el ambiente mediante una API `env` que provee los siguientes métodos:

- `getCurrentStateID`, que retorna un identificador de estado de un conjunto finito de posibles identificadores (recordemos que asumimos que el comportamiento del ambiente puede ser caracterizado mediante un LTS finito)

- `reset`, que envía el ambiente y el conocimiento hacia el estado inicial. Esto no implica eliminar lo aprendido, si no simplemente mover el estado actual al inicial.

- `try(C)`, que pide la ejecución de una de las acciones controlables en el conjunto `C` (que puede ser vacío) y retorna la acción que efectivamente se ejecutó. Es posible que la acción ejecutada no pertenezca a `C` si ocurrió una acción no controlable, es decir una condición de carrera. Es posible también que el método devuelva `null` si ninguna de las acciones de `C` estaban disponibles en el ambiente, y tampoco había ninguna acción no controlable disponible. Por simplicidad asumimos que la ejecución del algoritmo es lo suficientemente rápida para que como mucho ocurra una sola acción no controlable entre cada llamado a `try`. Sin este supuesto simplemente habría que manejar colas de acciones.

Si el W es efectivamente un LTS que refina el conocimiento inicial capturado por el MTS K entonces el algoritmo fuerza al ambiente a mostrar una traza que sea consistente con el controlador que resuelve el problema C&D determinado por la tupla $\langle K, G, \Sigma^c_! \rangle$.

Introducimos métodos adicionales para hacer más legible el algoritmo. Dado un MTS $K = (S_K, \Sigma_K, \Delta^r_K, \Delta^p_K, s^0_K)$ definimos el método `getController`(s) que resuelve el problema de control que aparece en el Algoritmo 3.1: Dado un estado $s \in S_K$, retorna un controlador para $\langle K^I, \mathcal{X}_{\overline{\Sigma}}(G), \Sigma^c \cup \overline{\Sigma} \rangle$ inicializado en el estado s, o, si no existe dicho controlador, retorna **None**. Notemos que inicializar en el estado s no sólo requiere cambiar el estado inicial de K^I si no también actualizar los valores iniciales de los fluents de G de acuerdo a la traza que llevó al estado s.

Asumimos que los métodos `solveMTS` y `getController` no recomputan el controlador a menos que el conocimiento haya cambiado. Esto nos ofrece consistencia y eficiencia. Mientras el conocimiento no se actualice, se usa un solo controlador, asegurándose de que si se usa ese controlador continuamente, los objetivos se satisfacen.

Definimos el método `copyState(`s`)`, que crea una copia del estado s con las mismas transiciones *salientes* y estados destino. Notemos que esta operación produce un MTS equivalente, es decir no cambia su conjunto de implementaciones.

Definimos el método `removeTransitions(`s, A`)` que, dado un estado s y un conjunto de acciones A, remueve toda las transiciones desde el estado s cuyas acciones pertenezcan a A. Definimos también un método `makeRequired(`s, e, s'`)` que, dada la transición (s, e, s'), la convierte en requerida. Finalmente, los métodos `getInitialState`, `set`, `InitialState`, `existsStateWithID`, y `getStateByID` funcionan de la manera esperada, y asumimos que los estados de K tienen métodos que obtienen y establecen sus identificadores.

En la Figura 5.1 se detalla el algoritmo que resuelve el problema C&D. El procedimiento principal `Control&Discover` asume que conoce el objetivo G, la API `env`, y el conjunto de acciones controlables Σ^c. Como precondición se pide que W refine a K.

Para poder explicarlo en detalle, usamos un ejemplo.

Supongamos que tenemos un mundo W como el de la Figura 5.1. Podemos pensarlo como un robot que tiene que explorar un cuarto para salir, pero que no controla la apertura de la puerta. Si asumimos que no tenemos ningún conocimiento inicial, nuestro modelo tiene la forma de la Figura 5.2. El objetivo, definido de manera informal, consiste en ejecutar la acción *salir* infinitas veces. La única acción no controlable es *puerta*.

Al comenzar la ejecución, copiamos el estado actual de K y todas sus transiciones. El modelo original no se modifica durante toda la ejecución (en este caso el estado 0), si no que se agregan estados. Al copiar el estado le asignamos un nuevo ID (en este caso 1) y lo asociamos con el del estado actual del mundo. Sabemos entonces que el estado 1 de K corresponde al estado 0 de W y lo marcamos como el estado inicial de K. La Figura 5.3 muestra K luego de ejecutar la línea 5.

La respuesta al problema de control desde el estado inicial es **Some**. Esto nos lleva a la línea 8 donde C^S es un controlador para el estado actual de K. El controlador se usa para computar el conjunto de acciones controlables A^c que garantizan acercar a cumplir el objetivo G para alguna implementación que refina a K. Desde el estado actual la respuesta también es **Some** lo que nos lleva a la línea 14. El controlador quiere ejecutar *salir* (dado que esa es su meta), entonces $A^c = \{salir\}$.

Luego se llama al método `try` del ambiente `env`, para intentar ejecutar alguna acción controlable del conjunto A^c. El método `try` retorna en e la acción que efectivamente se ejecutó, y se comporta de la siguiente manera:

- Si la acción es `null`, entonces quiere decir que desde el estado actual no existe ninguna acción de A^c y tampoco ninguna acción no controlable. Esto es porque asumimos que si se le da la oportunidad al ambiente, éste la toma y ejecuta una acción no controlable. Se actualiza K eliminando dichas acciones de s^c_K en la línea 17.

- Si la acción no es `null`, entonces se actualiza K llamando al método `updateKnowledge` (líne 19) informando e y s^c_K. Luego el algoritmo imprime la acción ejecutada e.

Por el momento no nos interesa cómo el ambiente elige la acción que efectivamente se ejecuta, y cómo esta elección puede afectar la resolución del problema. La respuesta a esta pregunta se puede ver en la Sección 6.

Si miramos el estado 0 de W, podemos ver que no hay acciones no controlables, y no está disponible *salir*. Entonces $e =$ `null`, y se eliminan de K las acciones A^c y las acciones no controlables.. La Figura 5.4 representa K luego de ejecutar la línea 17.

```
1  procedure Control&Discover(K):
2          output("reset!")
3          s^c_K = K.copyState(K.getInitialState)
4          s^c_K.setID(env.getCurrentStateID)
5          K.setInitialState(s^c_K)
6
7          while (solveMTS(K.getInitialState)≠ None)
8                  C^S = getController(s^c_K)
9                  if (C^S == null)
10                         env.reset
11                         s^c_K = K.getInitialState
12                         output("reset!")
13                 else
14                         A^c = enabledControllableActions(C^S)
15                         e = env.try(A^c)
16                         if (e == null)
17                                 K.removeTransitions(s^c_K, A^c ∪ Σ^u)
18                         else
19                                 updateKnowledge(K, e, s^c_K)
20                                 output(e)
21                         endif
22                 endif
23         endwhile
24         output("none!")
25 endprocedure
26
27 procedure updateKnowledge(K, e, s^c_K)
28       if (K.existsStateWithID(env.getCurrentState))
29          s'=K.getStateByID(env.getCurrentState)
30       else
31          s' = K.copyState(Δ_K(s^c_K, e))
32          s'.setID(env.getCurrentState)
33       endif
34       K.removeTransitions(s^c_K, {e})
35       K.makeRequired(s^c_K, e, s')
36       s^c_K = s'
37 endprocedure
```

Listing 5.1: Algoritmo que resuelve el problema C&D

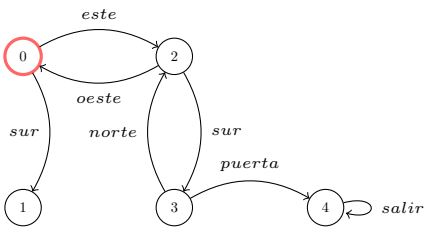

Fig. 5.1: Mundo

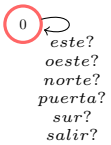

Fig. 5.2: Modelo

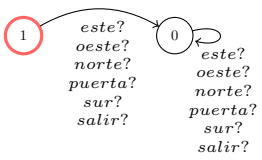

Fig. 5.3: Modelo al iniciar la exploración

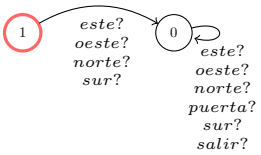

Fig. 5.4: No se ejecutó ninguna acción. Se eliminan *salir* y *puerta* del estado 1

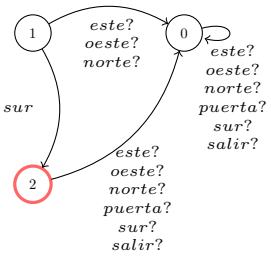

Fig. 5.5: Se ejecutó *sur*

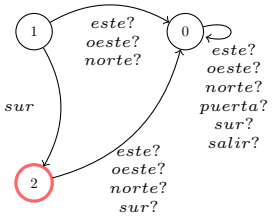

Fig. 5.6: Se eliminan *salir* y *puerta* del estado 2

El procedimiento se repite, y esta vez el controlador elige el conjunto de acciones controlables, dado que no sabe cuál le conviene para alcanzar el objetivo. En este caso se ejecuta la acción *sur*. Esto nos lleva a la línea 19.

El método `updateKnowledge` recibe como parámetros la acción ejecutada e y el estado actual s_K^c. Para actualizar correctamente el conocimiento, necesitamos primero saber si s_K^c ya había sido visitado. Si no había sido visitado, se copia el estado al que llevaba e desde s_K^c en K, y se lo asocia al estado actual de W. Si había sido visitado, simplemente se usa s_K^c. Una vez seleccionado el estado se marca e como acción requerida (es seguro que existe porque fue ejecutada). Finalmente, se actualiza el estado actual. Este método no necesariamente *refina* el conocimiento. Es decir, el conjunto de implementaciones del conocimiento actualizado no necesariamente es un subconjunto de las implementaciones del conocimiento sin actualizar. Podría pasar por ejemplo que e ya fuera una acción requerida, tanto porque así lo marcaba el conocimiento inicial, o porque ya había sido ejecutada. Es seguro sin embargo que W sigue siendo una implementación de K.

En el ejemplo es la primera vez que se visita el estado 1 de W, por ende se genera una copia del estado al que llegábamos en K (0), y se le da un nuevo ID. Ahora sabemos que en K desde 1 llegamos a 2 a través de *sur*, por lo que eliminamos la transición $(1, sur?, 0)$, y agregamos la transición $(1, sur, 2)$. Además, cambiamos el estado actual de K a 2. En la Figura 5.5 se puede ver el resultado de todos estos pasos.

En el siguiente paso se repite la situación de la Figura 5.4. Se intenta ejecutar *salir*, se descubre que no existe y se eliminan *salir* y *puerta* del estado 2. El resultado se ve en la Figura 5.6.

Ahora el controlador elige $\{este, oeste, norte, sur\}$, dado que no sabe cuál le conviene para lograr los objetivos. Nuevamente $e =$ `null`, pero ahora se eliminan todas las acciones del estado 2. El resultado de estos pasos se ve en la Figura 5.7. La respuesta al problema de control desde el estado inicial sigue siendo **Some**, pero desde el estado inicial es **None**, es decir la línea 9 evalúa verdadero, y se reinician tanto W como K (Figura 5.8).

Ahora el controlador aprendió que ejecutar *sur* desde el estado inicial lo lleva a un estado perdedor. En este caso entonces $A^c = \{este, oeste, norte\}$. La acción que ocurre es $e = este$, que lleva a un nuevo estado en W y por ende a la creación de un nuevo estado en

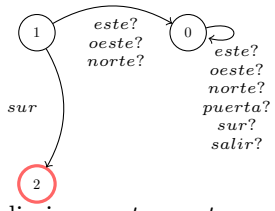

Fig. 5.7: Se eliminan *este*, *oeste*, *norte* y *sur* del estado 2

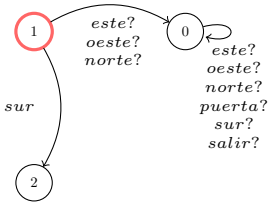

Fig. 5.8: Se reinician K y W

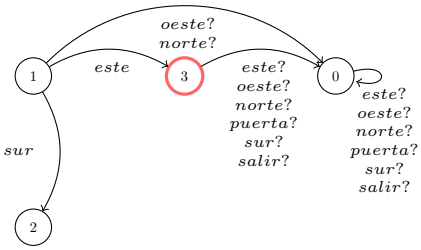

Fig. 5.9: Se ejecuta *este*

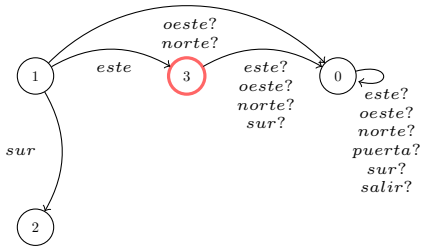

Fig. 5.10: Se eliminan *salir* y *puerta*

K (Figura 5.9). Nuevamente el controlador intenta ejecutar *salir* y nuevamente descubre que no existe en el estado actual (Figura 5.10).

El controlador elige $A^c = \{este, oeste, norte, sur\}$, y esta vez se ejecuta *oeste* (Figura 5.11). Es importante notar que podemos detectar el ciclo gracias a que asumimos que el ambiente es capaz de identificar sus estados y responder con el identificador del estado actual. Si solamente supiéramos las acciones habilitadas, no tendríamos forma de saber que efectivamente al ejecutar *oeste* volvemos a un estado previamente visitado.

El controlador ahora elige explorar el resto de las acciones controlables $\{oeste, norte\}$ y descubre que no existen en el estado 1 (Figura 5.12). Para poder seguir explorando hay que llegar al estado 3, entonces $A^c = \{este\}$ (Figura 5.13). Una vez en el estado 3, $A^c = \{este, norte, sur\}$, y la acción que efectivamente ocurre es *sur* (Figura 5.14).

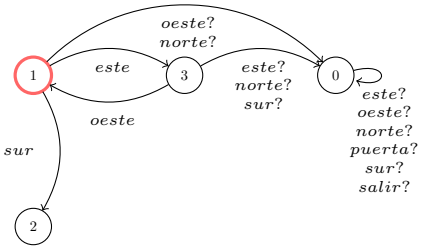

Fig. 5.11: Se ejecuta *oeste*

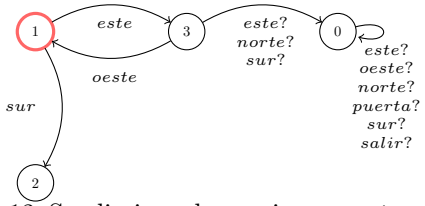

Fig. 5.12: Se eliminan las acciones *oeste* y *norte* del estado 1

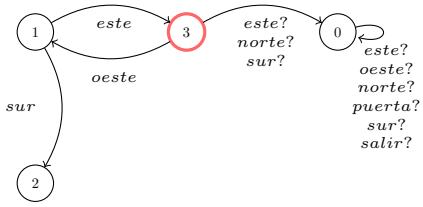

Fig. 5.13: Se ejecuta *este*

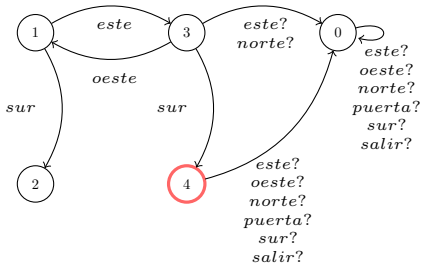

Fig. 5.14: Se ejecuta *sur*

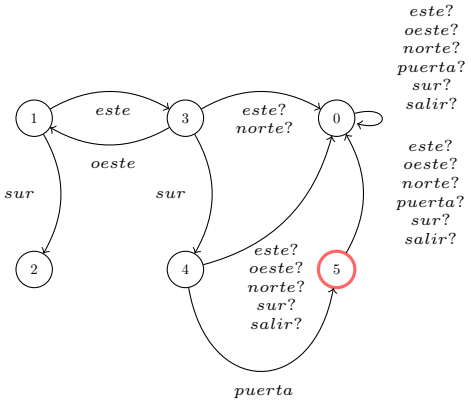

Fig. 5.15: se ejecuta *puerta*

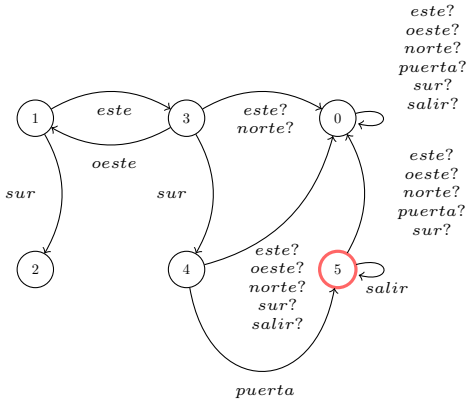

Fig. 5.16: Se ejecuta la acción *salir*

Como vimos previamente, el controlador selecciona *salir* porque es la acción que le permite cumplir el objetivo. En este caso, sin embargo, como existe una acción no controlable en el estado 3 de W, que corresponde al 4 de K, y salir no está habilitada, ocurre *puerta* (Figura 5.15). El controlador no aprende nada respecto de *salir*, dado que no puede distinguir entre la situación en la que ambas acciones están habilitadas pero ocurre *puerta*, y la situación en que *salir* no existe y por ende *puerta* debe ocurrir. Finalmente, el controlador intenta ejecutar *salir* y esta vez la acción existe (Figura 5.16). El algoritmo ejecutará infinitamente la acción *salir*, garantizando de este modo el objetivo.

Elección de acciones controlables

El método `enabledControllableActions` en la línea 14, encapsula la elección de acciones controlables a retornar en la línea 15. Estas son las acciones que el controlador quiere intentar ejecutar, como mencionamos informalmente en el ejemplo. Esta selección es importante, por lo que a continuación explicamos cómo se extrae este conjunto de acciones de C^S.

Sea $\mathcal{K} = \langle K, G, \Sigma^c \rangle$ el problema de control MTS, el LTS derivado $K^I = (S_{K^I}, \Sigma_{K^I}, \Delta_{K^I}, s^0_K)$, y el problema de control *some* $\mathcal{K}^I_S$. Como vimos en la definición 1.3.6 los estados de K^I son los estados s de K, y los estados de la forma (s, i), donde s es un estado de K y i es un conjunto de acciones. Desde cada estado de K^I con la forma s, para cada $\Delta^r_K(s) \subseteq i \subseteq \Delta^p_K(s)$ hay una transición al estado (s, i). Desde cada estado de K^I con la forma (s, i) hay transiciones al estado t tales que $l \in i$ y $(s, l, t) \in \Delta^p_K$.

La solución al problema de control $\mathcal{K}^I_S$ clasifica los estados de K^I como ganadores o perdedores. Un estado s es ganador si para algún conjunto de acciones i el estado (s, i)

es ganador. Dada la estructura de las transiciones de $\mathcal{K}_S^I$, existe un conjunto de sucesores de s que también son ganadores. Existe entonces un conjunto *maximal* único de acciones llamado i tal que todos los sucesores alcanzados por las transiciones $l \in i$ son ganadores. En nuestra implementación, el método `enabledControllableActions` retorna el conjunto i de acciones controlables que, desde el estado actual, llevan a estados ganadores. Este conjunto se llama A^c.

Es importante notar que cualquier conjunto no vacío de este conjunto maximal i sería adecuado para este propósito. Por ejemplo, se podría elegir explorar más o menos incluyendo más o menos acciones posibles (que no se sabe si son requeridas).

Demostración

En esta sección enunciamos y demostramos la corrección del Algoritmo 5.1.

Teorema 5.0.1. *El algoritmo* `control` *genera un controlador que resuelve el problema C&D para* $\mathcal{K} = \langle K, G, \Sigma_!^c \rangle$.

La demostración depende del siguiente lema.

Lema 5.0.2. *El estado actual de K siempre está asociado al ID del estado actual del ambiente W.*

Demostración. Esto vale inicialmente en la línea 5 al crear una copia del estado inicial de K y asociar el ID del estado inicial de W. Siempre que en W hay una transición no nula, este invariante se restablece obteniendo el estado de K que tiene asociado el identificador del estado actual de W, o creando un estado que lo cumpla. □

Lema 5.0.3. *Si inicialmente W es una implementación de K, entonces después de cada actualización lo sigue siendo.*

Demostración. Es posible demostrar algo más fuerte: El estado actual de W refina el estado actual de K, y para cada par de estados t de W y s de K tales que getID(t) esté asociado con getID(s), t refina s. Este invariante vale cada vez que se evalúa la condición del bucle.

Probamos este enunciado mediante inducción en el progreso del algoritmo. Inicialmente, asumimos que W refina a K, por lo que el estado inicial de W refina al estado inicial de K.

En la linea 5 se crea una copia del estado inicial de K y se le asocia identificador del estado inicial de W. Entonces el lema vale la primera vez que se evalúa la condición del ciclo. Suponemos por inducción que W refina a K, que el estado actual t de W refina el estado s de K, y para cada par t' y s' tales que t' tiene asociado el identificador de s', t' refina a s'. Hay tres posibles opciones:

- C^S es **None**
- C^S no es **None**, y la acción de la línea 15 es `null`
- C^S no es **None**, y la acción de la línea 15 no es `null`

Si C^S es **None**, entonces no se actualiza K, por ende tanto W y K retornan sus estados iniciales, que, por inducción, están en relación de refinamiento.

Supongamos que C^S no es **None**. Si el conjunto de acciones retornado en la línea 15 es `null`, entonces tanto W como K permanecen en el mismo estado. Se sabe que cada posible transición que se elimina de K no existe en el ambiente, por ende se mantiene la relación de refinamiento.

En el último caso, sea e la acción retornada en la línea 15, sea t el estado de W antes de tomar la acción e, y sea t' el estado de W luego de ejecutar la acción. Por hipótesis t refina a s^c_K. El procedimiento updateKnowledge restablece el invariante de la siguiente manera;

Si t' ya fue visitado, existe un estado s' de K tal que t' refina a s'. Entonces se toma la primera rama del `if` en `updateKnowledge`.

Marcamos la transición (s^c_K, e, s') como requerida en K. La transición (t, e, t'), como todas las acciones de W, es requerida y por ende también es posible. La transición (s^c_K, e, s') también es posible en K, cumpliendo con el requerimiento de la relación de refinamiento. La transición (s^c_K, e, s') es requerida en K, y (t, e, t') es requerida en W, cumpliendo nuevamente con la condición de la relación de refinamiento. Se puede ver entonces que este cambio reestablece la relación de refinamiento.

Si, en cambio, t' no fue visitado previamente, no existe un estado de K que tenga asociado el ID de t'. Entonces se toma la segunda rama del `if` en updateKnowledge. Como (t, e, t') es una acción requerida de W (y por ende también es posible), y por inducción t refina a s^c_K, existe un estado s' tal que (s^c_K, e, s') es una transición posible en K y t' refina a s'. Creamos una copia de s' y le asociamos el ID de t'. Llamemos s'' a dicha copia. Al ser una copia, es claro que t' refina a s''. Marcamos ahora la acción (s^c_K, e, s'') como requerida en K. Para restablecer el refinamiento, tenemos que mostrar que existe una transición correspondiente en W. En este caso esa acción es (t, e, t'), que es requerida en W.

□

Podemos ahora probar el Teorema 5.0.1.

Demostración. Cada secuencia de acciones tomada entre los eventos *reset*! o *none*! es una secuencia de acciones tomada por W. Podemos decir entonces que las subsecuencias maximales sin eventos *reset*! son trazas de W.

Mostramos que el algoritmo imprime *none*! solamente cuando $\langle W, G, \Sigma^c \rangle$ es irrealizable. El único caso en donde el algoritmo imprime *none*! es después de haber concluido que el resultado del problema de control para MTS $\langle K, G, \Sigma^c \rangle$ es **None**. Por la solidez del control para MTS se deduce que en toda implementación de K es imposible cumplir el objetivo G. Por el Lema 5.0.3 W implementa K lo que implica que $\langle W, G, \Sigma^c \rangle$ es irrealizable. Finalmente, mostramos que la traza satisface eventualmente *none*! o, luego del último *reset*!, eventualmente satisface el objetivo.

Supongamos que el algoritmo no imprime *none*!. Tenemos que mostrar que existe un *reset*! final y que luego de ese *reset*! la traza satisface G.

Se dice que un estado s de K fue visitado si en algún momento se lo eligió como s^c_K. Un estado puede ser marcado como perdedor una sola vez. Todo estado de K tiene un ID y dos estados distintos no pueden tener el mismo ID. Por ende, esto puede suceder una finita cantidad de veces. Esto es porque el controlador de $\langle K^I, \mathcal{X}_{\overline{\Sigma}}(G), \Sigma^c \cup \overline{\Sigma} \rangle$ nunca elige acciones que sabe que llevan a estados perdedores. Volver a visitar un estado perdedor s

de K puede suceder solamente al ejecutar un env.try (línea 15) y descubrir que el estado al que lleva la acción e tiene un ID que ya ha sido asignado a s (línea 29) y que es perdedor.

Dado que hay una relación de refinamiento entre K y W que es consistente con los IDs asignados hasta el momento, debe haber al menos una transición posible etiquetada e desde el estado s^c_K. Notemos que al ejecutar la línea 15 el estado actual (s^c_K) de K no era perdedor, lo que implica que (a) todas las transiciones no controlables llevan a estados ganadores y (b) todas las transiciones controlables requeridas llevan a estados ganadores. Entonces una de las dos siguientes opciones tiene que haber sucedido: (a) s^c_K tiene una transición no controlable posible (pero no requerida) al estado perdedor s y al ejecutar env.try se descubre que es requerida o (b) s^c_K tiene una transición posible o requerida (controlable o no controlable) a otro estado s' que aún no fue visitado. Sin embargo, al tomar la acción e aprendemos que el estado s' era en realidad s. En ambos casos se explora una transición nueva, lo cual puede suceder una cantidad finita de veces dado que hay finita cantidad de transiciones. Podemos concluir entonces que hay una cantidad finita de eventos *reset*!. Por razones similares, luego del último *reset*! la cantidad de actualizaciones de K es finita. Concluimos que se ejecuta un sufijo infinito de acuerdo a un controlador generado a partir de K. Además, este controlador garantiza el objetivo de acuerdo a $\langle K^I, \mathcal{X}_{\overline{\Sigma}}(G), \Sigma^c \cup \overline{\Sigma}\rangle$. Se puede ver entonces que el sufijo satisface G desde el último *reset*!.

□

6. VALIDACIÓN

El objetivo de esta sección es explorar la factibilidad y en particular el comportamiento que el C&D exhibe para distintos casos de estudio y como varía dependiendo de dos factores: la satisfacibilidad de los objetivos en el ambiente a controlar y descubrir, y el grado en que el ambiente exhibe comportamiento desconocido para el controlador, facilitando o impidiendo el descubrimiento de comportamiento. Usamos siete casos de la literatura en dos versiones distintas (realizable e irrealizable) cada uno con tres tipos distintos de ambiente y corremos el C&D siete veces. Cada caso de estudio incluye un modelo de comportamiento del ambiente y de las metas a lograr desarrollado por un tercero. Tratamos a este modelo como el W desconocido a ser descubierto y controlado. Nosotros definimos un conocimiento inicial básico K, Luego mostramos cómo el algoritmo controla el ambiente desconocido y logra los objetivos ó descubre suficiente comportamiento como para declarar que el ambiente no puede ser controlado.

6.1. Sujetos

Todos los casos de estudio se corren usando una extensión de la herramienta MTSA, que admite especificaciones de LTS y propiedades usando una notación basada en álgebras de proceso (FSP) y lógica temporal. MTSA también admite la síntesis de controladores para problemas de control SGR(1). La extensión consiste en aceptar la definición y resolución de problemas C&D, y también en ejecutar soluciones las soluciones contra los ambientes dados. La versión extendida de la herramienta y los casos de estudio se pueden encontrar en [12].

Para evitar sesgos y manipulación manual, usamos casos de estudio donde los ambientes a controlar están descritos como LTS y las metas como fórmulas de lógica temporal.

Usamos el caso de estudio de [13] tomado del Programa Europeo de Monitoreo Global para Ambiente y Seguridad (***GMES*** por sus siglas en inglés), un servicio de supervisión de eventos catastróficos como inundaciones, terremotos e incendios. En este escenario debemos coordinar servicios entre dos países distintos. En uno de estos países existe un centro de Control y Comando a cargo del monitoreo de bosques y supervisión de incendios forestales. El otro país brinda servicios de apoyo como UAV (vehículos aéreos no tripulados) y servicios de clima.

Usamos la especificación del caso de estudio ***Celda de Producción*** en [14], que se presentó originalmente en [15] y estudiado extensivamente: un brazo robótico coordina la aplicación de varias herramientas para construir un producto, cumpliendo con ciertos requerimientos de seguridad y *liveness*. Cuando hablamos de propiedades de *liveness* nos referimos a propiedades cuyos contraejemplos son infinitos. En general describen el progreso del sistema. En este caso, por ejemplo, el requerimiento de *liveness* consiste en construir infinita cantidad de productos. Se asume que si el controlador espera recibir nuevas piezas para construir un producto, finalmente las recibe. Las reglas de seguridad describen cómo y cuándo las herramientas pueden ser utilizadas y los requerimientos deben cumplirse antes de entregar productos a la cinta mecánica saliente. La especificación asume que las herramientas pueden fallar, sin embargo también asume que estas fallas son de naturaleza probabilística y por ende reintentar suficientes veces lleva finalmente a usarlas exitosamen-

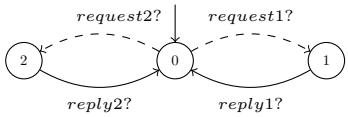

Fig. 6.1: MTS para patrones de pedido y respuesta.

te. El problema consiste en sintetizar un controlador para el brazo robótico que satisfaga la especificación basándose en el conocimiento de las interfaces de las herramientas.

Otro de nuestros sujetos es el caso de estudio ***Búsqueda y rescate*** presentado en [16]. En este caso un robot debe explorar una casa derrumbada y llevar suministros a personas atrapadas en uno de los cuartos. En esta situación una cantidad de obstáculos pueden intermitentemente impedir el movimiento del robot. El controlador debe ser sintetizado automáticamente basándose en la interfaz del robot y los requerimientos de la misión.

Adaptamos también el caso de estudio de ***Compra y Entrega*** de [17], que implica sintetizar un controlador para componer y monitorear servicios web distribuidos. También usamos la ***Librería*** de [18] que tiene similitudes con el de Compra y Entrega, dado que coordina varios servicios para proveer compra y entrega de libros que se encuentren disponibles. Usamos el caso de estudio de ***Mediación de Servicios*** de [19], que consiste en mediar entre dos servicios con distintas interfaces, y un caso (***Agencia de Viajes***) donde es necesario orquestar distintos servicios de reservas para ofrecer paquetes de vacaciones a los clientes. Para todos los casos mencionados usamos la formalización publicada en [14].

Todos los casos mencionados anteriormente tienen solución. Dada la falta de casos publicados que fueran irrealizables y expresados como LTS + LTL, modificamos manualmente cada uno de los mencionados para generar versiones irrealizables. Para el caso de GMES y el de Celda de Producción, permitimos fallas de las cuales es imposible recuperarse; en el primero las fallas provienen del aterrizaje de los UAVS, en el segundo de las herramientas. En el caso de Búsqueda y Rescate removemos transiciones en las que el ambiente permite al robot abrir la puerta una vez que tomó el paquete (pero aún no habiéndolo entregado), impidiendo cumplir los objetivos. Para el de Mediación de Servicios consideramos un escenario donde se termina el tiempo de espera para un servicio, y como consecuencia retorna un resultado no esperado. Para la Agencia de Viajes y la Librería eliminamos una suposición sobre la terminación exitosa de los servicios, y para el de Compra y Entrega agregamos la posibilidad de fallos en las compras. La versión no realizable de cada caso de estudio termina con un sufijo ÏN̈(por instatisfacible).

6.2. Experimentos

Para cada caso de estudio elegimos un MTS para representar el conocimiento inicial que el controlador tiene del ambiente. Usamos dos tipos distintos de conocimiento inicial. Para Celda de Producción, Mediación de Servicios, Búsqueda y Rescate, Agencia de Viajes, y Compra y Entrega asumimos que no existe ningún conocimiento inicial. En otras palabras, usamos un MTS con un sólo estado y con todas las acciones marcadas como posibles. Para el resto de los casos (GMES y Librería) asumimos cierta estructura del tipo pedido y respuesta, generando un MTS con un patrón mostrado en la Figura 6.1.

Para correr el algoritmo necesitamos proveerle una interfaz del ambiente concreto a controlar y descubrir. Para hacer esto, construimos un agente que, dada la descripción del comportamiento del ambiente en un LTS, implementa la interfaz del mundo. Como vimos

en el ejemplo del robot, una decisión clave es cómo este agente elige la próxima acción a ejecutar (es decir cómo se implementa `try`(A^c)). Esta decisión afecta cómo y cuánto se explora el ambiente.

Elegimos tres tipos de agentes: un ambiente que elige la próxima acción a ejecutar aleatoriamente, uno que facilita la exploración eligiendo transiciones que no han sido ejecutadas siempre que se pueda, y uno que dificulta la exploración eligiendo acciones previamente ejecutadas siempre que sea posible.

Dado que el algoritmo intenta controlar el ambiente cumpliendo la meta, de ser posible, *ad infinitum*, introducimos un criterio de parada para heurísticamente determinar si el proceso convergió (es decir, no ocurrirán más reset! en el futuro). Es importante notar que el algoritmo no puede reconocer este punto definitivamente, dado que no puede distinguir entre comportamiento que podría ocurrir pero que el ambiente decide no mostrar versus comportamiento que no puede ocurrir. Por eso usamos un criterio que frena el proceso de C&D cuando la cantidad de acciones de la traza de ejecución es 11 veces la cantidad de eventos que tomó llegar al último refinamiento. Después de frenar la ejecución, chequeamos si todo el comportamiento de W fue cubierto o si el comportamiento exhibido desde el último reset! corresponde a comportamiento que un controlador construido con total conocimiento de W hubiera ejecutado. Esta inspección contribuye a asegurar que efectivamente el comportamiento del controlador se estabilizó. Para todos los casos de estudio al momento de la interrupción el proceso C&D se estabilizó.

6.3. Resultados

En la Tabla 6.1 mostramos los resultados obtenidos al correr el algoritmo con el ambiente aleatorio (R), el ambiente que facilita la exploración (F) y el que lo dificulta (H). Reportamos la cantidad de refinamientos y eventos reset! ocurridos. Esto provee información sobre cuántas veces se incorpora nueva información al conocimiento que el algoritmo acumula, y la cantidad de veces que el algoritmo llega a un camino sin salida (es decir un momento donde es inevitable violar alguna propiedad) y tiene que reiniciar.

También reportamos la cantidad de acciones ejecutadas desde el inicio hasta el último refinamiento. Esto muestra qué tan rápido se consigue suficiente conocimiento como para controlar el ambiente o declarar la imposibilidad de cumplir las metas. Reportamos también la cantidad de acciones ejecutadas desde el inicio hasta el último reset!. En los casos en donde hubo reset! nos da una indicación sobre cuándo se empieza a cumplir el objetivo. Si no existe ningún reset! quiere decir que el comportamiento desde un principio cumple con los objetivos del sistema (es decir, los objetivos nunca fueron violados). Finalmente, reportamos el grado de cobertura del comportamiento del ambiente en términos de estados y transiciones.

En las tabla se pueden ver coberturas con gran variabilidad entre los distintos casos de estudio. Esta variabilidad puede explicarse por distintas razones, por ejemplo qué tan controlable es el ambiente, y cuánto ciertas acciones no controlables contribuyen a cumplir el objetivo, o llevan a violarlos. Por ejemplo, si un acción no controlable lleva a una violación de un objetivo, esto puede evitar explorar una gran parte de los estados. Obviamente, un ambiente que dificulta la exploración (H) típicamente conlleva menor cobertura que uno que facilite la exploración (F) o que uno aleatorio (R). Por eso la columna F tiende a producir mayor cobertura que R. Existe también alta variabilidad entre los casos de estudio

Caso de estudio	Refinamientos			Reinicios			Último refinamiento			Último reinicio (#Acciones)		
	R	F	H	R	F	H	R	F	H	R	F	H
Librería	19	19	8	0	0	0	60	54	8	-	-	-
Librería IN	17	17	9	1	1	0	54	30	9	-	24	-
GMES	45	45	6	0	0	0	318	350	7	-	-	-
GMES IN	33	39	8	5	6	0	69	73	8	104	150	-
Celda de Producción	54	58	4	6	6	0	176	171	5	137	148	-
Celda de Producción IN	38	29	15	4	3	3	89	56	54	81	-	-
Compra y Entrega	10	10	9	0	0	0	13	12	9	-	-	-
Compra y Entrega IN	19	20	11	0	0	0	37	38	13	-	-	-
Búsqueda y Rescate	36	48	30	0	0	0	132	388	238	-	-	-
Búsqueda y Rescate IN	9	11	4	1	1	0	12	13	5	-	-	-
Mediación de Servicios	27	38	29	0	0	0	1276	1354	3059	-	-	-
Mediación de Servicios IN	40	40	37	3	2	1	71	69	60	62	-	50
Agencia de Viajes	142	143	5	0	0	0	2767	1868	10	-	-	-
Agencia de Viajes IN	38	37	10	2	2	1	81	58	18	57	52	-

Caso de estudio	Cobertura de estados			Cobertura de transiciones		
	R	F	H	R	F	H
Librería	100	100	69	100	100	71
Librería IN	98	100	70	96	96	76
GMES	100	100	32	98	98	36
GMES IN	85	91	34	85	91	39
Celda de Producción	83	85	9	59	62	4
Celda de Producción IN	67	49	27	42	31	16
Compra y Entrega	25	25	24	9	9	7
Compra y Entrega IN	35	35	26	15	16	9
Búsqueda y Rescate	66	80	56	48	65	41
Búsqueda y Rescate IN	16	22	9	9	12	5
Mediación de Servicios	19	23	20	8	11	9
Mediación de Servicios IN	26	26	23	9	9	8
Agencia de Viajes	36	36	2	16	16	0
Agencia de Viajes IN	14	14	4	4	4	1

Tab. 6.1: Control y exploración para tres tipos de ambientes: uno que elige aleatoriamente (R), uno que facilita la exploración (F) y uno que dificulta la exploración (H). Los números se redondean al entero más cercano.

en términos de la cantidad de reset! y refinamientos requeridos para estabilizar el comportamiento o declarar insatisfacibilidad. Como era de esperarse, los ambientes donde no es posible realizar los objetivos tienden a tener más reset! que sus contrapartes realizables.

No reportamos tiempos de ejecución dado que son irrelevantes en nuestros experimentos usando ambientes sintéticos. El tiempo de ejecución en un contexto real dependerá de aspectos como la velocidad en la cual el servicio responde a acciones controlables o el tiempo que tarda en producirse un evento no controlable que el controlador debe esperar. El impacto del algoritmo C&D sobre el tiempo de ejecución depende del tiempo que cueste determinar la próxima acción a ejecutar. El tiempo promedio de esta operación es de menos de un segundo para 10 de los 14 casos de estudio, GMES y Agencia de Viajes son considerablemente más complejos con tiempos promedio de 18 y 32 segundos respectivamente.

Otro dato relevante que no se refleja en la tabla es cuáles de las ejecuciones para

casos de estudio irrealizables lograron los objetivos y no terminaron con el controlador concluyendo none!. Al igual que en nuestras hipótesis, al usar un ambiente que facilita la exploración (F), todas las ejecuciones concluyen none!. Esto sucede porque el controlador logra explorar suficiente comportamiento para concluir que es imposible lograr el objetivo. Lo mismo sucede para los ambientes que eligen las acciones de manera aleatoria (R) porque con suficientes intentos, todas las acciones no controlables se toman al menos una vez, y finalmente se consigue suficiente conocimiento como para concluir none!. Para ambientes que dificultan la exploración, 30 % de las ejecuciones no declaran none!, si no que cumplen los objetivos desde el último reset!. Estos casos corresponden a ejecuciones donde la primera vez que el ambiente elige acciones no controlables aleatoriamente, elige acciones que el controlador necesitaba para cumplir los objetivos, y luego sigue eligiendo la misma acción (para dificultar exploración de nuevos caminos), lo cual ayuda al controlador.

En contraposición a nuestro enfoque, en el cual se pueden lograr los objetivos en contextos irrealizables dependiendo del comportamiento del ambiente, consideremos un enfoque donde el controlador elige aleatoriamente qué acción tomar hasta que tenga suficiente conocimiento para producir una estrategia que gane o declarar insatisfacibilidad. Un enfoque de este tipo con un ambiente irrealizable corre el riesgo de no lograr ninguna de las dos. Experimentamos con un enfoque de este estilo y encontramos ejecuciones en donde el sistema controlado continuamente ejecuta reset! y nunca declara none!. Encontramos este comportamiento con el caso de Celda de Producción insatisfacible donde dejamos correr la ejecución 20 veces más tiempo que cualquier otra ejecución y encontramos que periódicamente se ejecuta reset!, sin jamás llegar a none!.

En conclusión, aplicamos C&D a versiones realizables e irrealizables de siete casos de estudio tomados de la literatura. Para cada caso usamos tres distintos tipos de ambiente que varían en la dificultad de exploración de su comportamiento. Para cada uno de estos 42 escenarios observamos que el algoritmo es exitoso en su tarea de C&D pero que el grado de descubrimiento del comportamiento y la cantidad de eventos reset! varía significativamente dependiendo de la estrategia del ambiente.

7. TRABAJOS RELACIONADOS Y DISCUSIÓN

En el contexto de sistemas orientados a servicios, gran parte de los estudios se enfoca en interactuar con ambientes desconocidos e intentar lograr objetivos del sistema. La exploración de servicios ya existe para muchos ambientes (por ejemplo Google API Discovery Service, WS-Discovery, Bonjour) y la descripción de interfaces se puede hacer mediante lenguajes como WSDL. La verificación de las estrategias de orquestación para tales ambientes típicamente asume la existencia de modelos de comportamiento de los servicios a coordinar (por ejemplo [20]).

La construcción automatizada de conectores, orquestadores y mediadores se puede entender como un problema de síntesis de controladores. Los enfoques de síntesis de conectores como [21] usan razonamiento ontológico y definiciones de restricciones para inferir automáticamente mapeos entre las interfaces de los componentes. Cuando no se tienen los modelos de comportamiento, estos enfoques sugieren usar, como preprocesado, técnicas de aprendizaje de autómatas (por ejemplo el *framework* MAT [1]) para la extracción de modelos de comportamiento [2] y para la construcción de *middleware* [3]. A diferencia de nuestro enfoque, el *framework* MAT requiere la capacidad de responder consultas de equivalencia, típicamente aproximadas mediante pruebas de conformidad (por ejemplo [22]). Además, se asume que el símbolo abstracto obtenido como resultado está determinado unívocamente por la secuencia de símbolos abstractos de input [23]. Esta suposición no vale en general cuando el ambiente muestra comportamiento no controlable. Nuestro enfoque busca sobrepasar esta limitación al integrar el proceso de exploración con la estrategia de control (es decir, no es un enfoque bifásico).

La comunidad que estudia la planificación de misiones de movimiento y de alto nivel (por ejemplo [24,25]) abordó este problema recientemente; desde descripciones imprecisas del ambiente ([26–31]), modelos parciales del ambiente ([32]) e inferencias de modelos ([33,34]). En el primer grupo de trabajos, los procesos de síntesis se adaptan para tolerar violaciones de las suposiciones hechas en los modelos. El trabajo presentado en [26] considera incertidumbres en sistemas de transiciones abiertos y finitos debido a transiciones no modeladas. Del mismo modo, [27] propone una manera de sintetizar controladores robustos que son capaces de ganar incluso si ocurren ciertas transiciones inesperadas una cantidad finita de veces. Similarmente, [28] resuelve el problema de sintetizar sistemas resistentes a errores desde especificaciones en lógica temporal. Resistencia a errores significa tolerar cualquier cantidad de violaciones a las suposiciones de seguridad. El trabajo en [29] presenta un *framework* escalonado para combinar modelos de comportamiento, cada uno con diferentes suposiciones y riesgos asociados debido al modelado de inexactitudes. En [30,31] se extiende el controlador para una especificación originalmente realizable con acciones que, si existen, preservan los requerimientos de seguridad y aseguran que el robot pueda progresar hacia sus metas cuando el ambiente retome el comportamiento esperado.

En este caso se empieza con una especificación realizable y el ambiente es quien puede agregar comportamiento inesperado. Esto no es adecuado cuando hay que descubrir el comportamiento para lograr los objetivos: en nuestro abordaje del problema, el ambiente puede exhibir o incluso eliminar comportamiento posible y no hay necesidad de empezar con una especificación realizable. Se han usado MTS [4] para razonar sobre conocimiento parcial de comportamiento de sistemas (por ejemplo, [35]) y también para sintetizar con-

troladores con conocimiento incompleto del ambiente [6,36]. Sin embargo, no se considera refinamiento progresivo de modelos MTS basados en observaciones en tiempo de ejecución.

En [32], se propone un método ad-hoc, inspirado en MTS [4], para producir planes de misiones con múltiples robots y conocimiento parcial. Los autores fijan tres fuentes de incertidumbre: conocimiento parcial de la ejecución de acciones, aprovisionamiento de servicios desconocido, y desconocimiento de la capacidad de los robots para encontrarse. Cada tipo de incertidumbre se maneja de manera distinta. Esta solución no busca manejar modelos parciales generales de sistemas reactivos de manera uniforme, como proponemos nosotros. En particular, el abordaje usado para la planificación no puede manejar incertidumbre respecto de acciones no controlables: solamente manejan casos de respuestas inciertas a acciones controlables y no pueden, por ejemplo, manejar la ocurrencia de eventos como interrupciones no controlables. Otra diferencia con nuestro trabajo es que en [32], si en una ocasión no ocurre una transición incierta, es suficiente para concluir, durante la ejecución, que dicha transición no está presente. En el ejemplo que damos como motivación para este trabajo (Sección 2) esto significa asumir que si la primera consulta responde que el libro está disponible, todas las consultas posteriores también responderán lo mismo.

Otra colección de resultados trabaja con planificación adaptativa (por ejemplo, [33,34]) y tratan un problema cercano al nuestro, donde la forma exacta del modelo es parcialmente desconocida. En [37] se usa un *framework* que integra inferencia gramática con control simbólico sobre sistemas de transición de estados finitos que interactúan con ambientes desconocidos, adversarios y gobernados por reglas: es decir, los estados son observables a través de la asignación de proposiciones que rigen la precondición y poscondición de las acciones. Sin embargo, aparecen algunas limitaciones que no están presentes en nuestro trabajo. En primer lugar, la clase de gramática que representa el ambiente tiene que ser conocida con anticipación [38]. En segundo lugar, se da una enumeración positiva del lenguaje del juego (una función que enumera todos los prefijos de las corridas del juego) como hipótesis de los resultados teóricos de jugar un juego equivalente. La enumeración positiva puede ser potencialmente obstaculizada por un ambiente adversario mientras que nuestro enfoque explícitamente trata con un ambiente que no colabora mostrando todo su comportamiento.

Hasta donde sabemos, *el uso de síntesis de controladores para desarrollar estrategias que también exploran es una idea nueva.* La síntesis basada en conocimiento parcial garantiza que cuando el controlador elige una acción controlable progresa hacia lograr los objetivos ó finalmente revela comportamiento del ambiente previamente desconocido. Es decir, garantiza que la acción elegida no es perdedora basándose en el conocimiento actual, aunque el ambiente puede prevenir que el objetivo se logre. Una exploración aleatoria puede elegir acciones que son perdedoras, llevando a comportamiento que no agrega información ni se acerca a cumplir las metas. Una estrategia de exploración que sólo busca explorar (por ejemplo el camino más corto a una acción posible en K) puede no descubrir todo el comportamiento y tampoco lograr los objetivos.

7.1. Limitaciones de nuestro enfoque

El enfoque presentado en esta tesis tiene dos suposiciones que pueden ser limitantes en varios contextos. Ambas suposiciones están relacionadas con las expectativas que se tiene de la interfaz del ambiente. Asumimos que es posible pedirle al ambiente un identificador único del estado actual en que se encuentra. También asumimos que es posible reiniciar

el ambiente, devolviéndolo a su estado inicial. Si bien puede parecer que esta suposición es muy fuerte, se puede argumentar que es menos restrictiva que las suposiciones hechas en los abordajes difundidos de aprendizaje basado en autómatas inspirados por el *framework* MAT [1]. En estos abordajes el uso de consultas de equivalencia implica responder una pregunta más compleja (equivalencia de comportamiento, que implica conocer el estado actual y sus capacidades de comportamiento) y se implementa típicamente mediante pruebas de conformidad que requieren la habilidad de reiniciar el ambiente.

Notemos que los sistemas orientados a servicios (en donde las sesiones se pueden pensar como reinicios, y donde los identificadores de estado se pueden implementar con funciones de hash) son un ejemplo de la clase de sistemas que pueden ser compatibles con nuestras suposiciones del ambiente. Además, notemos que para 6 de 7 casos de estudio no se necesitó ningún reinicio para explorar el ambiente lo suficientemente como para lograr el objetivo deseado.

Sin embargo, nuestras suposiciones limitan la aplicabilidad del abordaje. Más allá de sistemas orientados a servicios, los sistemas ciber-físicos son significativamente más difíciles de tratar. Para estos casos es importante considerar cómo extender el trabajo para considerar conocimiento parcial respecto de los identificadores de estados.

Si bien esta tesis va más allá de una solución puramente teórica al proveer una implementación y mostrar que se pueden tratar casos de estudios de la literatura, persisten importantes desafíos técnicos a la hora de efectivamente llevar a la práctica este trabajo. Uno de estos desafíos es el de desarrollar una interfaz para interactuar con el sistema a controlar. Nos imaginamos una gran cantidad de alternativas dependiendo de cómo se interactúa con la interfaz y de la infraestructura del sistema a controlar. Creemos que sería más fácil diseñar o alterar un componente para proveer una interfaz que alterarlo para dar siempre un modelo del protocolo actualizado. En última instancia, si un componente externo tiene comportamiento no controlable, no es posible usar abordajes tradicionales de aprendizaje con máquinas Mealy, se necesita algún tipo de exploración.

8. CONCLUSIONES

En esta tesis definimos el problema de control y exploración que permite controlar ambientes inicialmente desconocidos. El problema permite (pero no requiere) empezar con conocimiento parcial del comportamiento del ambiente, representado con un MTS. Presentamos también una solución al problema de control y exploración para metas FLTL construyendo sobre control para MTS. El algoritmo asume que el proceso de exploración puede identificar cuándo el ambiente retorna a un estado previamente visitado, y que el comportamiento del ambiente corresponde al de un lenguaje regular. La solución al problema de control de MTS provee un controlador que gana conocimiento sobre el ambiente incrementalmente ó garantiza las metas del sistema. Discutimos una implementación del algoritmo de control y exploración en MTSA y reportamos su uso en varios casos de estudios tomados de la literatura.

Bibliografía

[1] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987. [Online]. Available: http://dx.doi.org/10.1016/0890-5401(87)90052-6

[2] A. Bennaceur, V. Issarny, D. Sykes, F. Howar, M. Isberner, B. Steffen, R. Johansson, and A. Moschitti, "Machine learning for emergent middleware," in *Proceedings of the Second International Conference on Trustworthy Eternal Systems via Evolving Software, Data and Knowledge*, ser. EternalS'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 16–29.

[3] P. Inverardi, V. Issarny, and R. Spalazzese, "A theory of mediators for eternal connectors," in *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part II*, ser. ISoLA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 236–250. [Online]. Available: http://dl.acm.org/citation.cfm?id=1939345.1939374

[4] K. Larsen and B. Thomsen, ""A Modal Process Logic"," in *Proceedings of 3rd Annual Symposium on Logic in Computer Science (LICS'88)*. IEEE Computer Society Press, 1988, pp. 203–210.

[5] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive (1) designs," *Lecture notes in computer science*, vol. 3855, pp. 364–380, 2006.

[6] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, "The modal transition system control problem," in *FM 2012: Formal Methods*, D. Giannakopoulou and D. Méry, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 155–170.

[7] A. van Lamsweerde, *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[8] D. Giannakopoulou and J. Magee, "Fluent model checking for event-based systems," in *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-11. ACM, 2003, pp. 257–266.

[9] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *J. Comput. Syst. Sci.*, vol. 78, no. 3, pp. 911–938, 2012. [Online]. Available: https://doi.org/10.1016/j.jcss.2011.08.007

[10] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, "Synthesising non-anomalous event-based controllers for liveness goals," *ACM Tran. Softw. Eng. Methodol.*, vol. 22, 2013.

[11] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '89. ACM, 1989, pp. 179–190.

[12] N. D. y V. Braberman y N. Piterman y S. Uchitel y M. Keegan, "Extensión MTSA," 2016, https://bitbucket.org/ndippolito/2016-exploration/src/master/.

[13] N. Nostro, R. Spalazzese, F. D. Giandomenico, and P. Inverardi, "Achieving functional and non functional interoperability through synthesized connectors," *Journal of Systems and Software*, vol. 111, pp. 185 – 199, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215002149

[14] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel, "Synthesis of live behaviour models for fallible domains," in *ICSE*, R. N. Taylor, H. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 211–220.

[15] C. Lewerentz and T. Lindner, Eds., *Formal Development of Reactive Systems - Case Study Production Cell*, ser. Lecture Notes in Computer Science, vol. 891. London, UK: Springer-Verlag, 1995.

[16] W. Heaven, D. Sykes, J. Magee, and J. Kramer, "Software engineering for self-adaptive systems," in *Software Engineering for Self-Adaptive Systems*, B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer-Verlag, 2009, ch. A Case Study in Goal-Driven Architectural Adaptation, pp. 109–127.

[17] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso, "Planning and monitoring web service composition," in *Artificial Intelligence: Methodology, Systems, and Applications*, ser. Lecture Notes in Computer Science, C. Bussler and D. Fensel, Eds. Springer-Verlag, 2004, vol. 3192, pp. 106–115.

[18] P. Inverardi and M. Tivoli, "A reuse-based approach to the correct and automatic composition of web-services," in *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*, ser. ESSPE '07. ACM, 2007, pp. 29–33.

[19] A. Bennaceur and V. Issarny, "Mics: Mediator synthesis to connect components website," 2019. [Online]. Available: https://www.rocq.inria.fr/arles/index.php/software/219-mics

[20] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Model-based verification of web service compositions," in *ASE*, 2003, pp. 152–163.

[21] A. Bennaceur and V. Issarny, "Automated synthesis of mediators to support component interoperability," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 221–240, March 2015.

[22] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, "Combining model learning and model checking to analyze tcp implementations," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 454–471.

[23] F. Aarts, B. Jonsson, J. Uijen, and F. Vaandrager, "Generating models of infinite-state communication protocols using regular inference with abstraction," *Form. Methods Syst. Des.*, vol. 46, no. 1, pp. 1–41, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1007/s10703-014-0216-x

[24] G. E. Fainekos, A. Girard, H. Kress-Gazit, and G. J. Pappas, "Temporal logic motion planning for dynamic robots," *Automatica*, vol. 45, no. 2, pp. 343–352, Feb. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.automatica.2008.08.008

[25] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G. J. Pappas, "Symbolic planning and control of robot motion [grand challenges of robotics]," *IEEE Robotics Automation Magazine*, vol. 14, no. 1, pp. 61–70, March 2007.

[26] U. Topcu, N. Ozay, J. Liu, and R. M. Murray, "On synthesizing robust discrete controllers under modeling uncertainty," in *Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '12. New York, NY, USA: ACM, 2012, pp. 85–94. [Online]. Available: http://doi.acm.org/10.1145/2185632.2185648

[27] S. Dathathri, S. C. Livingston, and R. M. Murray, "Enhancing tolerance to unexpected jumps in gr(1) games," in *2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems (ICCPS)*, April 2017, pp. 37–48.

[28] R. Ehlers and U. Topcu, "Resilience to intermittent assumption violations in reactive synthesis," in *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '14. New York, NY, USA: ACM, 2014, pp. 203–212. [Online]. Available: http://doi.acm.org/10.1145/2562059.2562128

[29] N. D'Ippolito, V. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel, "Hope for the best, prepare for the worst: Multi-tier control for adaptive systems," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 688–699. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568264

[30] K. W. Wong, R. Ehlers, and H. Kress-Gazit, "Resilient, provably-correct, and high-level robot behaviors," *IEEE Transactions on Robotics*, vol. 34, no. 4, pp. 936–952, Aug 2018.

[31] ——, "Correct high-level robot behavior in environments with unexpected events," in *Proceedings of Robotics: Science and Systems*, Berkeley, USA, July 2014.

[32] C. Menghi, S. Garcia, P. Pelliccione, and J. Tumova, "Multi-robot ltl planning under uncertainty," in *Formal Methods*, K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, Eds. Cham: Springer International Publishing, 2018, pp. 399–417.

[33] J. Fu, H. G. Tanner, and J. Heinz, "Adaptive planning in unknown environments using grammatical inference," in *52nd IEEE Conference on Decision and Control*, Dec 2013, pp. 5357–5363.

[34] J. Fu, H. G. Tanner, J. N. Heinz, K. Karydis, J. Chandlee, and C. Koirala, "Symbolic planning and control using game theory and grammatical inference," *Engineering Applications of Artificial Intelligence*, vol. 37, pp. 378 – 391, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0952197614002401

[35] S. Uchitel, G. Brunet, and M. Chechik, "Synthesis of partial behavior models from properties and scenarios," *IEEE Transactions on Software Engineering*, vol. 35, pp. 384–406, May 2009.

[36] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel, "Controllability in partial and uncertain environments," in *14th International Conference on Application of Concurrency to System Design, ACSD 2014, Tunis La Marsa, Tunisia, June 23-27, 2014*. IEEE, 2014, pp. 52–61. [Online]. Available: http://dx.doi.org/10.1109/ACSD.2014.15

[37] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. New York, NY, USA: Cambridge University Press, 2010.

[38] J. Heinz, "String extension learning," in *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ser. ACL '10. Stroudsburg, PA, USA: Association for Computational Linguistics, 2010, pp. 897–906. [Online]. Available: http://dl.acm.org/citation.cfm?id=1858681.1858773