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

TESIS
Síntesis Dirigida de Controladores No Maximales para Requerimientos de tipo Non-Blocking

Tesis de Licenciatura en Ciencias de la Computación

Matías Duran, Florencia Zanollo

Director: Sebastián Uchitel

Buenos Aires, 2021

SINTESIS DE CONTROLADORES DIRIGIDA

Esta tesis presenta la primera implementación de síntesis de directores para resolver problemas de Control de Eventos Discretos con la propiedad central de tipo non-blocking.

El método aprovecha la naturaleza composicional del problema y su input compacto, minimizando la explosión exponencial al componer la planta total. Con el enfoque de exploración on-the-fly de forma best-first-search guiada por distintas heurísticas, se busca reducir la parte de la planta a componer.

La implementación fue incorporada al software MTSA[1], junto con una batería de tests para conservar su correctitud ante futuros cambios.

Palabras claves: Sistemas de Eventos Discretos, Síntesis de Controladores, Control Dirigido, Control Supervisado, Síntesis on-the-fly, LTS, Non-Blocking.

[1] Modal Transition System Analyser, https://bitbucket.org/lnahabedian/mtsa/src/master/

DIRECTED CONTROLLER SYNTHESIS

This thesis presents the first implementation of director synthesis, to solve problems from the field of Discrete Event Control, with the central property of non-blocking controlers.

The method takes advantage of the compositional nature of the problem and its compact input, minimizing the exponential explosion of composing the complete plant. The on-the-fly exploration with best-first-search guided by different heuristics aims to reduce the part of the plant needed to be composed.

The implementation was incorporated to the MTSA[2] project, along with a test set to assert the correct results in case of future changes.

Keywords: Discrete Event Systems, Controler Synthesis, Directed Control, Supervisory Control, On-The-Fly Synthesis, LTS, Non-Blocking.

[2] Modal Transition System Analyser, https://bitbucket.org/lnahabedian/mtsa/src/master/

AGRADECIMIENTOS

Florencia

A mi padre y mi madre, que siempre me apoyaron en todo y me brindaron la oportunidad de estudiar una carrera universitaria. Al amor de mi vida por ser tan buen compañero y bancar mis peores momentos. A mi hermano, mis amigos y amigas, tanto de la facultad como de la vida, por darme esos respiros, ayudas y escapadas. A mi familia y mis hermosas mascotas, por el amor de siempre. A gran parte de la gente del Departamento de Computación, es increíble la calidad humana con la que contamos; mención especial a muches del LaFHIS. Al excelente director que tuve la suerte de tener y, obviamente, a mi compañero de tesis; sin vos esto hubiese sido muy muy difícil.

Matías

A mi familia, por haberme dado tanto de quien soy, y por haber estado siempre presentes compartiendo amor a lo largo de los años. A mis hermanos, por soportarme este año apocalíptico y alegrar las tardes en que me saturaba de escribir este documento. A mis amigos y amigas por estar ahí, para las charlas, risas y momentos compartidos que todo lo valen. También a toda la gente que día a día trabaja para que el DC y el LaFHIS sean grandes lugares para desarrollarse, en lo personal y en lo académico. A nuestro maravilloso director de tesis, por su guía y generosidad. Finalmente, a mi co-tesista por su gran energía y personalidad, sin la cual este trabajo no hubiera sido tan disfrutable.

Índice general

1. INTRODUCCIÓN

El ser humano siempre buscó la manera de automatizar las tareas pesadas o tediosas, la ciencia de la computación surgió en parte a causa de ello pero ¿Es posible automatizar aún más? Hacer que una computadora "se programe" a sí misma dado un problema a resolver es en cierta medida una utopía, sin embargo hay muchas ramas de la computación que atacan esta idea. Ya sea Machine Learning, Inteligencia Artificial o, en el caso del presente trabajo, Síntesis Automática de Controladores.

Síntesis porque no es un humano quien desarrolla manualmente el código *solución* del problema sino que se le brinda a un programa las reglas y objetivos (o pre y post condiciones) a cumplir, dejando "a criterio del mismo" el cómo hacerlo. Lo que devuelve el programa es una *estrategia* que garantiza ganar (si existe forma, caso contrario avisa la inexistencia de solución), a esta estrategia se la conoce con el nombre de *Controlador*.

Un ejemplo de aplicación es controlar aviones para demarcar incendios forestales [14], donde a veces incluso es necesario que el programa pueda adaptarse en medio de su ejecución.

El problema de síntesis fue estudiado dentro de distintas áreas como: Control de Eventos Discretos [12], Síntesis Reactiva [11] y Planning automático [10]. Si bien este trabajo desarrolla resultados nuevos combinando enfoques de las tres áreas, nos basamos principalmente en Control de Eventos Discretos (Discrete Event Control ó DEC).

En DEC el problema es modelado utilizando autómatas finitos, también conocidos como máquinas de estados finitos. Un autómata finito, está conformado por estados y transiciones entre estos estados. Podemos pensar los estados como puntos y las transiciones como flechas de un sólo sentido que los unen. Se los llama estados ya que representan un momento particular en el objeto que se quiere modelar, por ejemplo en el caso del avión un estado podría ser "aterrizado" y otro "en vuelo". Las transiciones en nuestro caso son vistas como acciones, algunas las podemos controlar y otras están fuera de nuestro alcance; volviendo al ejemplo del avión una acción controlable podría ser "encender el motor" y una no controlable sería "quedarse sin combustible".

Además existen estados distinguidos dentro del autómata: el inicial (estado en cual empezamos) y los estados objetivos (a los cuales se quiere llegar). En nuestro caso a estos últimos los llamamos *estados marcados* y no sólo queremos que el sistema (el avión) pueda alcanzarlos, sino que sea capaz de hacerlo infinitas veces y de manera segura. Segura en el sentido de siempre tener una secuencia de acciones para llegar, evitando estados peligrosos (por ejemplo, "quedarse sin batería"). ¡No podemos quedarnos sin transiciones ni usarlas en el sentido contrario!

Habiendo presentado los autómatas, aclaramos que el valor de este trabajo y de un algoritmo de síntesis de controlador, es que generaliza a distintos problemas. El autómata puede ser sobre un avion, o una agencia de viajes como se verá en otro ejemplo, y nuestro algoritmo generará un controlador sin necesitar ningún cambio. Por un lado se encuentra el trabajo del experto en el área, al decidir por ejemplo que el estado marcado sea que el avión aterrice, o que forme un patrón en el cielo, o cualquier otro objetivo para cosas que no sean aviones. Lo importante es que ese experto no debe preocuparse por como se sintetiza el controlador que resuelve el autómata que modeló para su problema, esa parte está abstraída en el algoritmo de síntesis de controladores.

Por último, modelar un sistema complejo utilizando un solo autómata puede ser complicado y hasta imposible. Es por esto que se suele modelar sus partes como autómatas y componerlos para formar el modelo completo del sistema a controlar. La composición debe respetar ciertas reglas y al finalizar, un estado del sistema compuesto representa la combinación de estados en que está cada componente.

Tradicionalmente se toman estos autómatas componentes y se realiza la composición, obteniendo el autómata completo que luego será analizado por el programa para obtener el controlador. El problema es que en ciertos casos la cantidad de estados y transiciones resultantes del sistema es demasiado grande, haciendo imposible su análisis y en ocasiones hasta es imposible terminar de componer.

Es en este contexto que surge el enfoque de la exploración on-the-fly o "en el mientras". La idea es tratar de sacar conclusiones mientras se va calculando la composición para, de esta manera, evitar en lo posible construir todo el modelo. El enfoque fue presentado por primera vez en la tesis doctoral de Daniel Ciolek [2]; pero al desarrollar la exploración junto con las heurísticas se obtuvo un algoritmo dependiente de las mismas, con fallas en las conclusiones tomadas al explorar y sin la posibilidad de adaptarse a heurísticas diferentes. El enfoque principal de este trabajo fue corregir dichas fallas, obteniendo un algoritmo de exploración completo, correcto y capaz de soportar cualquier variación en la heurística. El resultado principal de este trabajo es el algoritmo de *Síntesis de Controladores Dirigida* (SCD, o DCS en inglés).

Para evidenciar la dificultad de la cual hablamos a la hora de componer el modelo completo presentaremos a continuación un ejemplo, con sus componentes y el sistema resultante de la composición.

En el siguiente capítulo, cap.2, presentamos los antecedentes, definiciones necesarias y un algoritmo tradicional monolítico. Luego en el capítulo 3, presentamos la idea básica de exploración on-the-fly junto con una descripción de nuestro enfoque en particular, como forma introductoria a la demostración formal.

Ya con las intuiciones del algoritmo introducidas, el capítulo 4 muestra nuestra propuesta de algoritmo de exploración en detalle, con el pseudocódigo, demostración de corrección y completitud para el mismo y análisis de la complejidad computacional de su peor caso.

Seguidamente, en el capítulo 5, exponemos detalles sobre la implementación, MTSA, heurísticas diseñadas para testeo y detalles sobre la batería de tests, desarrollada utilizando TDD (Test Driven Development).

Como paso final en cap.6 presentamos el benchmark utilizado y resultados de performance; tanto entre distintos resultados de SCD como versus otros algoritmos del estado del arte.

Finalmente, en cap.7 listamos las conclusiones y aportes del trabajo presentado.

1.1. Ejemplo

A continuación se presenta un ejemplo para comprender el problema a resolver. Se trata de una versión simplificada del problema *TravelAgency* utilizado en el capítulo 6 para medir la performance del algoritmo.

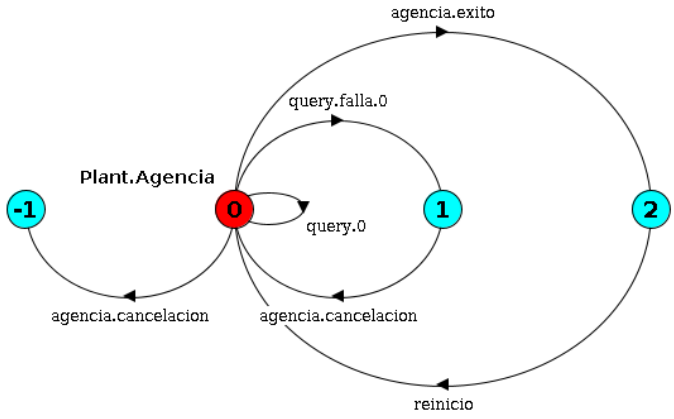
Se desea armar una agencia de venta online de paquetes vacacionales que reservará de forma automática una variedad de servicios (alquiler de auto, hotel, pasaje de avión, etc.) asegurando que no se perderá nada de dinero a menos que se reserve el paquete completo. Es decir, no se quiere perder dinero por la reserva del hotel si no va a ser posible comprar

un pasaje de avión.

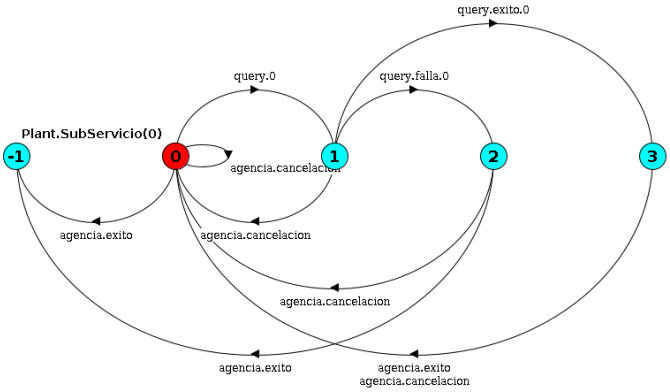
Para cada servicio que se desea sub-contratar se consulta (query) si ese servicio está disponible. En caso de tener éxito y confirmar la disponibilidad, se espera a confirmar el resto para comprar todos al mismo tiempo. En caso de que algún otro servicio no esté disponible, se cancela la compra y se vuelve al estado inicial, lo cual no implica un gasto. Esta sincronización de solo darse la compra en caso de poder contratar todos los servicios se da ya que el evento *agencia.exito* (que representa la confirmación de la compra) se encuentra en todos los componentes, por lo que no puede darse a menos que todos lo tengan habilitado (esto se comprenderá mejor con la definición 2, en el capítulo 2).

El problema puede escalar de forma muy rápida si se incrementa la cantidad de servicios a contratar o la cantidad de pasos para reservar cada servicio (como se verá en la sección 6), por lo que para este ejemplo ilustrativo asumimos que no hacen falta múltiples pasos para reservar un servicio, y solo mostramos el problema con 1 y 2 sub-servicios.

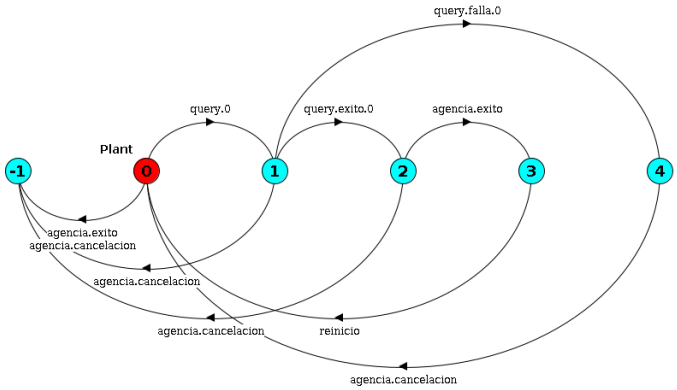
Mostramos en la figura 1.1 una representación gráfica de los LTS (labeled transition system) para cada uno de los componentes descriptos (agencia y sub-servicio), el LTS compuesto para el caso en el que se sub-contrata un solo servicio y el controlador resultante de aplicar nuestro algoritmo.



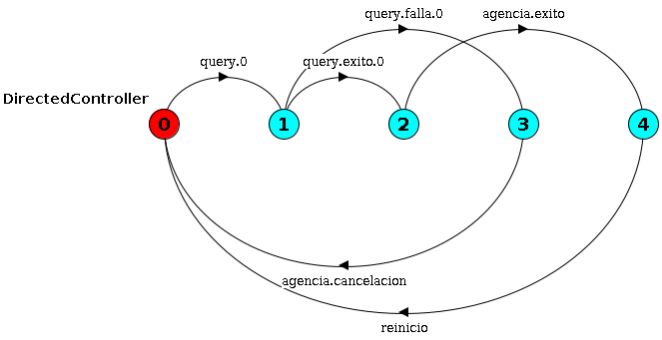
(a) la agencia



(b) el sub servicio a contratar



(c) Planta compuesta



(d) Controlador resultante

Fig. 1.1: Caso con un solo sub-servicio

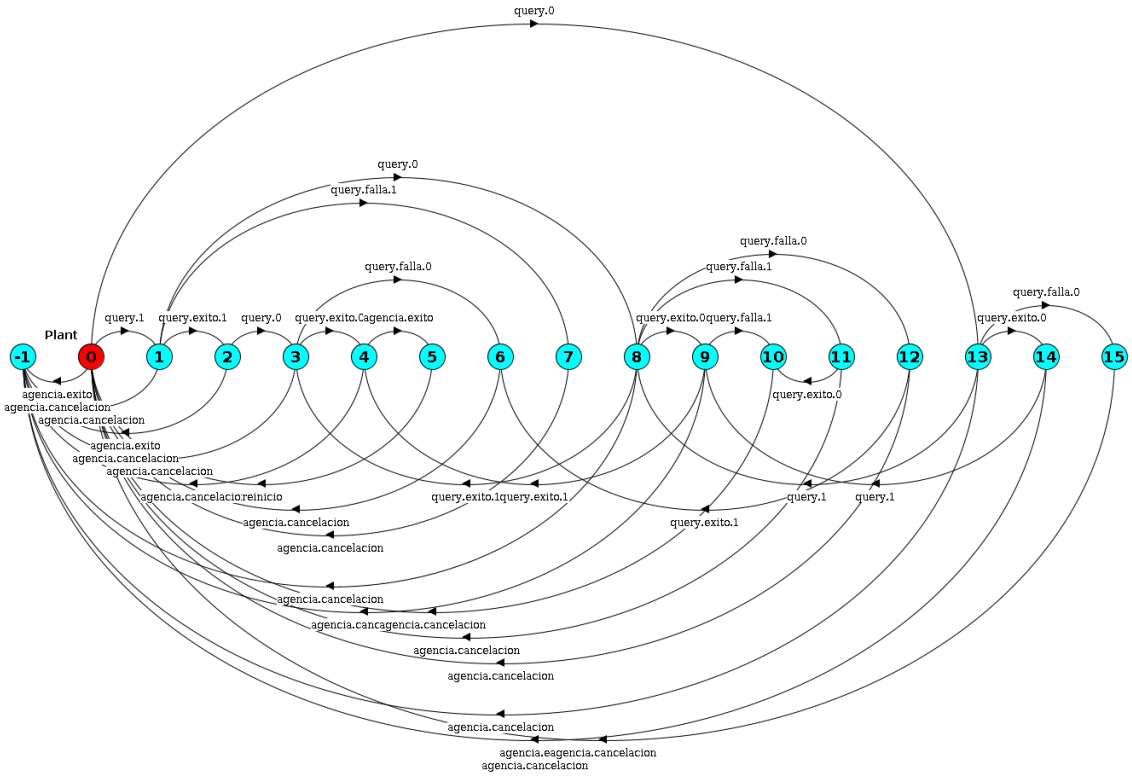


Fig. 1.2: Planta compuesta con 2 sub-servicios

En el ejemplo, la figura 1.1a representa a la agencia, que comienza en el estado inicial (0), y lo que quiere es confirmar la compra, el estado 2 es nuestro estado marcado, objetivo que queremos alcanzar infinitas veces. El estado 0 representa un estado en el que no se detectaron problemas con ningún sub-servicio, por lo que permite, cuando los otros componentes se sincronicen, tomar el evento *agencia.exito*. En cambio, al detectar algún *query.falla*, se mueve al estado 1, señalando que hubo un problema y solo permite cancelar la compra de todos los otros servicios consultados.

El componente del subservicio (figura 1.1b) también tiene el evento *agencia.exito*, pero debe previamente haber recibido la consulta (*query.0*) y confirmado su disponibilidad tomando el evento *query.exito.0*. En caso de no ser posible la reserva (*query.falla.0*), solo permite la cancelación de la compra (*agencia.cancelacion*), yendo del estado 2 al 0; si se quisiera confirmar la compra (*agencia.exito*) se llega al estado -1, reservado para los errores que el controlador debe evitar. Además, en caso de que la agencia cancele la compra del paquete de servicios, se vuelve al estado 0 y se espera una consulta.

Recordemos que queremos representar un sistema complejo con interacción de múltiples componentes. Esta sincronización se ve reflejada en la figura 1.1c, que muestra la planta que representa al sistema con los dos componentes previamente descriptos.

Finalmente, 1.1d muestra un posible controlador para este problema. Destacamos que todos los eventos controlables que llevaban al estado -1 (el estado error), no son una posibilidad permitida por el controlador.

Puede verse que para el caso de solo un sub-servicio que debe ser contratado el problema es manejable. Los modelos gráficos de la planta y el controlador, generados au-

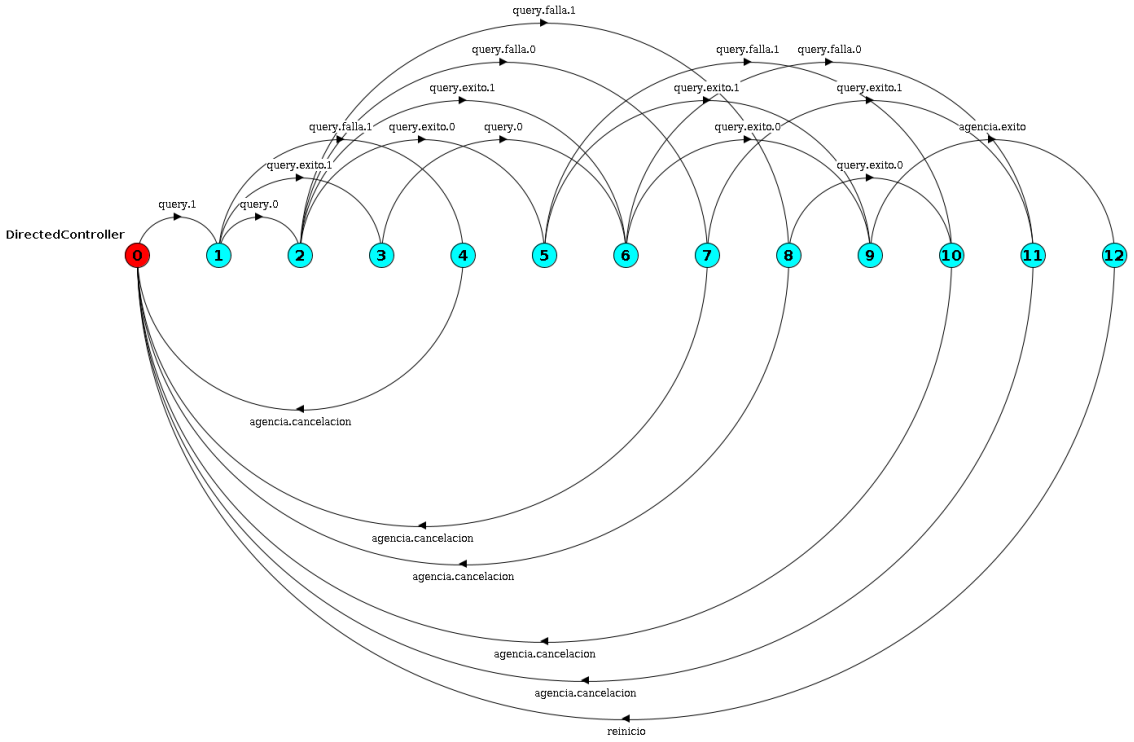


Fig. 1.3: Controlador para el problema con 2 sub-servicios

tomáticamente por MTSA[1] se comprenden con un vistazo. Ya el caso con $N = 2$ (fig 1.2) si bien puede generarse una representación gráfica, requiere un trabajo considerable para comprender qué estado del problema representa cada estado del modelo. Simplemente aumentando a $N = 5$ ya la planta compuesta cuenta con 1025 estados y 4085 transiciones, imposibilitando mostrarlo en una figura.

Destacamos del problema con 2 sub-servicios, que el controlador (figura 1.3) no solo imposibilita alcanzar el estado error, si no que también reduce las posibilidades en mayor medida. Por ejemplo, en la planta, desde el estado 0, ambos eventos controlables *query.0* y *query.1* son válidos, pero el controlador permite solo *query.1*, esto es acorde a la definición de director (def 4, sección 2.2).

Los algoritmos tradicionales necesitan construir la planta compuesta entera antes de empezar a construir un controlador. Dada la explosión de estados y transiciones evidente en este simple ejemplo, esto resulta muy costoso (a veces imposible), por ende las técnicas del área enfrentan serios inconvenientes al escalar el tamaño de los problemas a resolver. La exploración on-the-fly dirigida busca construir sólo la parte necesaria de la planta, sacando conclusiones en función de la información obtenida; en el peor caso, termina contruyendo toda la planta y obtiene el controlador de forma tradicional.

[1] Modal Transition System Analyser, https://bitbucket.org/lnahabedian/mtsa/src/master/

2. ANTECEDENTES

A continuación definimos formalmente el problema composicional de síntesis de controlador nonblocking.

Definición 1 (Autómata Determinístico): Un *autómata determinístico* es una tupla $T = (S_T, A_T, \rightarrow_T, \bar{t}, M_T)$, donde: S_T es un *conjunto finito de estados*; A_T es el *conjunto de eventos* del autómata; $\rightarrow_T \subseteq (S_T \times A_T \times S_T)$ es una *función de transición*; $\bar{t} \in S_T$ es el *estado inicial*; y $M_T \subseteq S_T$ es un conjunto de *estados marcados*.

Notación 1 (Pasos y corridas): Notamos $(t, \ell, t') \in \rightarrow_T$ como $t \xrightarrow{\ell}_T t'$ y lo llamamos *paso*. A su vez, una *corrida* de una palabra $w = \ell_0, \ldots, \ell_k$ en T, es una secuencia de pasos tal que $t_i \xrightarrow{\ell_i}_T t_{i+1}$ para todo $0 \leq i \leq k$, notado como $t_0 \overset{w}{\dashrightarrow}_T t_{k+1}$.

Notación 2: Decimos que un estado s es alcanzado por una palabra w en una corrida comenzando en el estado s', anotado como $s \in w(s')$, cuando $\exists w_0 \,.\, \exists w_1 \,.\, w = w_0.w_1$ y $s' \overset{w_0}{\dashrightarrow}_E s$

Los autómatas definen un lenguaje, un conjunto de palabras, que aceptan. Dado un conjunto de eventos A, notamos con A^* al conjunto de palabras finitas de eventos de A. El lenguaje generado por un autómata T (notado como $\mathcal{L}(T)$) es el conjunto de palabras formadas por sus eventos que cumplen $\rightarrow_T$. Formalmente, si $w \in A_T^*$, entonces $w \in \mathcal{L}(T)$ si y solo si existe una corrida para w comenzando desde el estado inicial $\bar{t}$ de T, que notamos $\bar{t} \overset{w}{\dashrightarrow}_T t_{k+1}$.

Generalmente los estados marcados se utilizan para indicar el logro de una tarea. Por lo tanto, consideramos como lenguaje *marcado* (o *aceptado*) por T (lo cual notamos $\mathcal{L}_m(T)$) al conjunto de palabras cuya corrida termina en un estado marcado. Formalmente, sea $w \in \mathcal{L}(T)$, entonces $w \in \mathcal{L}_m(T)$ si y solo si hay una corrida de w que comienza en el estado inicial $\bar{t}$ y alcanza un estado marcado $t_m \in M_T$, que notamos $\bar{t} \overset{w}{\dashrightarrow}_T t_m$.

Este último concepto es relevante para la definición de la composición paralela. Es un parte fundamental del trabajo para definir las palabras aceptadas por el problema `nonblocking` ya que el lenguaje aceptado por la planta compuesta es el mismo que el aceptado por cada uno de los componentes por separado.

Definición 2 (Composición Paralela): La *composición paralela* ($\|$) de dos autómatas T y Q es un operador simétrico y asociativo que produce un autómata $T\|Q = (S_T \times S_Q, A_T \cup A_Q, \rightarrow_{T\|Q}, \langle \bar{t}, \bar{q} \rangle, M_T \times M_Q)$, donde $\rightarrow_{T\|Q}$ es la menor relación que satisface las siguientes reglas (omitimos la versión simétrica de la primera regla):

$$\frac{t \xrightarrow{\ell}_T t'}{\langle t, q \rangle \xrightarrow{\ell}_{T\|Q} \langle t', q \rangle}\ell \in A_T \setminus A_Q \qquad \frac{t \xrightarrow{\ell}_T t' \quad q \xrightarrow{\ell}_Q q'}{\langle t, q \rangle \xrightarrow{\ell}_{T\|Q} \langle t', q' \rangle}\ell \in A_T \cap A_Q$$

Hay ciertas características de la composición paralela a destacar. En primer lugar, la última regla realiza la sincronización entre componentes. Segundo, que el número de estados en $T_0\|...\|T_n$ puede crecer exponencialmente con respecto al número de autómatas a componer. Tercero, que el lenguaje aceptado por la composición contiene las palabras que alcanzan estados marcados en todos los componentes *simultáneamente*.

2.1. Controlador

Dado un (conjunto de) autómata(s) y una partición de sus eventos en dos subconjuntos: *controlables* y *nocontrolables*, lo que buscamos es un *controlador* (director) que restrinja el vocabulario aceptado de forma de mantener un camino posible a los estados marcados del autómata.

Un controlador observa las transiciones no controlables y deshabilita alguna transiciones controlables para generar una planta restringida. Una palabra w pertenece al lenguaje generado por T restringido por una función del controlador $\sigma : A_T^* \mapsto 2^{A_T}$ (anotado como $\mathcal{L}^\sigma(T)$) si cada prefijo de w "sobrevive" a σ. Formalmente, sea $w = \ell_0, \ldots, \ell_k$ una palabra en $\mathcal{L}(T)$, entonces $w \in \mathcal{L}^\sigma(T)$ si y solo si para todo $0 \leq i \leq k$: $\bar{t} \xrightarrow{\ell_0 \ldots \ell_i}_T t_{i+1} \wedge \ell_i \in \sigma(\ell_0, \ldots, \ell_{i-1})$

Definición 3 (Problema de Control Safe y Non-Blocking): Un *Problema de Control* con objetivos *Safe* y *Non-Blocking* composicional es una tupla $\mathcal{E} = (E, A_E^C)$, donde E es un conjunto de autómatas $\{E_0, \ldots, E_n\}$ (podemos abusar la notación y usar $E = (S_E, A_E, \rightarrow_E, \bar{e}, M_E)$ para referirnos a la composición $E_0 \| \ldots \| E_n$), y $A_E^C \subseteq A_E$ es el conjunto de eventos controlables (i.e., $A_E^U = A_E \setminus A_E^C$ es el conjunto de eventos no controlables). Una solución para $\mathcal{E}$ es un supervisor $\sigma : A_E^* \mapsto 2^{A_E}$, tal que σ es:

- *Controlable*: $A_E^U \subseteq \sigma(w)$ con $w \in A_E^*$; y
- *Safe y Nonblocking*: para cada palabra $w \in \mathcal{L}^\sigma(E)$ existe una palabra no vacía $w' \in A_E^*$ tal que, la concatenación $ww' \in \mathcal{L}^\sigma(E)$ y $\bar{e} \xrightarrow{ww'}_E e_m$ con $e_m \in M_E$ (i.e., un estado marcado de E).

En resumen, un supervisor σ es controlable si solo deshabilita eventos controlables, y es safe y nonblocking si restringe el lenguaje generado a palabras que siempre puedan extenderse para llegar a un estado marcado. Notar que no es necesario efectivamente alcanzar un estado marcado, puede haber eventos no controlables que lo eviten, pero siempre debe haber una extensión válida para alcanzar tal estado marcado.

Sabiendo que el objetivo del problema es sintetizar un controlador, queremos distinguir estados *ganadores* (resp. *perdedores*), aquellos estados que podemos incluir (resp. no podemos incluir) en un controlador.

Notación 3: Decimos que un estado s es ganador["winning"] (resp. perdedor, errores, ["losing"]) en el problema $\mathcal{E} = (E, A_E^C)$ si hay (resp. no hay) una solución para (E_s, A_E^C) donde E_s es el resultado de cambiar el estado inicial de E a s. Nos referimos como controlador para s en E a una solución de (E_s, A_E^C). Nos referimos a los estados ganadores y perdedores de E cuando A_E^C es inferible del contexto, también usamos W_E y L_E para denotar el conjunto de estados ganadores y perdedores de $\mathcal{E}$.

Podemos pensar en un controlador non-blocking como un jugador optimista. Se encarga de no perder, y solo requiere conocer un futuro camino posible para llegar a un estado ganador.

Es importante notar que como se busca que cualquier palabra sea extensible a otro estado marcado, lo que se busca es pasar por algún estado marcado infinitas veces. Es decir, un estado 'e' marcado que tenga un camino para que el jugador pueda volver controlablemente al mismo estado 'e'.

Por esto, las estructuras claves que analizamos en nuestro algoritmo son los ciclos (*loops*), ya que los primeros estados ganadores son aquellos que están en un loop controlable con un estado marcado dentro. Luego anotamos como ganadores también a cualquier estado que controlablemente alcanza un estado ganador.

Los ciclos también son esenciales para encontrar los estados perdedores, ya que la única forma de que un estado sea perdedor es que no pueda alcanzar un estado ganador. En otras palabras, los estados perdedores son aquellos que forman parte de un loop que no tiene estados marcados ni transiciones salientes.

2.2. Director

En particular, buscamos como solución al problema de control, controladores que sean directores, como en [6, 7]. Un director se destaca por habilitar a lo sumo un evento controlable en cada punto de la ejecución.

Definición 4 (Director): Dado un controlador $\sigma : A_E^* \mapsto 2^{A_E}$ de un problema de control $\mathcal{E}$, decimos que σ es un director si $\forall w \in A_E^*$, $\|\sigma(w) \cap A_E^C\| \leq 1$.

Esto es en contraste con las soluciones tradicionales de Discrete Event Control y sus herramientas, como SUPREMICA [9] que presentan supervisores maximales. Los supervisores deben habilitar todos los eventos controlables que sean válidos en algún controlador que cumpla el objetivo del problema. Es decir que un director será un controlador que cumpla el mismo objetivo que un supervisor, pero restringiendo las palabras posibles a un subconjunto del lenguaje aceptado por el supervisor.

El foco en la construcción de directores tiene las siguientes razones:

- Los directores pueden ser más apropiados en contextos donde el controlador *ejecuta* las acciones controlables [7].
- La construcción de directores puede requerir una menor exploración de la planta que la construcción de un supervisor, ya que al encontrar un camino ganador a partir del estado s, puede evitar explorar otros eventos controlables de s. Esto se sinergiza y potencia las ganancias en tiempo de la técnica al explorar la composición on-the-fly y permite componer una proporción menor de la planta.
- En el caso de poder sintetizar un director explorando menos de la planta, se podría usar tanto para controlar la planta como para probar la controlabilidad de un problema donde herramientas de construcción de supervisores fallan por tener que explorar en mayor medida un problema de gran tamaño.
- Hay hasta la fecha una falta de herramientas disponibles para la síntesis de directores

Notar que en [7] se prueba que un director existe si y solo si un supervisor maximal existe.

2.3. Algoritmo monolítico

Una solución a este problema, anteriormente estudiada [4] se basa en un menor punto fijo. Simplemente se comienza con el conjunto de los estados que no tienen ningún camino

para alcanzar un estado marcado. Luego en cada iteración se agrega al conjunto de los estados perdedores todos aquellos que en un paso son forzados al conjunto de la iteración anterior, ya sea porque tienen una transición no controlable hacia dicho conjunto, o porque ninguna de sus transiciones lo evita.

Si al concluir el punto fijo el estado inicial no se encuentra en el conjunto entonces existe un controlador para el problema en cuestión y para construirlo se deben evitar las transiciones controlables que llevan al conjunto de estados perdedores.

Presentamos en el listing 2.1 una simplificación del algoritmo monólitico (que resuelve toda la planta ya compuesta).

```
Algorithm classicalSolver(E, A_E^C):
        B = {s ∈ S_E | ∄w . s -w->_E m ∧ m ∈ M_E}
        B' = ∅
        while B' ≠ B :
                B' = B
                B = B ∪ {s ∈ S_E | forcedTo(s, e, E) ∧ e ∈ B}
        return ē ∉ B

function forcedTo(s, Dest, Z):
        return  (∃ℓ_u ∈ A_Z^U . ∃e ∈ Dest . s -ℓ_u->_Z e) ∨
                (∀ℓ ∈ A_Z (s -ℓ->_Z e ⇒ e ∈ Dest))
```

Listing 2.1: Algoritmo Monolítico

Nuestro problema surge de que para el primer paso, encontrar el conjunto B inicial de estados que no alcanzan un marcado, necesitaríamos conocer los caminos que puede tomar cualquier estado, lo cual implica componer toda la planta.

Sin embargo, utilizamos un punto fijo que detecta errores en la función `findNewGoalsIn` ya que en ese momento no lo podemos evitar, y simplemente asumimos que lo no explorado no puede llegar a un estado marcado. Esto se discutirá en mayor profundidad en el capítulo 4.

3. EXPLORACIÓN ON-THE-FLY

El problema de síntesis de controlador ya tiene una solución clásica, por lo que la dificultad del trabajo no consistió en desarrollar un algoritmo que detectara estados ganadores y perdedores de una composición de LTS totalmente explorada.

El conflicto reside en que al componer distintos DES, la cantidad de estados de la composición es exponencial respecto de los estados en los componentes. Esto es de suma relevancia ya que la solución clásica, que compone toda la planta para luego explorarla, tiene un límite de escalabilidad en el cual la composición de la planta llega al límite de tiempo o memoria, y nunca se llega a la exploración.

Para combatir esto, la exploración on-the-fly, presentada en [2], clasifica estados como ganadores o perdedores durante la composición. Se espera que con esto sea posible, en primer lugar, cortar la exploración de una rama de la planta que ya se sabe que es perdedora o ganadora, reduciendo así la memoria y tiempo necesarios. Pero más aún, si el estado inicial fuera marcado como ganador o perdedor antes de la composición completa de la planta, ni siquiera sería necesario completar el proceso de composición.

En el listing 3.1 mostramos la estructura básica de este método. Trabajando sobre la parte de la planta compuesta hasta el momento (estructura explorada, ES), se expande ES consiguiendo nueva información hasta que sea seguro si el estado inicial es ganador o perdedor en E. Al llegar a esta conclusión se retorna el controlador para $\bar{e}$ en E o se notifica que no es controlable.

Definición 5 (Exploración Parcial): Sea $E = (S_E,\ A_E, \rightarrow_E, \bar{e}, M_E)$. Decimos que ES es una exploración parcial de E ($ES \subseteq E$) si $S_{ES} \subseteq S_E$ y $ES = (S_{ES}, A_E, \rightarrow_{ES}, \bar{e}, M_E \cap S_{ES})$, donde $\rightarrow_{ES} \subseteq (\rightarrow_E \cap (S_{ES} \times A_E \times S_{ES}))$. Escribimos $ES \subset E$ cuando $S_{ES} \subset S_E$.

```
Algorithm basicOTF-Exploration(E, A_E^C):
  ē = ⟨ē^0, ..., ē^n⟩
  ES = (S_ES, A_E, →_ES, ē, M_E ∩ S_ES) . ES ⊆ E
  until surelyWinsOrLoses(ē):
    expandES()
    computeNewWinnersAndLosers(ES)
  if initial ∈ Goals:
    return buildController(ē)
  else:
    return "UNREALIZABLE"
```

Listing 3.1: Algoritmo on-the-fly básico

Puede haber muchas variantes de cada una de estas partes, cómo se expande, cómo se computan nuevos ganadores y perdedores, etc. En particular, nuestro enfoque se muestra en el listing 3.2 y consiste en ir agregando una transición a la vez a la parte conocida de la planta, y en cada paso ver si esta nueva transición permite concluir que un estado es ganador o perdedor. Si algún nuevo estado se clasifica como ganador o perdedor, se propaga esta información a sus antecesores, posiblemente marcándolos a su vez como ganadores o perdedores, respectivamente.

A medida que exploramos mantenemos dos conjuntos de estados de ES (*Goals* y *Errors*) para los cuáles ya se tiene una conclusión, es decir, se sabe que son ganadores (o perdedores) en E.

Como se demostrará en el capítulo 4, al expandir ES con una transición (e, ℓ, e') a la vez, a menos que e' ya fuera un ganador/perdedor entonces solo puede haber nueva información si existe un loop entre e y e'. Es más, si hay un nuevo estado ganador/perdedor, entonces e seguramente es también un nuevo ganador/perdedor, y todos los nuevos estados clasificados son antecesores de e. Ésto permite optimizar la detección de nuevos ganadores/perdedores en lugar de ejecutar un algoritmo clásico sobre todo ES.

Para explicar el algoritmo y argumentar su correctitud y completitud introducimos dos nuevos problemas de control para exploraciones parciales. Uno toma una visión optimista de la región no explorada ($\top$) asumiendo que todas las transiciones no exploradas llevan a un estado ganador. El otro toma una visión pesimista ($\bot$) asumiendo que las transiciones no exploradas llevan a estados perdedores.

Definición 6 (Problemas de Control $\top$ y $\bot$): Sean $\mathcal{E} = (E, A_E^C)$, $E = (S_E, A_E, \rightarrow_E, \bar{e}, M_E)$ y $ES = (S_{ES}, A_E, \rightarrow_{ES}, \bar{e}, M_E \cap S_{ES})$, y $ES \subseteq E$.
Definimos $\mathcal{E}_\top$ como $(ES_\top, A_E^C)$ donde $ES_\top = (S_{ES} \cup \{\top\}, A_E, \rightarrow_\top, \bar{e}, (M_E \cap S_{ES}) \cup \{\top\})$ y $\rightarrow_\top = \rightarrow_{ES} \cup \{(s, \ell, \top) \mid \exists s' \,.\, (s, \ell, s') \in (\rightarrow_E \setminus \rightarrow_{ES})\} \cup \{(\top, \ell, \top) \mid \ell \in A_E\}$
Definimos $\mathcal{E}_\bot$ como $(ES_\bot, A_E^C)$ donde $ES_\bot = (S_{ES} \cup \{\bot\}, A_E, \rightarrow_\bot, \bar{e}, M_E \cap S_{ES})$ y $\rightarrow_\bot = \rightarrow_{ES} \cup \{(s, \ell, \bot) \mid \exists s' \,.\, (s, \ell, s') \in (\rightarrow_E \setminus \rightarrow_{ES})\}$

Usamos estos problemas de control para decidir tempranamente si un estado s es ganador o perdedor en E basado en lo que exploramos previamente en ES. Si s es ganador en $ES_\bot$ esto significa que sin importar a dónde lleven las transiciones no exploradas, s también va a ser ganador en E. Similarmente, s es perdedor en E si es perdedor en $ES_\top$. El Lema 1 refuerza este razonamiento.

Lema 1: ***(Monotonicidad de $W_{ES_\bot}$ y $L_{ES_\top}$)*** Sean ES y ES' dos exploraciones parciales de E tal que $ES \subset ES'$ entonces $W_{ES_\bot} \subseteq W_{ES'_\bot}$ y $L_{ES_\top} \subseteq L_{ES'_\top}$.

El algoritmo agrega iterativamente una transición de E a ES a la vez y asegura que al final de cada iteración, los estados en ES están correcta y completamente clasificados en ganadores y perdedores si hay suficiente información de E en ES. Los conjuntos de estados $Errors$, $Goals$ y $None$ se usan para este propósito.

En el peor caso, si no se pudo concluir nada antes de componer la planta en su totalidad, se perdió tiempo en los puntos fijos, intentando clasificar estados, y se realiza una última vez el algoritmo clásico con la planta totalmente explorada. Esto garantiza la completitud del algoritmo, como se detalla en mayor profundidad en el capítulo 4.

Para explorar en el orden más conveniente, y componer una menor parte de la planta, se utiliza una heurística de exploración Best First Search [2]. La misma busca ganar controlablemente o perder no controlablemente, para garantizar con la menor exploración posible que el estado actual es ganador o perdedor.

Una heurística no presenta garantía de explorar en la dirección correcta, más bien da una recomendación. Son ampliamante utilizadas al optimizar, pero es importante que la correctitud de los algoritmos no dependan de estas recomendaciones, ya que por su misma naturaleza no tienen garantías fuertes.

```
Algorithm genericOTF-Exploration(E, A_E^C):
  ē = ⟨ē^0, ..., ē^n⟩
  Goals = Errors = ∅
  ES = initial //la parte conocida de la planta
  while initial ∉ Goals ∪ Errors:
    (e, ℓ, e') =nextTransition(ES, heuristica)
    expandES(ES, (e, ℓ, e'))
    if e' ∈ Errors:
      propagateError(e')
    else if e' ∈ Goals:
      propagateGoal(e')
    else if isLoop(e, e'):
      if newWinningLoop(e, e'):
        propagateGoal(e')
      else if newLosingLoop(e, e'):
        propagateError(e')

  if initial ∈ Goals:
    return buildController(Goals)
  else:
    return "UNREALIZABLE"
```

Listing 3.2: Nuestro enfoque on-the-fly

3.1. Agnosticismo a la heurística

Una distinción clave del algoritmo *on-the-fly* es que está dividido en dos partes. Por un lado se tiene el algoritmo de exploración responsable de que al final se llegue al resultado correcto, por el otro tenemos una heurística que le brinda la próxima transición a explorar. Ese algoritmo de exploración no puede depender de la heurística, ya que la misma no garantiza siempre elegir el mejor camino posible, sino solo la mejor aproximación que encuentre. Uno de los focos de nuestro trabajo fue en esa corrección independiente del orden de exploración de las transiciones.

El proyecto `MTSA` inicialmente contaba con dos heurísticas `Best First Search` para exploración, *Monotonic Abstraction* y *Ready Abstraction*.

El inconveniente con la versión anterior del algoritmo de exploración es que había sido desarrollado en conjunto con las heurísticas. Si bien esto ayudaba a la eficiencia del mismo, generaba una dependencia del orden de observación de las transiciones, dando resultados erróneos al cambiar las recomendaciones. El nuevo enfoque no depende de la forma de explorar, por ende, da una mayor libertad de investigar a futuro nuevos criterios de evaluación para mejorar la eficiencia de la técnica sin comprometer corrección ni completitud. Esto fue útil durante el desarrollo del trabajo, ya que facilitó la inclusión de nuestras nuevas heurísticas (`Dummy` y `Breadth First Search`, ver sección 5.2) para la experimentación.

3.2. Ejemplo de ejecución

Para ilustrar el funcionamiento y ventajas de la exploración on-the-fly, mostramos una ejecución posible sobre una planta ejemplo. Supongamos que dados ciertos autómatas calculamos su composición y llegamos a la planta de la figura 3.1. Las transiciones se van a explorar en orden de menor a mayor, en el diagrama las (u)/(c) son no controlables/-controlables respectivamente y el estado marcado tiene doble círculo. No especificamos la controlabilidad de todas las transiciones de manera de hacer más legible el gráfico y

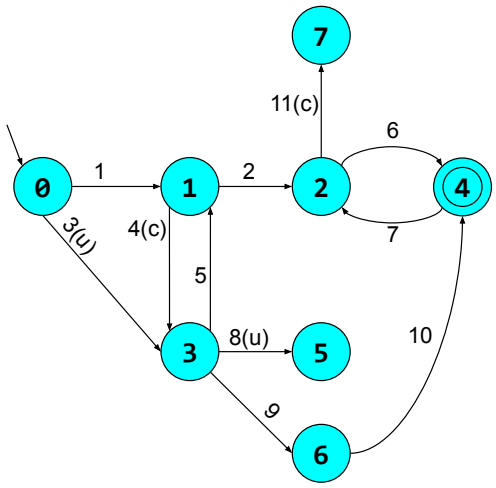


Fig. 3.1: Planta compuesta completa

además porque, mirando las definidas, no nos modifica la conclusión el hecho de que sean o no controlables.

Al iniciar el algoritmo no vemos la planta completa sino sólo el primer estado **0**, desde el mismo expandimos la primer transición, en este caso *1*. Como llega a un estado sin explorar **1**, y que no es deadlock, no hay información para conseguir. Entonces mientras lleguemos a "algo nuevo" no tenemos nada para hacer más que seguir explorando. Incluso si llegamos a un estado explorado (no deadlock) no va a haber nueva información si no tenemos un loop. Veremos más en detalle y con demostración por qué esto es así en el capítulo 4; pero la intuición es que no va a haber nuevos ganadores puesto que un ganador necesita llegar a un marcado infinitas veces, por ende necesita como mínimo que haya un loop. Entonces hasta ahora llevamos explorado lo mostrado en la figura 3.2.

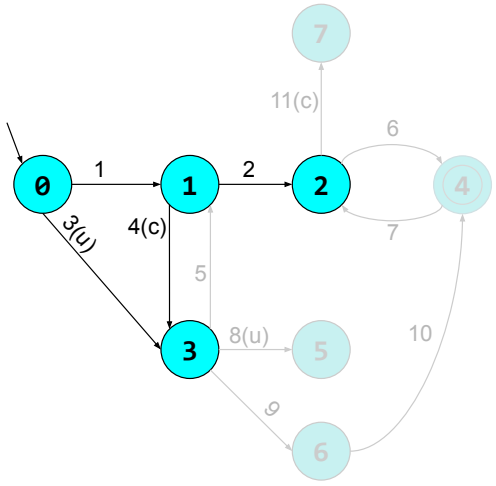


Fig. 3.2: Primeros pasos de exploración

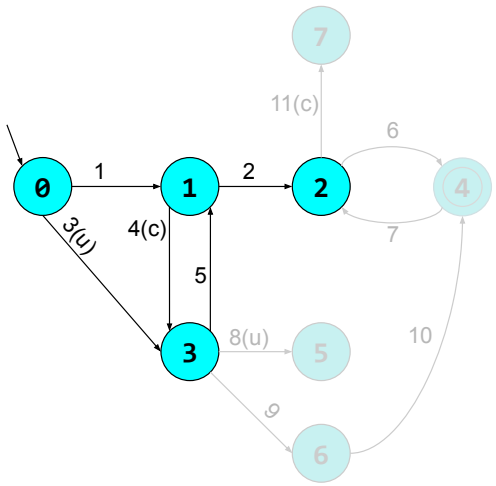


Fig. 3.3: Hay loop sin conclusiones nuevas

Si seguimos explorando por la transición *5* cerramos un loop [**1**, **3**], pero **3** puede ser forzado a algo no explorado usando la transición no controlable *8*, entonces todavía no podemos sacar conclusión alguna. Exploramos hasta figura 3.3.

Continuamos y llegamos a un estado marcado **4**, si fuese sólo alcanzar un marcado podríamos decir que es ganador pero necesitamos siempre poder llegar a uno (incluso desde sí mismo), por ende ahora sigue sin ser nada. Luego vemos la transición *7*, cerrando loop [**2**, **4**]. Éste loop tiene un estado marcado y **2** no puede ser forzado a otra cosa, ya que el resto de sus transiciones son controlables; por ende obtenemos nuestros primeros ganadores (**2** y **4**). A continuación debemos propagar esta nueva información llegando a marcar **1** como ganador, porque puede controlablemente llegar al loop [**2**, **4**]. Lamentablemente todavía no hay conclusión sobre el estado inicial debido a que puede ser forzado a algo sin explorar. Ver figura 3.4.

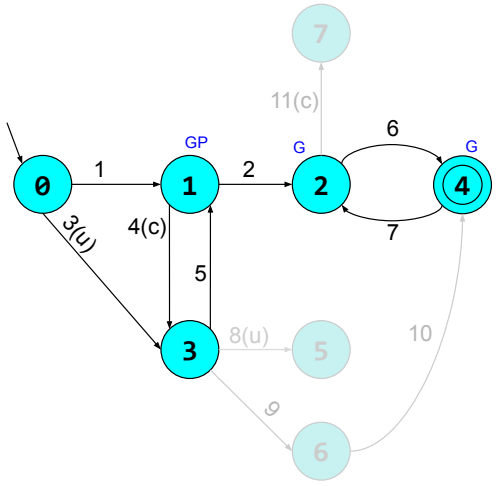


Fig. 3.4: Se encuentra Goal(G) y propaga(GP)

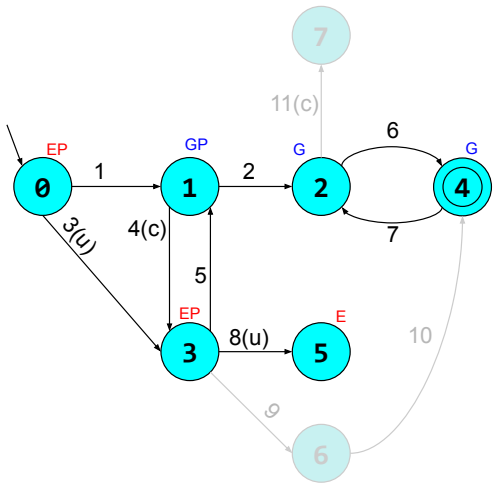


Fig. 3.5: Se encuentra Error(E) y propaga(EP) hasta inicial, dejando de explorar

Proseguimos con la transición *8* y llegamos a explorar el estado **5**, que es un deadlock por ende es error directamente. Propagando esta información marcamos los estados **3** y **0** como errores, llegando así a una conclusión sobre el inicial. Entonces ya no nos es necesario seguir explorando y se retorna que *no existe controlador para la planta*. El estado de exploración final de la planta se ve en la figura 3.5.

Notar que no vemos las demás transiciones de los estados **2** ni **3** ya que, en el caso de **2** gana por alguna y el resto son controlables, y en el caso de **3** pierde por una no controlable. Es decir, **2** puede controlablemente ganar y **3** es forzado a perder sin importar la forma del resto.

Si la transición *8* fuese controlable podríamos evitar el error y seguiríamos explorando, encontrando el estado **6**, su conexión al loop ganador y propagando ganador hasta el estado inicial. Notar que ésta sería una planta *diferente*, controlable a diferencia de la propuesta en el ejemplo.

4. SINTESIS DE CONTROLADORES DIRIGIDA

En esta sección presentamos el nuevo algoritmo SCD, que realiza una exploración sobre la marcha del espacio de estados. Por medio de dicha exploración el algoritmo encuentra un director que es solución para el problema dado de control composicional. También discutimos la correctitud y completitud del nuevo algoritmo SCD, demostramos los lemas presentados y detallamos la complejidad de nuesstro algoritmo.

4.1. Propuesta de nuevo algoritmo

```
1  function SCD(E=(E, A_E^C), heuristic):
2    ē = ⟨ē^0, ..., ē^n⟩
3    ES = ({ē}, A_E, ∅, ē, M_E ∩ {ē})
4    Goals = Errors = Witnesses = ∅
5    None = {ē}
6    if (isDeadlock(ē)):
7    Errors = {ē}
8    None = ∅
9    while ē ∉ Errors ∪ Goals:
10     (e, ℓ, e') = expandNext(heuristic)
11     S_ES' = S_ES ∪ {e'}
12     ES' = (S_ES', A_E, →_ES ∪{e -ℓ-> e'}, ē, M_E ∩ S_ES')
13     if e' ∈ Errors:
14       propagateError({e'})
15     else if e' ∈ Goals:
16       propagateGoal({e'})
17     else if canReach(e, e', ES):
18       loops = getMaxLoop(e,e')
19       if canBeWinningLoop(loops):
20         C = findNewGoalsIn(loops)
21         Witnesses = Witnesses ∪ (C ∩ M_ES')
22         Goals = Goals ∪ C
23         None = None \ C
24         propagateGoal(C)
25       else:
26         P = findNewErrorsIn(loops)
27         Errors = Errors ∪ P
28         None = None \ P
29         propagateError(P)
30     ES = ES'
31
32   if ē ∈ Goals:
33     r = rankStates(ES)
34     return λw . {ℓ | ē -w->*_ES e -ℓ->_ES e' ∧ e' ∈ Goals
35     ∧(l ∈ A_E^C ⇒ ℓ = bestControllable(s, r, ES))}
36   else:
37     return UNREALIZABLE
```

Listing 4.1: Algoritmo de exploración dirigida on-the-fly.

```
function propagateGoal(newGoals):
  C' = ∅; C = ancestorsNone(newGoals)
  while C' ≠ C:
    C' = C
    C = C \ {s ∈ C |
      forcedTo(s, S_ES'⊥ \ (C ∪ Goals), ES'⊥) ∨
      cannotReachGoalIn(s, C)}
  Goals = Goals ∪ C
  None = None \ C

procedure propagateError(newErrors):
  P = ancestorsNone(newErrors)
  C = P; C' = ∅
  while C' ≠ C:
    C' = C
    C = C \ {s ∈ C |
      forcedTo(s, Errors, ES'⊤) ∨
      cannotReachGoalIn(s, C)}
  P = P \ C
  Errors = Errors ∪ P
  None = None \ P
```

Listing 4.2: Algoritmos de propagación.

```
function findNewGoalsIn(loops):
  C = loops; C' = ∅
  while C' ≠ C:
    C' = C; C'' = ∅
    while C'' ≠ C:
      C'' = C
      C = C \ {s ∈ C |
            forcedTo(s, S_ES'⊥ \ (C ∪ Goals), ES'⊥) ∨
            cannotBeReached(s, C) }
    C = C \ {s ∈ C | cannotReachGoalOrMarkedIn(s, C)}
  return C

function findNewErrorsIn(loops):
  if (∃s ∈ loops . s -ℓ'→_ES'⊤ s' ∧ (s' ∉ loops ∧ s' ∉ Errors)):
    return ∅
  else:
    return loops
```

Listing 4.3: Confirmación de clasificaciones

```
procedure expandNext(heuristic):
  let (e, ℓ, e') . e ∈ S_ES ∧ e -ℓ→_E e' ∧ ¬e -ℓ→_ES e' ∧ e ∈ None∧
       ∀(s, ℓ', s') . s ∈ S_ES ∧ s -ℓ→_E s' ∧ ¬s -ℓ→_ES s' ∧ s ∈ None
          ⟹ heuristic(e, ℓ, e') >= heuristic(s, ℓ', s'))
  if isDeadlock(e'):
    Errors = Errors ∪ {e'}
  if e' ∉ Errors ∪ Goals:
    None = None ∪ {e'}
  return (e, ℓ, e')

function ancestorsNone(targets):
  return { e ∈ ES' | ∃e' ∈ targets . ∃w . e -w-···→_ES' e'∧
          ∄s ∈ w(e) . s ≠ e'∧ ∈ Goals ∪ Errors}

function canBeWinningLoop(loop):
  return (∃e_m ∈ loop . e_m ∈ M_ES')∨
          (∃s ∈ loop . canReachInOneStep(s, ES, Goals))

function getMaxLoop(e,e'):
  return {s | ∃w, w' . e -w-···→_ES' s ∧ s -w'-···→_ES' e'∧
          ∄s' . (s' ∈ w(e) ∨ s' ∈ w'(s)) ∧ s' ≠ e' ∧ s' ∈ Goals ∪ Errors}

function forcedTo(s, Dest, Z):
  return (∃ℓ_u ∈ A_Z^U . ∃e ∈ Dest . s -ℓ_u→_Z e) ∨
          (∀ℓ ∈ A_Z (s -ℓ→_Z e ⇒ e ∈ Dest))

function cannotBeReached(s, C):
   return ∄s' ∈ C, ∃ℓ . s' -ℓ→_ES' s

function cannotReachGoalOrMarkedIn(s, C)
   return ∄w . s -w-···→_C s' ∧ s' ∈ C ∧
          (canReachInOneStep(s', ES, Goals)
          ∨ s' ∈ M_ES')

function cannotReachGoalIn(s, C)
   return ∄w . s -w-···→_C s' ∧ s' ∈ C ∧
          canReachInOneStep(s', ES, Goals)

function canReach(s', s)
   return ∃w . s' -w-···→_ES' s

function canReachInOneStep(s, Targets)
   return ∃l . s -ℓ→_ES' s' ∧ s' ∈ Targets

function isDeadlock(s)
   return ∄l . s -ℓ→_E s'
```

Listing 4.4: Métodos auxiliares

```
function rankStates(ES)
    r = 0; W' = Witnesses; W = ∅
    while W' ≠ W :
      ∀w ∈ W',rank(w) = r
      W = W ∪ W'
      W' = {s ∈ (Goals \ W) | ∃s' ∈ W . s →_ES s'}
      r = r + 1
    return rank

function bestControllable(e, r, ES)
    return ℓ ∈ A_E^C . e -ℓ->_ES e' ∧ ∄ℓ' ∈ A_E^C .
                                e -ℓ'->_ES e'', r(e'') ≤ r(e')
```

Listing 4.5: Métodos de ranking

4.2. Demostración de correctitud y completitud

Propiedad 1 (Invariante): El loop principal del Algorithm 4.1 tiene el siguiente invariante: $ES \subseteq E \wedge \forall s \in ES \;.\; (s \in \mathit{Goals} \Leftrightarrow s \in W_{ES_\bot}) \wedge (s \in \mathit{Errors} \Leftrightarrow s \in L_{ES_\top}) \wedge s \in \mathit{Errors} \uplus \mathit{Goals} \uplus \mathit{None}$

La explicación del algoritmo 4.1 que detallamos a continuación sirve también como un esquema de demostración para la propiedad 1.

Para empezar, notar que la función `expandNext` (line 10) retorna una nueva transición $e \xrightarrow{\ell}_E e'$ garantizando que e ya se encontraba en ES y $e \in \mathit{None}$. Esto significa que en cada iteración, hay algo de información nueva disponible para un estado que actualmente no está clasificado en ganador ni perdedor.

Si el estado e' ya es clasificado como ganador en $ES_\bot$ (line 15) o perdedor en $ES_\top$ (line 13) entonces esta información necesita ser propagada a los estados en None para ver si pueden convertirse en ganadores en ${ES'}_\bot$ o perdedores en ${ES'}_\top$. Tanto `propagateGoal` como `propagateError` realizan un punto fijo estándar [13] sobre $ES_\bot$ y $ES_\top$ pero solo sobre predecesores de e' que están en None. El Lema 2 asegura la completitud de esta propagación restringida.

Lema 2: ***(Ganadores/Perdedores nuevos tienen camino de estados-None a transición nueva)*** Sea la transición $e \xrightarrow{\ell}_{ES} e'$ la única diferencia entre dos exploraciones parciales, ES y ES', de E. Si $s \notin (W_{ES_\bot} \cup L_{ES_\top})$ y $s \in (W_{{ES'}_\bot} \cup L_{{ES'}_\top})$, entonces hay $s_0, \ldots, s_n \notin (W_{ES_\bot} \cup L_{ES_\top})$ tal que $s = s_0 \wedge s_0 \xrightarrow{\ell_0}_{ES} \ldots s_n \xrightarrow{\ell}_{ES} e'$.

Ya en la linea 17 sabemos que e' no es ganador en $ES_\bot$ ni perdedor en $ES_\top$, determinamos si $e \xrightarrow{\ell}_{ES} e'$ cierra un nuevo loop al chequear si e' puede alcanzar a e. Si no es el caso, entonces no hay nada que hacer ya que e' alcanza las mismas transiciones en ES' que en ES (o es un estado nuevo cuyas transiciones salientes no están exploradas). Entonces, $e' \notin ({W_{ES'}}_\bot \cup {L_{ES'}}_\top)$ ya que cualquier controlador para e' en ${ES'}_\bot$ (resp. ${ES'}_\top$) es también un controlador en $ES_\bot$ (resp. $ES_\top$) y viceversa. Más aún, que no haya nueva información para e' implica que no hay nuevos ganadores o perdedores (Lemma 3)

Lema 3: ***(Nuevos ganadores/perdedores solo si e' es un nuevo ganador/perdedor)*** Sea $e \xrightarrow{\ell} e'$ la única diferencia entre dos exploraciones parciales, ES y ES', y $e' \notin L_{ES_\top} \cup W_{ES_\bot}$. Si ${W_{ES'}}_\bot \neq W_{ES_\bot}$ entonces $e' \in {W_{ES'}}_\bot \setminus W_{ES_\bot}$, y si ${L_{ES'}}_\top \neq L_{ES_\top}$ entonces $e' \in {L_{ES'}}_\top \setminus L_{ES_\top}$.

Si se cerró un nuevo loop(line 17), por Lemma 3 alcanza con analizar si $e' \in {W_{ES'}}_\bot \uplus {L_{ES'}}_\top$, y por Lemma 2 propagar cualquier información nueva de e' a sus predecesores.

En la linea 18 computamos *loops*, el conjunto de estados que pertenecen a un loop que pasa por $e \xrightarrow{\ell}_{ES'} e'$ y nunca por $W_{ES_\bot} \cup L_{ES_\top}$. Intuitivamente, cualquier controlador para e' va a depender de alguno de estos loops. O, en términos del Lemma 2, para que e' cambie su estado, debe ser a través de un camino de estados None.

En la linea 19 usamos `canBeWinningLoop`(*loops*) para chequear si existe algún estado marcado en *loops* o si es posible escapar de *loops* y alcanzar un *goal* en un paso. Esto distingue entre dos posibles opciones: $e' \in {W_{ES'}}_\bot$ o $e' \in {L_{ES'}}_\top$ (ver Lemma 4).

Lema 4: ***(Condición necesaria/suficiente para ganar/perder)*** Sea $e \xrightarrow{\ell} e'$ la única diferencia entre dos exploraciones parciales, ES y ES'. Sea $loops =$ `getMaxLoop(e, e')`. Si $e' \in W_{ES'_\perp} \setminus W_{ES_\perp}$ entonces `canBeWinningLoop(loops)`. Además, si `canBeWinningLoop(loops)` entonces $e' \notin L_{ES'_\top}$

Si `canBeWinningLoop()` retorna true, en la linea 20, sabemos que si e' cambia su estado es porque $e' \in W_{ES'_\perp}$. Para ver si este cambio se produce, se realiza una computación de punto fijo basada en la solución del problema monolítico (2.3). Sin embargo, el método `findNewGoalsIn` aplica una optimización basada en Lemma 2; solo considera estados que están en un *None*-loop a través de la nueva transición (*loops*).

Si `canBeWinningLoop()` retorna false, entonces debemos comprobar si $e' \in L_{ES'_\top}$. Esto puede hacerse de forma más eficiente que con un punto fijo usando el Lemma 5 que muestra que alcanza con observar si no es posible escapar de *loops* alcanzando en un paso un estado que no esté en $L_{ES_\top}$.

Lema 5: ***(findNewErrorsIn es correcto y completo)*** Si $loops =$ `getMaxLoop(e, e')` $\wedge \neg$ `canBeWinningLoop`$(loops)$ y $P =$ `findNewErrorsIn`$(loops)$ entonces $(e' \in L_{ES'_\top} \Rightarrow e' \in P \subseteq L_{ES'_\top}) \wedge (e' \notin L_{ES'_\top} \Rightarrow P = \emptyset)$

Por motivos de eficiencia, `findNewGoalsIn` y `findNewErrorsIn` no solo verifican si $e' \in W_{ES'_\perp}/e' \in L_{ES'_\top}$ sino que también agregan estados ganadores/perdedores cuando pueden. La detección completa de nuevos estados ganadores y perdedores se hace finalmente con los procesos de propagación.

Habiendo argumentado que la propiedad 1 es válida, la correctitud y completitud se desprenden de las siguientes observaciones: *i*) El `main loop` termina cuando logró determinar que $\bar{e}$ se encuentra en $L_{ES'_\top}$ o $W_{ES'_\perp}$. Esto en peor caso eventualmente sucede cuando $ES = E$. *ii*) Las lineas 33 a 35 extraen un director del conjunto de estados ganadoes (*Goals*). Usamos el mismo punto fijo descripto en el Algorithm 4 de [7] para computar un ranking (line 33) y una función determinística (line 35) que retorna cuál transición controlable debe ser habilitada, cumpliendo la condición de un director.

Teorema 1 (Correctitud y Completitud): Sea $\mathcal{E} = (E, A_E^C)$ un problema de control composicional según la Definición 3. Existe una solución para $\mathcal{E}$ si y solo si el algoritmo SCD retorna un director para $\mathcal{E}$.

Demostración (Correctitud y completitud): El teorema se desprende del invariante de ciclo del algoritmo (Definition 1), el Lemma 1, la correctitud del algoritmo 4 de [7] y el hecho de que en el peor caso todas las transiciones son agregadas a la estructura de exploración. Entonces, $E = ES = ES_\perp = ES_\top$.

4.3. Demostración de Lemas

Demostración Lemma 1: (Idea: Para probar $W_{ES_\perp} \subseteq W_{ES'_\perp}$ mostramos que un controlador para un estado s en $W_{ES_\perp}$ puede ser usado como un controlador para s en $W_{ES'_\perp}$. Para $L_{ES_\top} \subseteq L_{ES'_\top}$, asumimos que hay un estado $s \in L_{ES_\top} \setminus L_{ES'_\top}$. Llegamos a una contradicción mostrando que el controlador que s debe tener en $ES'_\top$ es también un controlador para s en $ES_\top$.)

Si $s \in W_{ES_\perp}$ entonces existe un controlador σ para el problema de control $ES_\perp$. Sea Z tal que $ES \subseteq Z$. Demostraremos que σ es un controlador para $Z_\perp$. Esto requiere dos condiciones según la Definición 3. La primera, que σ es controlable, es trivial ya que los conjuntos de eventos controlables y no controlables no fueron cambiados.

Para la segunda, nonblocking, primero mostramos que $\mathcal{L}^\sigma(Z_\perp) = \mathcal{L}^\sigma(ES_\perp)$.

Si asumimos que $\mathcal{L}^\sigma(Z_\perp) \not\subseteq \mathcal{L}^\sigma(ES_\perp)$ y $w \in \mathcal{L}^\sigma(Z_\perp) \setminus \mathcal{L}^\sigma(ES_\perp)$, la corrida que verifica w debe permanecer siempre en Z o alcanzar eventualmente un estado *deadlock* en $Z_\perp$. En cualquier caso, sea w_0 el prefijo más largo en ES. Sabemos que w_0 es un prefijo no vacío de w. Sea ℓ tal que $w_0.\ell$ es un prefijo de w. Por la definición de $ES_\perp$, $w_0.\ell$ alcanza un estado *deadlock* en $ES_\perp$. Esto es una contradicción, ya que σ es un controlador para $ES_\perp$.

Para mostrar que $\mathcal{L}^\sigma(Z_\perp) \supseteq \mathcal{L}^\sigma(ES_\perp)$, asumimos que $w \in \mathcal{L}^\sigma(ES_\perp)$. Si w también está en $\mathcal{L}^\sigma(ES)$ entonces debe pertenecer a $\mathcal{L}^\sigma(Z)$ y $\mathcal{L}^\sigma(Z_\perp)$. De otra forma, $w = w_0.\ell$ alcanza un estado *deadlock* en $ES_\perp$. Como w_0 pertenece a $\mathcal{L}^\sigma(ES)$, debe pertenecer también a $\mathcal{L}^\sigma(Z)$. Consideramos el estado s alcanzado por w_0 en E, debe tener una transición etiquetada como ℓ para justificar su inclusión en $ES_\perp$. En Z, el estado s o tiene la transición y por lo tanto $w_0.\ell \in \mathcal{L}^\sigma(Z) \subseteq \mathcal{L}^\sigma(Z_\perp)$, o no tiene la transición, pero el estado en $Z_\perp$ tiene una transición ℓ a un estado *deadlock*, por lo tanto $w_0.\ell \in L^\sigma(Z_\perp)$.

Ahora, sabiendo que $\mathcal{L}^\sigma(Z_\perp) = \mathcal{L}^\sigma(ES_\perp)$, procedemos a *nonblocking*. Sea una palabra $w \in \mathcal{L}^\sigma(Z_\perp)$ que no puede ser extendida con w' tal que $w.w'$ se encuentra en $L^\sigma(Z_\perp)$ y alcanza un estado marcado de $Z_\perp$. Como w también se encuentra en $L^\sigma(ES_\perp)$ entonces, como σ es un controlador para $ES_\perp$, existe un w' tal que $w.w' \in L^\sigma(ES_\perp) = L^\sigma(Z_\perp)$ y alcanza un estado marcado. Notar que la corrida para $w.w'$ siempre se encuentra en ES, lo que significa que la corrida también está en $Z_\perp$. Finalmente llegamos a una contradicción.

Para demostrar que $L_{ES_\top} \subseteq L_{ES'_\top}$, asumimos que existe un estado $s \in L_{ES_\top} \setminus L_{ES'_\top}$. Como $s \notin L_{ES'_\top}$, tiene un controlador σ en $ES'_\top$, pero $s \in L_{ES_\top}$ por lo que no puede existir un controlador válido σ' para s en $ES_\top$. Esto es falso, más aún, mostraremos que si σ es un controlador para s en $ES'_\top$, entonces hay un controlador válido σ' para s en cualquier $Z_\top$ si $Z \subseteq ES$.

σ es un controlador valido en $ES'_\top$, por lo que cualquier palabra en $\mathcal{L}^\sigma(ES'_\top)$ puede ser extendida para alcanzar un estado marcado. Solo hay una cantidad finita de estados en $ES'_\top$, por lo que deben existir w' y w'' tal que $w.w'.w'' \in \mathcal{L}^\sigma(ES'_\top)$, $w.w'$ llega a un estado marcado, y $w.w'.w''$ llega al mismo estado que $w.w'$.

Si $w.w'.w''$ está en $\mathcal{L}^\sigma(Z)$, entonces no hay nada que hacer, es claro que σ tiene la misma forma de extender w en $Z_\top$. Si no, notemos que $w.w'.w'' = w_0.l.w_1$ tal que w_0 es el prefijo más largo de $w.w'.w''$ en $\mathcal{L}^\sigma(Z)$, esto significa que $w_0.l$ alcanza el estado marcado ganador $\top$, y desde ahí toda extensión de la palabra solo puede permanecer en ese mismo estado, por lo tanto, σ también es un controlador válido en $Z_\top$.

□

Demostración Lemma 2: (Idea: Si s no es un predecesor de e', como $e \xrightarrow{l}_{ES'} e'$ es la única diferencia entre ES y ES', entonces los descendientes de s son los mismos, por lo tanto sus posibles controladores en $ES'_\top$ y $ES'_\perp$ no cambiaron. Entonces, $s \notin W_{ES'_\perp} \cup L_{ES'_\top}$ lo cual es una contradicción.

Como paso siguiente probamos que hay al menos un camino desde s a e' a través de estados *None* por contradicción asumiendo que todos los caminos a e' en ES' atraviesan

un estado $s' \in (W_{ES_\perp} \cup L_{ES_\top})$. Como $s \notin L_{ES_\top}$, hay un controlador σ desde s en $ES_\top$. σ no va a alcanzar estados en $L_{ES_\top}$, luego para todo s' que alcance, debe haber un controlador $\sigma_{s'}$ para $ES_\perp$ (por lo tanto $\sigma_{s'}$ evita la nueva transición ℓ). Usamos σ y $\sigma_{s'}$ para construir un controlador para s en ${ES'}_\top$ para mostrar que $s \notin L_{{ES'}_\top}$. Un controlador para s en ${ES'}_\perp$ no puede existir porque de otra forma podríamos usarlo para construir un controlador para s en $ES_\perp$ usando un razonamiento similar al anterior. Esto significa que $s \in W_{ES_\perp}$ contradiciendo la hipótesis.)

Si un estado s no se encuentra en $W_{ES_\perp} \cup L_{ES_\top}$ es porque tiene un controlador σ en $ES_\top$ pero no tiene uno para $ES_\perp$. Esto depende únicamente de los descendientes de s, dado que estos son los únicos estados que σ puede alcanzar. Si s no es un predecesor de e', y $e \xrightarrow{l}_{ES'} e'$ es la única diferencia entre ES y ES' entonces los descendientes de s son los mismos, por lo que los posibles controladores no tuvieron ningún cambio, y s sigue siendo NONE.

Lo que no es tan claro, es que s no tiene nuevos controladores posibles si tiene un camino que puede alcanzar e' pero solo pasando por al menos un estado de $W_{ES_\perp} \cup L_{ES_\top}$. Asumiendo que debe pasar por estados en $W_{ES_\perp} \cup L_{ES_\top}$ mostramos que:

- Sabiendo que s tenía un controlador σ en $ES_\top$, mostramos que s tiene un controlador válido σ' en ${ES'}_\top$:

  $\sigma'(w) = \sigma(w)$ si no existe un w_0 sufijo de w tal que $s \xrightarrow{w_0}_{ES'} s_i \wedge s_i \in L_{ES_\top} \cup W_{ES_\perp}$.

  $\sigma'(w) = \sigma_{s_i}(w_1)$ donde w_0 es el sufijo más corto de $w = w_0.w_1$ tal que $s \xrightarrow{w_0}_{ES'} s_i \wedge s_i \in W_{ES_\perp}$. σ_{s_i} es el controlador que sabemos que s_i tiene en $ES_\perp$ ya que $s_i \in W_{ES_\perp}$, y que cada controlador válido en $ES_\perp$ es también válido en $ES_\top$.

  Como σ es un controlador válido, sabemos que no puede alcanzar estados en $L_{ES_\top}$.

  Finalmente, es claro que σ' es un controlador válido para s en ${ES'}_\top$. Notar que σ' no depende de la nueva transición.

- Sabiendo que s no tiene controlador en $ES_\perp$, mostramos que s no tiene controlador en ${ES'}_\perp$ asumiendo que tiene uno y llegando a una contradicción:

  Suponemos que existe un controlador σ' para s en ${ES'}_\perp$, y que $e \xrightarrow{l}_{ES'} e'$ es la única diferencia entre ES y ES'.

  Con σ' construimos σ, un controlador para s en $ES_\perp$.

  $\sigma(w) = \sigma'(w)$ si no existe un w_0 que sea prefijo de w y que $s \xrightarrow{w_0}_{ES} s_i \wedge s_i \in W_{ES_\perp} \cup L_{ES_\top}$. Como σ' es un controlador válido, sabemos que no puede alcanzar estados en $L_{{ES'}_\top}$. Notar que w no puede alcanzar $e \xrightarrow{\ell}_{ES'} e'$ porque s no tiene un camino de estados *None* a e'.

  Si $w = w_0.w_1$ y $s \xrightarrow{w_0}_{ES} s' \wedge s' \in W_{ES_\perp}$ entonces $\sigma(w_0.w_1) = \sigma_{s'}(w_1)$ donde $\sigma_{s'}$ es el controlador para s' en $ES_\perp$. Notar que una vez que se alcanza s' siempre seguimos $\sigma_{s'}$.

  Como σ nunca alcanza la nueva transición sabemos que σ es válido en $ES_\perp$.

  Vemos entonces que asumiendo que existe un controlador válido σ' para s en ${ES'}_\perp$ estamos implicando la existencia de un controlador σ para s en $ES_\perp$. ABS!

□

Demostración Lemma 3: (Idea: Asumiendo $e' \notin W_{ES'_\perp}$, usamos un testigo s de $W_{ES'_\perp} \neq W_{ES_\perp}$ para llegar a una contradicción. El estado s debe tener un controlador en $W_{ES'_\perp}$ que evita $e \xrightarrow{\ell} e'$, la única diferencia entre $ES_\perp$ y $ES'_\perp$. Este controlador entonces es también un controlador para s en $W_{ES_\perp}$ llegando a un absurdo.

Asumimos $e' \notin L_{ES'_\top}$ y usamos un testigo s de $L_{ES'_\top} \neq L_{ES_\top}$ para llegar a una contradicción. Notar que como $e' \notin L_{ES'_\top}$, hay un controlador σ desde e' en $ES'_\top$. Como $s \notin L_{ES_\top}$ también debe haber un controlador σ' desde s en $ES_\top$. Construimos un nuevo controlador para $ES'_\top$ desde s que funciona exactamente como σ' pero cuando alcanza $e \xrightarrow{\ell} e'$ se comporta como σ. Este nuevo controlador prueba que $s \notin L_{ES'_\top}$ lo cual es una contradicción.)

Probamos ambas implicaciones por contradicción.

Primero asumimos que $e' \notin W_{ES'_\perp} \setminus W_{ES_\perp}$. Notar que como $e' \notin W_{ES_\perp}$ entonces $e' \notin W_{ES'_\perp}$. Como $W_{ES'_\perp} \neq W_{ES_\perp}$ y por la monotonicidad del (Lemma1) debe existir un estado s tal que $s \in W_{ES'_\perp} \setminus W_{ES_\perp}$, entonces s debe tener un controlador en $W_{ES'_\perp}$. Este controlador no puede alcanzar e' porque si lo hiciera, debería haber un controlador para e' y comenzamos asumiendo que $e' \notin W_{ES'_\perp}$. Más aún, si el controlador alcanzara e, entonces ℓ debe ser controlable (si fuera no controlable, el controlador alcanzaría e' lo cual ya establecimos que no es posible). Entonces, el controlador evita $e \xrightarrow{\ell} e'$ lo que significa que debe ser también un controlador para s en $ES_\perp$ (i.e., $s \in W_{ES_\perp}$) y alcanzamos una contradicción.

Ahora asumimos $e' \notin L_{ES'_\top} \setminus L_{ES_\top}$. Notar que como $e' \notin L_{ES_\top}$ entonces $e' \notin L_{ES'_\top}$. Sea $s \in L_{ES'_\top}$ y $s \notin L_{ES_\top}$. Sabemos que desde s debe haber un controlador σ' para $ES_\top$. Este controlador puede o ser también un controlador para ES o alcanzar el estado $\top$ en $ES_\top$. En el primer caso, es también un controlador en ES' y en $ES'_\top$, una contradicción. En el segundo caso, o usa una transición que no se encuentra ni en ES' ni en ES, lo que significa que en $ES'_\top$ va a alcanzar un estado ganador $\top$; o usa una transición que se encuentra en ES' pero no en ES lo que lleva a e'. Como sabemos que e' no se encuentra en $L_{ES'_\top}$, entonces cuenta con un controlador en $ES'_\top$, entonces sabemos que existe un controlador σ'' que incluye tanto a σ' como al controlador para e'. Finalmente, σ'' es un controlador para s en $ES'_\top$, pero $s \notin L_{ES'_\top}$, ABS!

□

Demostración Lemma 4: (Idea: Para probar que $e' \in W_{ES'_\perp} \setminus W_{ES_\perp}$ implica `canBeWinningLoop(`*loops*`)`, asumimos
$\neg$`canBeWinningLoop(`*loops*`)` y mostramos que $e' \notin W_{ES'_\perp} \setminus W_{ES_\perp}$. Para esto, basta con ver que si $\neg$`canBeWinningLoop(`*loops*`)` entonces para alcanzar un estado marcado desde e' se debe salir de *loops* a un estado $s \notin loops \cup W_{ES_\perp}$ lo que implica $s \notin W_{ES'_\perp}$ ya que s no tiene ningún camino de estados *None* que llegue a $e \xrightarrow{\ell} e'$ (Lemma 2). Como s no tiene controlador en $ES'_\perp$, es imposible que e' tenga uno.

Para probar que `canBeWinningLoop(`*loops*`)` implica $e' \notin L_{ES'_\top}$ construimos un controlador σ' para e' en $ES'_\top$ de la siguiente forma: Para una traza que se quede dentro de *loops*, solo elegimos sucesores controlables que no estén en $L_{ES_\top}$. Notar que no puede haber sucesores no controlables en $L_{ES_\top}$ ya que $loops \cap L_{ES_\top} = \emptyset$. Tan pronto como la

traza sale de *loops* a un estado s' usamos el controlador para s' en ${ES'}_\top$. Como s' no puede alcanzar $e \xrightarrow{\ell}_{ES'} e'$ usando estados *None*, por el Lemma 2, s' debe tener el controlador que necesitamos.)

Sea $e' \in {W_{ES'}}_\bot \setminus W_{ES_\bot}$ pero $\neg$`canBeWinningLoop(`*loops*`)`. Existe un controlador σ para e' en ${ES'}_\bot$, esto significa que debe existir un camino w desde e' hasta un estado marcado m. Dado que $\neg$`canBeWinningLoop(`*loops*`)`, no hay estados marcados en *loops*, w debe salir de *loops*. Sea s el primer estado que alcanza w fuera de *loops*, s pertenece a $L_{ES_\top} \cup None$, y no tiene un camino de estados *None* hasta e' entonces según Lemma 2 s va a seguir sin cambiar su estado. Dado que $s \notin {W_{ES'}}_\bot$, s no tiene un controlador en ${ES'}_\bot$, pero σ acepta corridas que llevan a s, llegamos a un absurdo.

Asumiendo `canBeWinningLoop(`*loops*`)` simplemente construimos un controlador σ^4 para e' en ${ES'}_\top$ para probar que $e' \notin {L_{ES'}}_\top$

Definimos σ^4 tal que habilita todas las transiciones no controlables (para que sea *controllable*).

$\sigma^4(w_0.w_1) = \sigma_{s'}(w_1)$ si w_0 es el camino más corto tal que existe $s' \notin loops \wedge s' \notin L_{ES_\top}$ y $e' \xrightarrow{w_0}_{{ES'}_\top} s'$. En otro caso $e' \xrightarrow{w}_{{ES'}_\top} p \wedge p \in loops$ entonces para toda ℓ' controlable, $\ell' \in \sigma^4(w)$ si y solo si $\exists p' \,.\, p \xrightarrow{l'} p' \wedge p' \notin L_{ES_\top}$.

Usamos los controladores $\sigma_{s'}$ donde s' es tal que existe $s \in loops$ y $s \xrightarrow{\ell'}_{{ES'}_\top} s' \wedge s' \notin loops \wedge s' \notin L_{ES_\top}$. Si no existe tal s', sabemos que debe haber un estado marcado en *loops* y σ^4 nunca abandona el conjunto *loops* ya que todos los estados alcanzables desde *loops* pertenecen a $L_{ES_\top}$.

Probaremos que σ^4 es $controllable$ y $non-blocking$.

Dado que habilitamos todas las transiciones no controlables, σ^4 es trivialmente *controllable*.

Para $non-blocking$, sea w compatible con σ^4, mostraremos que puede ser extendido. Si $w = w_0.w_1$ y w_0 es la palabra más corta tal que existe una $s' \notin loops \wedge s' \notin L_{ES_\top}$ y $e' \xrightarrow{w_0}_{{ES'}_\top} s'$. Entonces por definición de σ^4 sabemos que $\sigma^4(w_0.w_1.w_2) = \sigma_{s'}(w_1.w_2)$ para todo w_2. Ya que $\sigma_{s'}$ es $non-blocking$, existe un w_2 tal que $\sigma_{s'}(w_1.w_2)$ alcanza un estado marcado. Entonces $\sigma^4(w_0.w_1)$ puede ser extendido para alcanzar ese estado marcado.

En otro caso, w nunca abandona *loops*. Debemos probar que para todo ℓ' tal que $w.\ell'$ sea consistente con σ^4, $w.\ell'$ puede ser extendido con un w' para alcanzar un estado marcado. Sea p' tal que $e' \xrightarrow{w.\ell'} p'$. Si $p' \notin loops$ entonces $\sigma^4(w.\ell') = \sigma_{p'}(\lambda)$ y, como antes, sabemos que $\sigma_{p'}$ es $non-blocking$ entonces $w.l'$ puede extenderse para llegar a un estado marcado.

Si $p' \in loops$, entonces $w.l'$ puede extenderse para llegar a cualquier s en *loops*. Sabemos que o existe un estado marcado en *loops* o algún estado en $W_{ES_\bot}$ es alcanzable en un paso desde *loops*, de cualquier forma podemos extender $w.l'$ para llegar a un estado marcado.

□

Demostración Lemma 5: (Idea: Dividimos la prueba según la estructura del if/then/else de `findNewErrorsIn`. En el caso de que el `if` sea true, es suficiente probar que $e' \notin {L_{ES'}}_\top$. Para esto, construimos un controlador σ' para e' en ${ES'}_\top$ de la siguiente forma: Para una traza que se queda dentro de *loops*, solo tomamos sucesores controlables que no estén en $L_{ES_\top}$. Notar que no puede haber sucesores no controlables en $L_{ES_\top}$ ya que

$loops \cap L_{ES_\top} = \emptyset$. Tan pronto como la traza sale de *loops* al estado s' usamos el controlador de s' en ${ES'}_\top$. Como s' no puede alcanzar $e \xrightarrow{\ell}_{ES'} e'$ usando estados *None*, por el Lemma 2, s' debe tener tal controlador.

Cuando el `if` es false, alcanza con probar que $P = loops \subseteq L_{{ES'}_\top}$. Alcanzamos una contradicción asumiendo que $s \in loops \setminus L_{{ES'}_\top}$: Si $s \notin L_{{ES'}_\top}$ entonces tiene un controlador σ que acepta una traza w alcanzando un estado marcado. Como no hay estados marcados en *loops*, w alcanza un estado $s' \notin loops$. Como el `if` era false, $s' \in L_{{ES'}_\top}$ por lo que σ no es un controlador.)

En primer lugar, sabemos que cada estado $s' \notin loops$ tal que $\exists s \in loops \; . \; s \xrightarrow{l} s'$, puede o ser y seguir siendo un estado perdedor ($s' \in L_{ES_\top} \wedge s' \in L_{{ES'}_\top}$) o es y seguirá siendo *None* (porque s no es un predecesor$-None$ de un estado en *loops*, de otra forma s estaría en *loops*).

Esto significa que ningún estado $s' \notin loops \wedge s' \notin L_{ES_\top}$ puede ser forzado a un estado en $L_{{ES'}_\top}$. Entonces, si alcanzamos un estado *None* sabemos que tiene un controlador válido $\sigma_{s'}$ en ${ES'}_\top$.

En el caso de que la declaración `if` sea verdad, probaremos que $e' \notin L_{{ES'}_\top}$:

Usamos el mismo σ^4 de la demostración Lemma 4. Ya sabemos que σ^4 es tanto *controllable* como $non - blocking$ en esta situación por el Lemma anterior.

De otra forma entramos en el bloque `else`:

Si $\nexists s \in loops \; . \; s \xrightarrow{\ell'}_{{ES'}_\top} s' \wedge (s' \notin loops \wedge s' \notin Errors)$ probamos que $\forall s \in loops$, $s \in L_{{ES'}_\top}$

Sea σ' un controlador para s en ${ES'}_\top$ entonces $\exists w'$ tal que $s \xrightarrow{\lambda.w'}_{{ES'}_\top} e_m \wedge e_m \in M_{{ES'}_\top}$. Ya que no hay estados marcados en *loops*, partiendo desde s y siguiendo w' eventualmente se abandona *loops*.

Sea $w' = w_0.w_1$ tal que w_0 es la palabra más corta tal que $s \xrightarrow{w_0}_{{ES'}_\top} s' \wedge s' \notin loops$. Dado que $s' \in Errors \Rightarrow s' \in L_{ES_\top}$ no es posible que un controlador válido σ' acepte palabras que alcancen ese estado. ABS! Entonces no hay controlador para s en ${ES'}_\top$ lo que implica $\forall s \in loops$, $s \in L_{{ES'}_\top}$.

□

4.4. Complejidad computacional

La complejidad del algoritmo en el Listing 4.1 está acotada por $O(|S_E|^3 \times |A_E|^2)$, donde $|S_E|$ y $|A_E|$ son el número de estados y eventos en E. Esto se desprende del hecho de que el loop principal se ejecuta a lo sumo una vez por transición (i.e., $|T_E|$), que $|T_E| \leq |S_E| \times |A_E|$ y que la complejidad de una iteración del loop está acotada por $O(|S_{ES}|^2 \times |A_E|) \leq O(|S_E|^2 \times |A_E|)$.

La complejidad de cada iteración está acotada por la de `findNewGoalsIn`, la función más compleja de todas las llamadas durante cada iteración: `propagateError`, `canReach`, `propagateGoal`, `getMaxLoop`, `canBeWinningLoop`, `findNewGoalsIn`, y `findNewErrorsIn`.

A continuación presentamos un análisis detallado de la complejidad de cada sub-rutina:

Usaremos $|S_C|$ y $|T_C|$ para referirnos al número de estados y transiciones de una exploración parcial C de la planta E.

1. O(findNewGoalsIn(*loop*))<= $O(|S_{ES}| \times |T_{ES}|)$: Un punto fijo progresivamente remueve todos los estados que no son ganadores en ${ES'}_{\perp}$. En el peor caso se deben remover $|S_{ES}|$ estados, y el costo de remover cada un estado está acotado por $|T_{ES}|$. Cada iteración ejecuta otro punto fijo que va quitando los estados que son perdedores (en ${ES'}_{\perp}$) o son inalcanzables desde dentro de C. Esto se obtiene al chequear todas las transiciones entrantes y salientes para cada estado ($O(|T_{ES}|)$). Luego, cada iteración del primer punto fijo realiza una exploración "backwards BFS"sobre C ($O(|T_{ES}|)$), comenzando desde los estados marcados de C (o estados que tienen un hijo en *Goals*). Cuando la exploración termina, todos los estados de C que no fueron alcanzados son removidos.

2. O(findNewErrorsIn(*loops*))<= $O(|S_{ES}| \times |A_E|)$: Simplemente iteramos sobre todos los estados del loop y chequeamos si hay al menos un estado con un hijo NONE fuera del loop. Como nuestro autómata es determinístico sabemos que para cualquier estado s, $|hijos(s)| <= |A_E|$ (la cantidad de eventos). Además, la cantidad de estados del *loops* está acotada por S_{ES}.

3. O(propagateGoals(*newGoals*))<= $O(|S_{ES}| \times |T_{ES}|)$: El razonamiento es similar al de findNewGoalsIn, pero ahora iterando sobre todos los ancestros-NONE, cuya cantidad está acotada por S_{ES}. Luego, la complejidad de la función está acotada por $O(|S_{ES}| \times$ ($O($ iteración))).

   Cada iteración quita los estados s que son forzados a perder en un paso o no pueden alcanzar un estado *goal* dentro de C. La implementación logra esto con complejidad $O(|T_{ES}|)$. Revisar si cada estado s está forzado a perder en un paso solo requiere chequear sus hijos ($O(|T_{ES}|)$). Chequear cuales estados pueden alcanzar un goal requiere una búsqueda en anchura que comienza en los estados de C con un hijo *goal* y propagar hacia los ancestros (dentro de C). En el peor caso esto requiere iterar sobre todas las transiciones $|T_{ES}|$.

   En conclusón $O($ iteración$) = |T_{ES}|$ y O(propagateGoals(*newGoals*))$\leq O(|S_{ES}| \times |T_{ES}|)$.

4. O(propagateErrors(*newErrors*))= $O(|S_{ES}| \times |T_{ES}|)$): Esta función es similar a propagateGoals y se aplica el mismo razonamiento.

5. O(canReach(e, e'))= $O(|T_{ES}|)$: Simplemente requiere una búsqueda en anchura sobre las transiciones salientes.

6. O(getMaxLoop(e, e'))= $O(|T_{ES}|)$: Comenzando por e se puede realizar una búsqueda en anchura para iterar sobre todos sus ancestros, cortando la exploración cuando se llega a un estado GOAL, ERROR o e'.

7. O(canBeWinningLoop(*loops*))= $O(|T_{ES}|)$: Requiere verificar si hay un estado marcado en *loops* ($|loops| \leq |S_{ES}|$) y ver todos los hijos de estados en *loops* para ver si alguno es un estado en GOAL o ERROR ($|T_{ES}|$). Tenemos entonces $O(|S_{ES}| + |T_{ES}|) = O(|T_{ES}|)$

Entonces concluimos que la complejidad máxima de todas las funciones está acotada por $O(|S_{ES}| \times |T_{ES}|)$

Considerando que $|T_{ES}| \leq |S_{ES}| \times |A_E|$ y que en el peor caso para el tamaño de ES se da cuando $ES = E$, entonces todas las iteraciones del loop principal están acotadas por $O(|S_E|^2 \times |A_E|)$.

Finalmente, la complejidad total de nuestro algoritmo es: $O(|T_E| \times (|S_E|^2 \times |A_E|)) \leq O(|S_E|^3 \times |A_E|^2)$.

Previamente, [6] presentó una solución teórica para construir un director optimal en $O(|X| \times |\Sigma|(|Xm - Xt| + 1))$ (donde $X = S_E$, $\Sigma = A_E$ y $X_m - X_t$ es el número de estados marcados que no son terminales. Luego, comparando [6] con la última iteración de nuestro enfoque (cuando $ES = E$), la diferencia reside en la relación entre $|S_E|$ y $|X_m - X_t|$. Aunque $|S_E| \geq |X_m - X_t|$, el número de estados marcados no terminales puede sufrir proporcionalmente la misma explosión que los estados de la planta. Por lo tanto, en el peor caso, $O(|S_E|) = O(|X_m - X_t|)$. Lo que permite la afirmación de que la última iteración de nuestro algoritmo tiene una complejidad peor caso comparable a la complejidad de [6]. Sin embargo, nuestro trabajo sufre una penalidad a nivel de complejidad a raíz de intentar resolver el problema antes de conocer la composición completa de la planta. En SCD el loop es ejecuta hasta —T_E— veces, cada vez que a ES se le agrega una transición. Este es el costo en complejidad a pagar por la construcción on-the-fly. Los resultados experimentales muestran que esta complejidad teórica adicional puede compensarse con el tiempo ganado al evitar la construcción completa del state space, y en consecuencia tener iteraciones menos costosas sobre un espacio menor.

5. IMPLEMENTACIÓN

El algoritmo fue implementado en el lenguaje Java, agregando a la funcionalidad del programa Modal Transition System Analyser (MTSA)[1].

5.1. MTSA

El software utilizado cuenta con una gran cantidad de funcionalidad. Principalmente nos interesa la forma de escribir Labelled Transition Systems (LTS), esto se puede hacer mediante Finite State Process (FSP). En el listing 5.1 volvemos al caso de estudio presentado en 1.1 esta vez escrito en la herrmienta MTSA.

En primer lugar definimos las constantes que determinan la cantidad de sub-servicios a contratar. Luego definimos la componente `Agencia`, con un único estado que vuelve a sí mismo con las siguientes transiciones: `cancelacion, compra, query[servicio]`, este último se lee como cualquier transición `query[i]` si $i \in Servicio$.

Con la definición de los sub-servicios se puede ver una definición genérica compacta de múltiples componentes idénticas del problema, tantas como elementos haya en `Servicio`, cada una con su `id`, comenzado en 0, que es utilizado para definir transiciones únicas para cada componente. También se ve que para un componente más complejo simplemente se declaran más estados, cada uno con su nombre en mayúscula, sin necesidad de aclarar que pertenecen a un componente. Si se quiere que un estado tenga 2 transiciones que lleven a estados distintos se logra con el operador `|` (como ejemplo, ver el estado `Queried`).

Finalmente mostramos cómo se pueden componer las distintas LTSs con el comando `||`, con el cual nuestra planta compuesta tendrá tanto a la agencia como a los subservicios.

```
const N = 1
range Servicio = 0..(N-1)

Agencia = (
{cancelacion,compra} -> Agencia |
query[servicio] -> Agencia ).

SubService(id=0) = Unqueried,
Unqueried = (cancelacion -> Unqueried |
query[id] -> Queried),
Queried = (valido -> Disponible | noValido -> Imposible),
Disponible = ({compra,cancelacion} -> Unqueried),
Imposible = (canelacion -> Unqueried).

||Plant = Agencia || SubService[Servicio].
...
```

Listing 5.1: Ejemplo de LTS y composición

En el listing 5.2 vemos un ejemplo de una especificación de un problema de control, a la cual asignamos el nombre *Goal*. En el área de *Automated planning* el objetivo es alcanzar algún estado *final*, es decir, los estados son los marcados; sin embargo en el contexto MTSA y al representar el problema con LTSs solo podemos marcar transiciones. La interpretación final es que las transiciones señaladas como *marcadas* llevan a estados marcados y estos serán nuestros objetivos. En el ejemplo se declara la transición *compra* como marcada, señalando la compra sin errores de un paquete. En este punto aclaramos, únicamente para

facilitar cualquier intento de reproducción, que al utilizar transiciones marcadas, se puede generar un desdoblamiento de los estados. Como ejemplo notar que el estado `Agencia` va a tener (internamente) para la herramienta 2 versiones, una marcada alcanzada por *compra* y otra no marcada alcanzada por *cancelacion* y *query*[*servicio*].

Luego definimos el conjunto de transiciones controlables, en este caso *cancelacion*, *compra* y *query*[*Servicio*]. Por último debemos agregar la palabra clave `nonblocking`, en caso contrario se intentará, por defecto, resolver otro problema de control fuera del alcance de este trabajo.

En la última línea utilizamos la palabra clave `heuristic` para aclarar que queremos utilizar el algoritmo de SCD, luego nombramos como *DirectedController* al controlador que devuelve nuestro algoritmo cuando lo definimos como el LTS *Compuesto* (*Plant*)[1] que cumple con la especificación *Goal*.

```
...
controllerSpec Goal = {
marking = {compra}
controllable = {cancelacion, compra, query[Servicio]}
nonblocking
}

heuristic ||DirectedController = Plant~{Goal}.
```

Listing 5.2: Ejemplo de Controller y SCD

5.2. Heurísticas adicionales

5.2.1. Dummy/Debugging

Como dijimos en el capítulo 2, el algoritmo de SCD debe ser agnóstico a la heurística. Al comenzar nuestro trabajo en el proyecto y una vez que pudimos generar cierto conocimiento sobre el pseudocódigo existente nos percatamos de casos borde que podían no ser bien resueltos. Sin embargo, al correr dichos casos el resultado era correcto, esto se debía a que la heurística era muy buena y lograba ir por un camino directo al Goal (o Error, depende el caso); entonces no caía en nuestra "trampa".

En función de poner a prueba sólo el algoritmo de exploración desarrollamos una heurística de debugging o *Dummy*. La misma ordena las transiciones a explorar alfabéticamente, dejando primero las no controlables pero sin mirar información sobre distancia a marcados o error. Decidimos dejar el ordenamiento de no controlables primero ya que esto no es heurístico, se sabe por especificación qué transiciones son controlables y cuáles no.

A partir de entonces usamos los nombres de las transiciones para explorar nuestros casos de test de la forma que nos interesaba, ganando así control sobre los tests.

5.2.2. Breadth First Search

Si bien el algoritmo debe ser agnóstico a la heurística, la misma puede (y seguramente lo haga) modificar los tiempos de ejecución. Ya que si llega antes a alguna conclusión sobre un estado ésta información se puede propagar, cortando más ramas y/o más grandes.

La segunda heurística que desarrollamos es un simple Breadth First Search (BFS). La razón es para mostrar qué tanto se pueden mejorar los tiempos del algoritmo con una

[1] Cabe destacar que a esta altura la composición todavía no fue calculada

buena heurística. Esto se puede ver en detalle en el capítulo 6, donde mostramos resultados de un mismo benchmark para las diferentes heurísticas.

5.3. Testing

Luego de haber mostrado la sintaxis con la cual desarrollamos nuestros tests vamos a hablar un poco de ellos y detalles sobre algunos a destacar. El listing 5.3 muestra un ejemplo de sintaxis completo (test 1). Si se desea ver la especificación de cada uno de los tests puede encontrarse en el repositorio de MTSA[2].

```
Ejemplo = A1,
A1 = (u12 -> A2 | u14 ->A4),
A2 = (u21 -> A1),
A4 = (c45 ->A5),
A5 = (u55 -> A5).

||Plant = Ejemplo.

controllerSpec Goal = {
controllable = {c45}
marking = {u55}
nonblocking
}

heuristic ||C = Plant~{Goal}.
```

Listing 5.3: Test 1 a modo de ejemplo

En total tenemos una batería de 52 tests. Los cuales no fueron desarrollados uno detrás del otro sino que utilizamos una técnica llamada TDD (Test Driven Development). Desde el pseudocódigo e implementación existente de SCD iniciamos la batería de tests con algunos casos borde que resolvía mal. Luego entramos en un ciclo donde fuimos generando versiones que corregían los casos y más tests que rompían otras partes del algoritmo. Además en ciertos puntos críticos decidimos refactorizar completamente partes amplias de la implementación, fuera de las refactorizaciones chicas al pasar los tests.

Dentro de la batería queremos destacar algunos de especial interés, agrupados según qué problema específico atacan. Antes necesitamos explicar un poco de notación: las transiciones empezadas con u/c son no-controlables/controlables respectivamente, los estados marcados son los que tienen doble borde, y los triángulos son conjuntos de estados totalmente explorados. En ciertos tests con transiciones nombradas de forma diferente vamos a aclarar en la explicación cuáles son las controlables.

5.3.1. Marcado explícito de errores

Un estado deadlock (sin transiciones de salida) es obviamente un error, ya que si caemos en él no podemos alcanzar nunca un estado marcado. Ahora, ¿qué pasa si tenemos un conjunto de estados conectados entre sí pero sin conexión a otros fuera del conjunto?. Depende de si existe un estado marcado dentro del conjunto o no, ya que de no haberlo no tenemos forma de alcanzar uno (no podemos salir del conjunto).

La figura 5.1 ejemplifica este caso, el conjunto o rama B fue totalmente explorado y no contiene ningún estado marcado. En este momento es importante que agreguemos todos

[2] https://bitbucket.org/lnahabedian/mtsa/src/master/maven-root/mtsa/src/test/resources/NonBlocking/

los estados de B a *Errors*, de no hacerlo podríamos incurrir en una falla al propagar goal desde otra rama. Es decir, si se mira primero la rama de abajo y no lo marcamos como error (a pesar de estar completamente explorado), entonces al mirar la de arriba diremos que es goal y propagaremos dicha información, equivocadamente, más allá de e.

Éste problema se evita respetando el invariante presentado en la propiedad 1, ya que si una rama B cuya raíz es s está totalmente explorada, entonces $E_s \subseteq E$ es la reducción del problema que decide si s es un estado ganador o perdedor. Además, como está totalmente explorada $E_{s\perp} = E_{s\top}$, por lo que $W_{E_{s\perp}} \cup L_{E_{s\top}} = S_{E_s}$ y no puede darse el caso de la figura 5.1 en el que los estados de la rama no fueron clasificados.

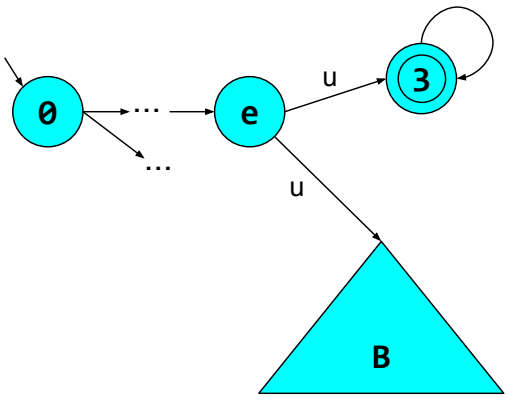


Fig. 5.1: Caso sub-autómata completamente explorado (B), sin marcados dentro.

Test 8 Fig 5.2 Este es un caso muy similar al presentado como ejemplo pero en lugar de una rama perdedora (llamada B) totalmente explorada a partir de u23 tenemos un solo estado (3). Si no se señala el estado 3 como error, porque no se detecta que es un livelock, entonces podríamos sacar la conclusión errada de que la planta es controlable; ya que 2 forma un loop con el estado inicial marcado. Los livelocks y las ramas perdedoras son difíciles de detectar hasta ser totalmente exploradas, pero en ese momento deben ser categorizados como errores, de otra forma no pueden distinguirse las ramas ya confirmadas como error y las parcialmente exploradas. Una vez que el estado 3 es marcado como error, concluir que la planta no es controlable resulta sencillo.

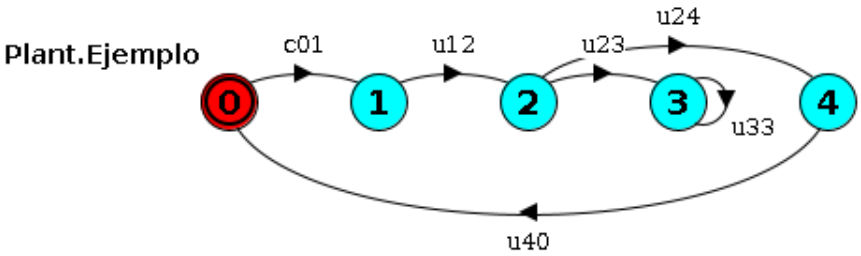


Fig. 5.2: LTS del test 8

Test 12 Fig 5.3 Variante del test 8, en este caso existe controlador ya que los estados [0,3,4,5] forman un loop con un estado marcado, y puede desactivarse c_0_m1 para evitar

salir del loop. Notar que aunque el estado 1 esté marcado es necesario dejarlo fuera del controlador para poder asegurar la victoria, ya que si se alcanza el estado 1 no puede evitarse una extensión al estado 2, el cual es perdedor. Notar que con extenderse para llegar a un estado marcado es suficiente, no es necesario alcanzarlos a todos.

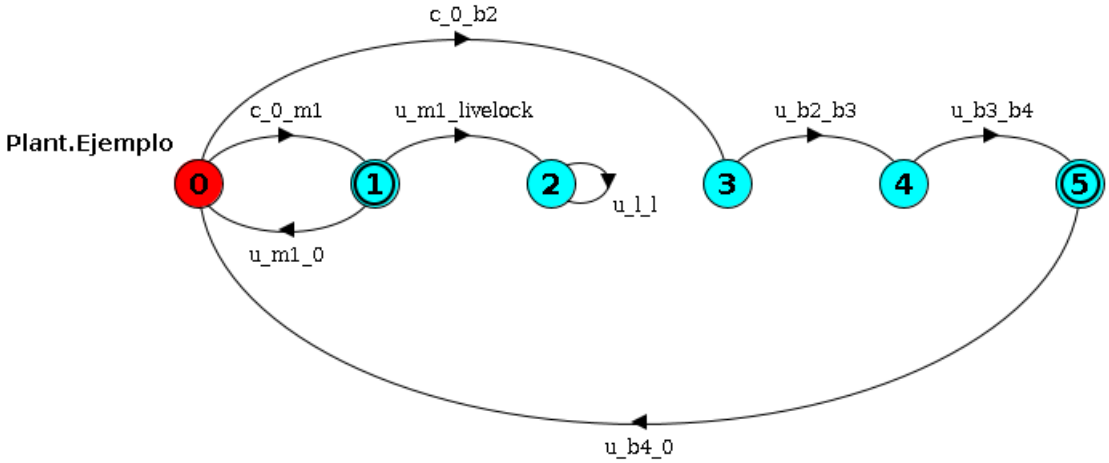


Fig. 5.3: LTS del test 12

5.3.2. Propagación local vs por conjuntos

Una vez obtenido un resultado necesitamos propagarlo hacia los estados ancestros. Necesitamos saber, para cada estado, si es posible sacar una conclusión dada la nueva información. Muchas veces es imposible hacerlo teniendo una mirada local, analizando solo un estado a la vez, ya que se pierde información sobre lo que sucede dentro del "conjunto" de ancestros vecinos.

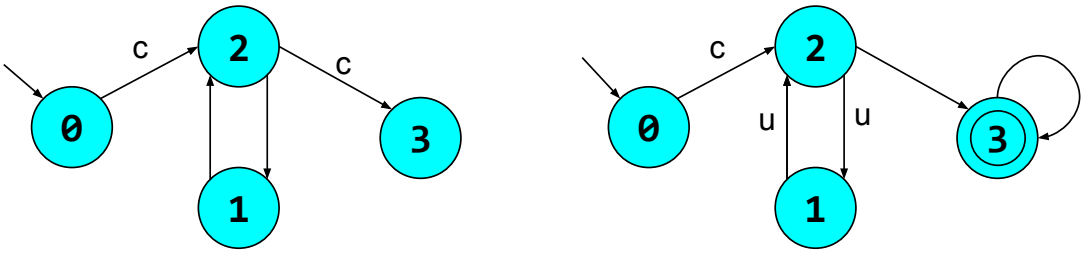
Por ejemplo en el caso de la figura 5.4a, hay un loop controlable entre dos estados, el cual se explora primero, y uno de ellos va controlablemente a un error. Debe habilitarse alguna controlable saliente del estado 2 (caso contrario sería un deadlock), pero ninguna lleva a un estado ganador, por ende la planta es no controlable. Sin embargo, según la mirada local nunca se concluiría que los estados [1, 2] son errores, porque ambos tienen *una forma de escapar del error*, el otro estado del loop, y no se sabe que ese otro estado está en la misma situación.

Equivalentemente la mirada local tampoco funciona propagando *Goals*, en la figura 5.4b se puede ver un ejemplo. El estado 2 llega al estado marcado 3 pero no puede forzarlo, sin importar si la transición a 3 es controlable o no ya que tiene una transición no controlable a 1. Como 1 solo puede volver a 2 ésta situación no nos molestaría (por ser non-blocking) y el modelo debería ser controlable. Ahora si miramos localmente al propagar *Goal* desde 3 *no sabemos dónde nos lleva la transición no controlable de 2 a 1*, y deberíamos suponer lo peor, sin poder marcar a 2 como ganador.

Entonces una mirada local no funciona a la hora de propagar. Pero ¿qué conjunto de ancestros vecinos deberíamos tomar? Es difícil decidir dónde hacer el corte; son muchos casos y no se puede, localmente, distinguirlos a todos. Por ende es necesario una propagación más inteligente, con una mirada global del conjunto de ancestros.

Como se detalló en el capítulo 4, al momento de propagar resultados recurrimos a

un punto fijo, tanto para *Goals* como *Errors*. De esta forma tenemos en cuenta toda la información acumulada de los estados en cuestión. Si bien esto implica un mayor costo de cómputo, asegura la correcta propagación de información, lo que más adelante facilita, por ejemplo, la detección de ciclos perdedores.



(a) Propagar errores (b) Propagar goals

Fig. 5.4: Problemas de propagación local.

Test 1 Fig 5.5 En este test, al descubrir que el estado 2 es ganador y propagar esa información, debe decidirse si el estado 0 es ganador. En principio parecería que no, puesto que un evento no controlable puede forzarlo a 3, pero es necesario darse cuenta que, si bien 0 puede ser forzado a 3, luego está obligado a volver, por lo que siempre hay una extensión que alcanza a 2. Por ende existe un director, que es el mismo autómata.

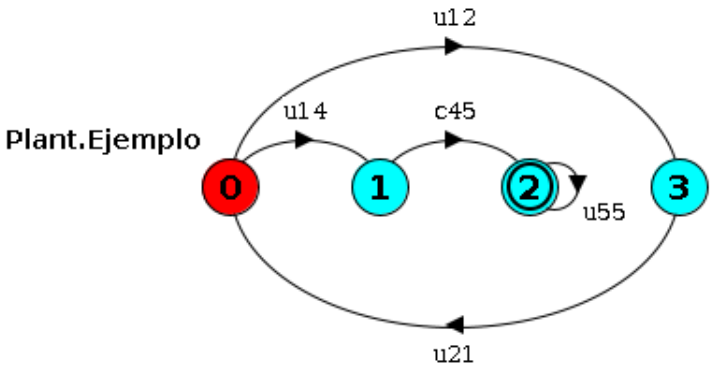


Fig. 5.5: LTS del test 1

Test 26 Fig 5.6 El loop [2,3,4] es ganador, pero desde el estado 2 el ambiente puede no controlablemente tomar el evento $u20$ para llegar al estado 0 y escapar al loop. La clave es que desde [0,1,2] siempre puede alcanzarse el mismo estado marcado 4, en realidad toda la planta es una componente fuertemente conexa con un estado marcado y por lo tanto es controlable. Si la propagación fuera local viendo un estado a la vez, concluiríamos que el estado 2 no puede marcarse como ganador porque un evento no controlable lo puede forzar al estado 0, el cual no sabemos si es perdedor.

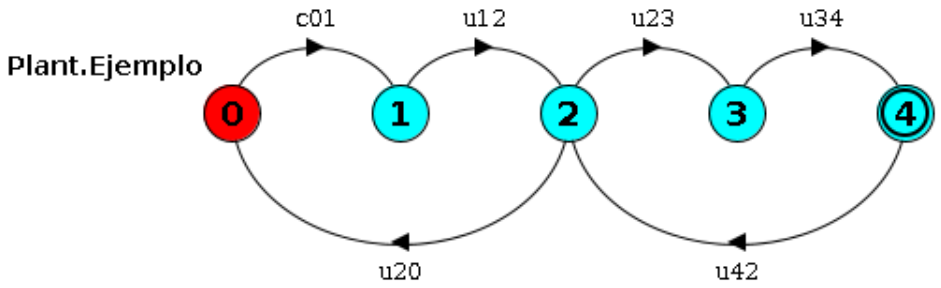


Fig. 5.6: LTS del test 26

5.3.3. Correcta detección de loops ganadores

La detección de loops con estados ganadores es una pieza central para la optimización y corrección del algoritmo, y tampoco puede solucionarse con una mirada local. Es sencillo dar una descripción declarativa de los estados a encontrar (ver listing 5.4) pero no resulta claro cómo implementarla.

Luego de varios intentos más performantes pero que presentaban fallas, llegamos a la conclusión de utilizar un algoritmo de punto fijo clásico (similar a listing 2.1) pero con una planta reducida. Los enfoques más veloces que intentamos sí funcionaron a la hora de detectar errores y terminaron sembrando las bases para `findNewErrorsIn(`*loops*`)`.

Como se vió en el capítulo 4, para detectar ganadores corremos un algoritmo clásico sobre una versión pesimista de la planta explorada, asumiendo que toda transición no vista es perdedora. Además solo tomamos en cuenta un grupo reducido, de los estados ya explorados, que forma un loop sobre la última transición expandida. Este enfoque otorga la completitud del algoritmo tradicional sosteniendo la eficiencia de la exploración on-the-fly.

```
function buildMCCC(e, e'):
let C such that
C = {e_i | (e' -w->_ES e_i -w'->_ES e ∨ e -w->_ES e_i -w'->_ES e') ∧ extendsCCC(e_i, C ∪ Goals) ∧
(∃w . e -w->_ES e_m ∧ e_m ∈ M_E ∩ (C ∪ Goals))}
return C

function extendsCCC(e, C):
return (∃ℓ . e -ℓ->_E e' ∧ e' ∈ C) ∧ (∀ℓ_u ∈ A_U . e -ℓ_u->_E e' ⇒ e' ∈ C)
```

Listing 5.4: vieja descripción estados ganadores

Test 35 Fig 5.7 Este test utiliza la heurística debugging y su funcionalidad de ordenar las transiciones alfabéticamente. Las transiciones controlables son {e, a, i}; recordar que los eventos no-controlables se exploran primero.

A la hora de armar por punto fijo un conjunto de estados ganadores hay que tener mucho cuidado de no quedarse sin estados marcados o a un paso de goal, ya que estos son los que le dan la capacidad de ganar al conjunto (ver función `findNewGoalsIn` en el listing 4.3). El test 35 resalta esta problemática al analizar el loop [2,3,4,5,6], como el evento f es no controlable y el estado 5 es un livelock, la función `findNewGoalsIn` remueve al estado 4 del conjunto a analizar. Esto causa que el estado 2 sea inalcanzable por los estados restantes del conjunto, y también es removido; pero como e es controlable todavía tenemos un conjunto controlablemente cerrado [3,6] que se querría marcar como ganadores. Aquí

podría terminar el punto fijo, pero debe tenerse cuidado en que en el conjunto resultante todavía haya estados marcados (o a un paso de goal) a los cuales alcanzar, cosa que no ocurre en este caso, por lo que ningún estado de [2,3,4,5,6] es ganador.

La correcta solución para esta planta es un controlador que desactiva el evento a y es ganador por alcanzar el self loop del estado marcado 1.

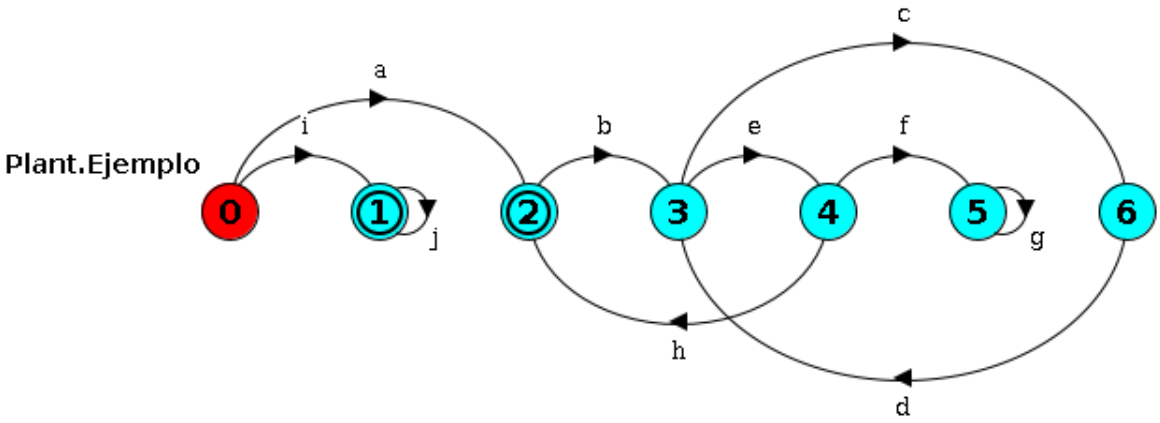


Fig. 5.7: LTS del test 35

Test 41 Fig 5.8 También se utiliza la heurística debugging, y la transición controlable es $a04c31$. La clave es que `findNewGoalsIn` debe detectar nuevos loops ganadores tanto si tienen un estado ganador a un paso como si contienen estados marcados. La heurística debugging es esencial para el test, porque queremos testear la detección de nuevos loops ganadores, no la propagación. El estado 2 es marcado como ganador antes de explorar $a03u23$ y cerrar el loop [0,1,3]. Esto evita que al propagar el goal del estado 2 se marque a los estados 0,1 y 3 como ganadores; porque no se sabe a donde lleva $a03u23$. Luego, el conjunto [0,1,3] es ganador por tener un estado Goal a un paso (el estado 2).

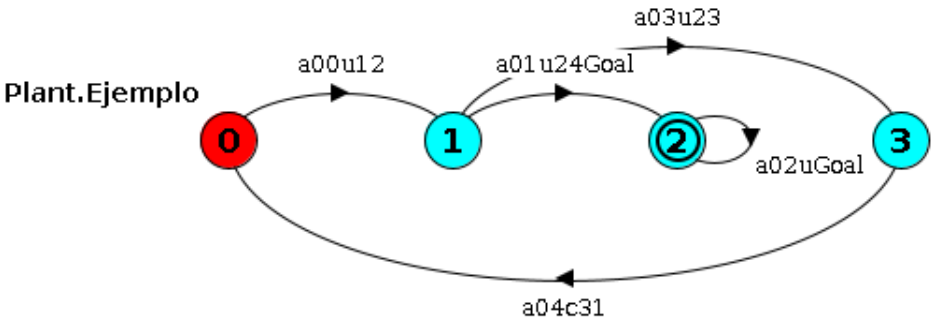


Fig. 5.8: LTS del test 41

Test 49 Fig 5.9 Es similar al test 35 pero en este caso no existe controlador. Luego de hacer el punto fijo en `findNewGoalsIn` para quedarse con los estados ganadores, puede quedar más de un componente fuertemente conexo de estados. En dicho caso solo los conjuntos que tengan marcados/ganadores a un paso son los que deberían ir a *Goals*, ya que los estados de otros componentes no podrán llegar a ellos de manera segura. En particular `findNewGoalsIn` remueve los estados que no pueden alcanzar goals o marcados dentro del conjunto al final de cada iteración del punto fijo. Esto asegura que las componentes

conexas incluyan un estado marcado o estén a un paso de un estado ya señalado como goal.

Concretamente lo que sucede al explorar la planta es que por el orden de la heurística debugging se van descubriendo los loops chicos, terminando de cerrar todo con la transición $u10$. En este momento se buscan ganadores dentro del loop [0, 1, 2, 3, 4, 5], como el estado 2 tiene una no-controlable por explorar ($u99$ se explora penúltima) es removido del conjunto dejando dos componentes conexas, [0, 1] y [3, 4, 5]. De no tener cuidado podríamos asumir que todos son goals ya que existe un marcado dentro del conjunto total [0, 1, 3, 4, 5]. Pero este (estado 5) no puede ser alcanzado de forma segura por la componente conexa [0, 1]. Por esto, se marca a los estados [3, 4, 5] como ganadores pero ya que el estado inicial 0 no puede alcanzar los estados ganadores sin pasar por 2 que es perdedor, la planta no es controlable.

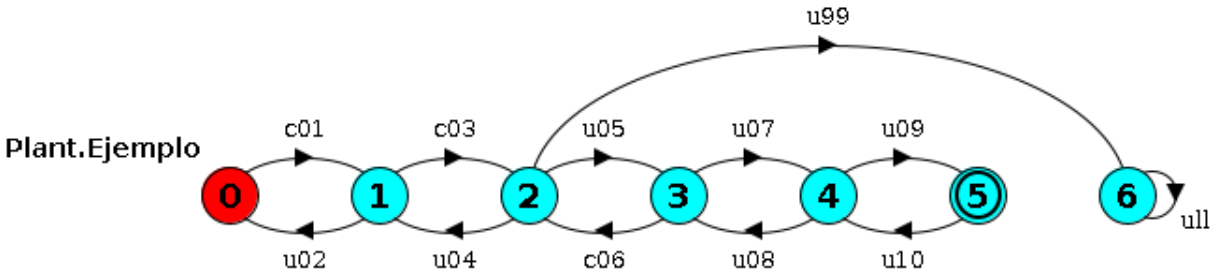


Fig. 5.9: LTS del test 49

6. PERFORMANCE

Para realizar las pruebas de performance decidimos usar el mismo conjunto de problemas que el utilizado para examinar el algoritmo original de SCD en [2]. En esta sección presentamos los resultados de la comparación versus dicha versión y, además, contra diversos programas del estado del arte. Todos los casos de estudios fueron escritos con la posibilidad de ser escalados en el número de componentes y estados.

Transfer Line Automatización de una fábrica, un dominio de mucho interés en el área de supervisory control. TL consiste de n máquinas conectadas por n buffers cada uno con capacidad de k unidades, termina en una máquina adicional llamada Test Unit.

Dinning Philosophers Problema clásico de concurrencia. En DP hay n filósofos sentados en una mesa redonda, cada uno comparte un tenedor con sus vecinos aledaños. El objetivo del sistema es controlar el acceso a los tenedores de manera que los filósofos puedan alternar entre comer y pensar; evitando *deadlock* y *starvation*. Adicionalmente, cada filósofo, luego de tomar un tenedor, debe cumplir con k pasos de etiqueta antes de comer.

Cat and Mouse Juego de dos jugadores donde cada uno toma turnos para moverse a una casilla adyacente dentro de un mapa de la forma de un corredor dividido en $2k + 1$ áreas. En CM n gatos y la misma cantidad de ratones son colocados en extremos opuestos del corredor. El objetivo es mover a los ratones de manera que no terminen en el mismo lugar que un gato. Los movimientos de los gatos no son controlables. En el centro del corredor hay un agujero que lleva a los ratones a un área segura.

Bidding Workflow Modela el proceso de evaluación de proyectos de una empresa. El proyecto debe ser aprobado por n equipos. El objetivo es sintetizar un flujo de trabajo que intente llegar a un consenso, es decir, aprobar/rechazar el proyecto cuando todos los equipos lo aceptan/rechazan. La propuesta puede ser reasignada para reevaluación por un equipo hasta k veces, no se puede reasignar si el equipo ya lo había aceptado. Cuando un equipo lo rechaza k veces el proyecto puede ser rechazado sin consenso. Es un caso de estudio típico del dominio de Business Process Management.

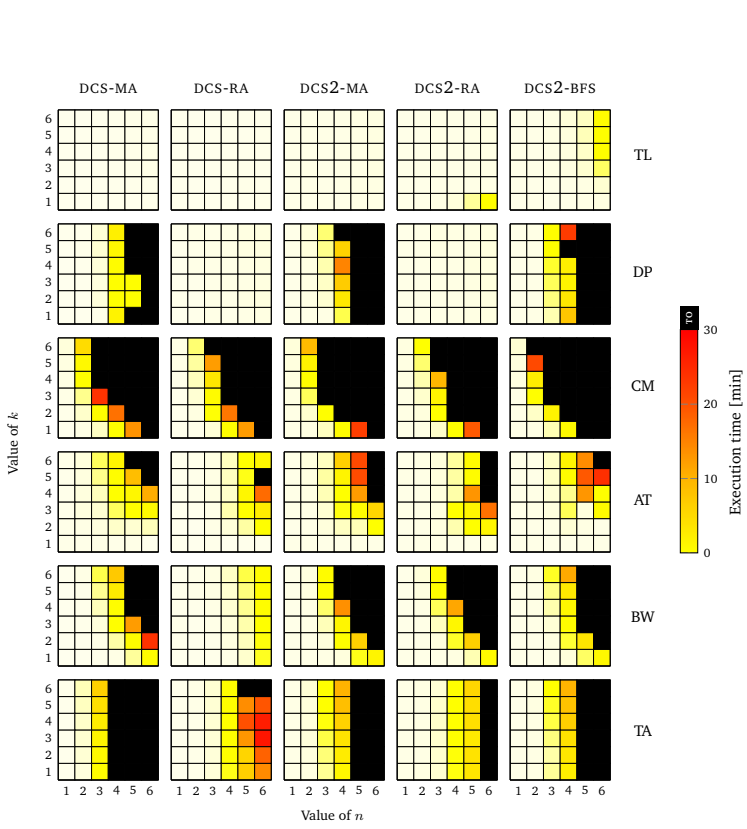
Air-Traffic Management Representa la torre de control de un aeropuerto, que recibe n peticiones de aterrizaje simultáneas. La torre necesita avisar si tiene permiso para aterrizar o, en caso contrario, en cuál de los k espacios aéreos debe realizar maniobras de espera. El objetivo es que todos los aviones puedan aterrizar de manera segura. El problema solo tiene solución si la cantidad de aviones es menor a la de espacios aéreos ($n < k$).

Travel Agency Modela una página on-line de ventas de paquetes de viajes. El sistema depende de n servicios de terceros para realizar las reservas (ej. alquiler de auto, compra de pasajes, etc). Los protocolos para utilizar los servicios pueden variar de manera no controlable; una variante es la selección de hasta k atributos (ej. destino del vuelo, clase y fechas). El objetivo del sistema es orquestar los servicios de manera de obtener un paquete de vacaciones completo de ser posible, evitando pagar por paquetes incompletos.

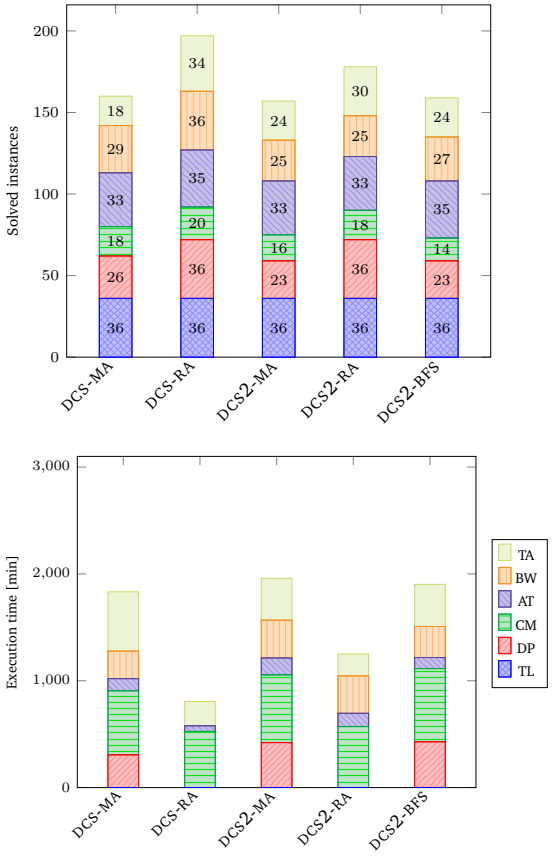
6.1. Comparación entre resultados de SCD

Como ya dijimos, el foco del trabajo no estuvo solo en corregir los errores encontrados en el algoritmo sino también en brindar una mayor seguridad sobre la corrección y completitud de la exploración on the fly. Esto debía hacerse sin perder la buena performance que aportaba la técnica, permitiendo aplicarla a casos de mayor tamaño. Aclaramos que el benchmark se corrió en un equipo de las mismas características para ambas versiones de SCD.

La comparación se realizó para las dos heurísticas desarrolladas en [2], Monotonic Abstraction y Ready Abstraction. Agregamos además una de las heurísticas naif, Breadth First Search, que desarrollamos para depurar el código; esto es en función de mostrar la posible ganancia obtenida a través de una buena heurística.



(a) resultados de comparación SCD en detalle.



(b) Instancias resueltas totales (arriba) y tiempo de ejecución (abajo) con SCD.

Fig. 6.1: Resultados de comparación entre SCD.

En la figura 6.1 se pueden observar los resultados con respecto al algoritmo de SCD en [2]. Antes que nada podemos concluir que hay una ganancia al utilizar mejores heurísticas, ya que BFS pierde contra RA tanto en cantidad de instancias como tiempo. Sorprendentemente a pesar de ser muy naif termina compitiendo contra MA. Esto tiene su explicación en el cambio de objetivo, ya que MA fue presentada en [3] para *reachability*, en lugar de *non-blocking*.

Nuestra versión (SCD2) mantiene la performance en la mayoría de los casos de estudio. En TA hay algunas instancias nuevas que dan timeout, pero el resto concluyen rápidamente. El único caso donde pierde notoriamente es BW, pero mirando otras herramientas (con la excepción de MYND) vemos que en general tienen problemas con los casos más grandes de BW. Nuestra suposición es que la implementación SCD anterior clasificaba estados como *Goals* de manera anticipada y posiblemente erronea, quizás casos como los vistos en la sección 5.3. Esto puede hacer que se llegue a una conclusión apurada que posiblemente devuelva un controlador incorrecto, pero que en el benchmark parezca resolver más instancias.

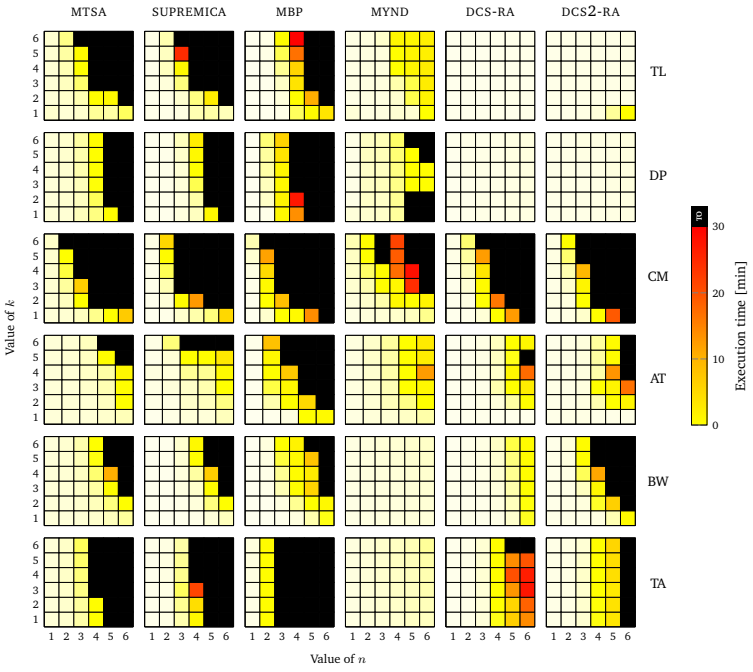
6.2. Comparación con otros programas

En la comparación entre distintas herramientas es importante destacar que se hacen de forma ilustrativa para justificar que este trabajo presenta una herramienta del estado del arte. No debe usarse esta comparación para justificar que una herramienta es mejor que otra. En especial, cabe recordar que las herramientas no sintetizan soluciones con las mismas características. Supremica garantiza que su controlador es maximalmente permisivo, lo cual puede incurrir en un mayor costo computacional.

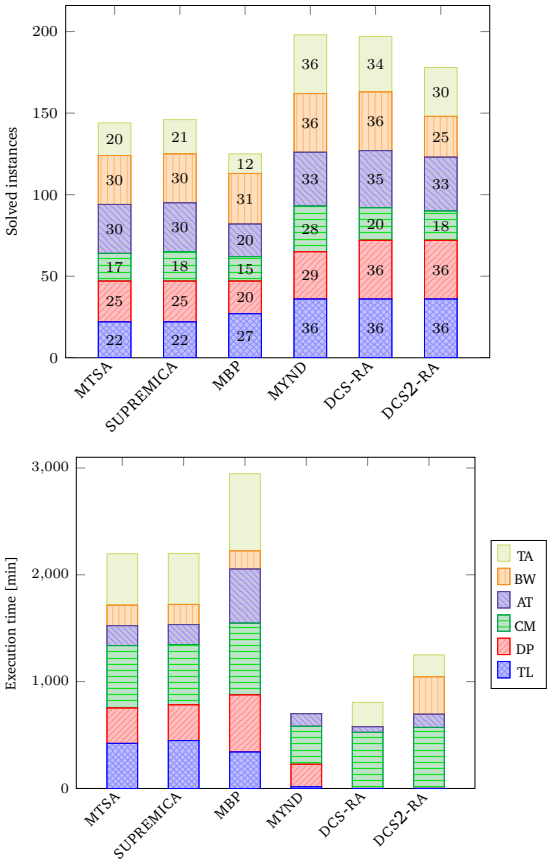
Adicionalmente, al comparar MTSA contra otras herramientas el benchmark es traducido a la sintaxis aceptada por MBP [5], MYND [8] y SUPREMICA [9]. La traducción se realiza de forma automática y fue probada correcta en [3], sin embargo, la construcción automática de especificaciones puede omitir ciertas optimizaciones de modelado conocidas por los expertos del área. En [3] se muestra que las traducciones no implican un aumento exponencial de estados, pero eso no descarta que la traducción pueda afectar los desempeños de las otras herramientas.

Si bien el algoritmo presentado parece perder un poco de escalabilidad en ciertos casos, en la figura 6.2 podemos ver que sigue estando en una posición competitiva con respecto a otras herramientas del estado del arte. Para que sea más simple la comparación decidimos mostrar solo RA, ya que es la heurística de mejor desempeño, y ambas versiones de SCD.

Aunque la nueva versión esté un poco más lejos de MYND, en calidad de cantidad de intancias y tiempo supera ampliamente el resto de las opciones, referirse a fig 6.2b.



(a) detalle de comparación entre herramientas.



(b) Instancias resueltas totales (arriba) y tiempo de ejecución (abajo) para cada herramienta.

Fig. 6.2: Resultados de comparación entre herramientas.

7. CONCLUSIONES

Al empezar con el proyecto y leer sobre control supervisado descubrimos que hay todo un mundo detrás. Primero debimos aprender sobre los algoritmos composicionales y no composicionales. Luego entender el algoritmo estándar y parte de su implementación para finalmente poder arrancar con el algoritmo on-the-fly. Este último lo debimos entender a la perfección, para poder descubrir y solucionar los diversos problemas.

MTSA es un proyecto con gran trayectoria y muchos avances en diversos frentes hechos por diferentes personas y grupos de investigación; como tal su código puede ser muy complejo, teniendo partes escritas incluso en versiones antiguas de java.

Pese a estos desafíos, logramos las siguientes contribuciones:

- Un nuevo algoritmo de exploración, cuya correctitud es agnóstica a la heurística utilizada. Esto facilita el desarrollo y utilización de nuevas heurísticas sin comprometer la corrección o completitud del algoritmo.
- Hasta donde sabemos, este trabajo presenta la primera implementación para síntesis de directores.
- Una prueba de la correctitud y completitud del algoritmo presentado.
- Un análisis de la complejidad de dicho algoritmo.
- Una batería de tests de regresión como una adición permanente al proyecto de MTSA para garantizar la continua correctitud de su feature de síntesis de controladores con exploración heurística.
- Dos nuevas heurísticas de exploración, para ayudar al testeo y comprobar el agnosticismo del algoritmo a la heurística.
- Resultados experimentales para comprobar que las modificaciones a la exploración siguen manteniendo la buena performance de la técnica y que el algoritmo está en nivel de competición con otras herramientas del estado del arte.

Como trabajo a futuro, presentamos las siguientes ideas:

- Generación automática de tests por medio de mutaciones, para facilitar el desarrollo de nuevos algoritmos de síntesis sin la carga de generar nuevos tests a mano.
- Modificación del algoritmo de exploración on-the-fly para resolución de otros problemas, por ejemplo cambiando el objetivo de nonblocking por objetivos de tipo GR1. Acompañado del desarrollo de heurísticas correspondientes al nuevo problema.

Bibliografía

[1] Modal transition system analyser (mtsa).

[2] D. Ciolek. *Síntesis dirigida de controladores para sistemas de eventos discretos*. PhD thesis, Laboratorio de Fundamentos y Herramientas para la Ingeniería del Software (LaFHIS), FCEyN, UBA, 2018.

[3] D. Ciolek, V. Braberman, N. D'Ippolito, and S. Uchitel. Directed Controller Synthesis of discrete event systems: Taming composition with heuristics. In *Proc. of the IEEE Conf. on Decision and Control*, CDC, pages 4764–4769, 2016.

[4] Ruediger Ehlers, Stephane Lafortune, Stavros Tripakis, and Moshe Vardi. Reactive synthesis vs. supervisory control: Bridging the gap. Technical Report UCB/EECS-2013-162, EECS Department, University of California, Berkeley, Sep 2013.

[5] A. Gromyko, M. Pistore, and P. Traverso. A Tool for Controller Synthesis via Symbolic Model Checking. In *Proc. of the Int. Workshop on Discrete Event Systems*, 2006.

[6] Jing Huang and Ratnesh Kumar. Optimal nonblocking directed control of discrete event systems. *American Control Conference*, pp. 4285–4290, 2007.

[7] Jing Huang and Ratnesh Kumar. Directed Control of Discrete Event Systems for Safety and Nonblocking. *IEEE Trans. Automation Science & Engineering*, 5, 2008.

[8] Robert Mattmüller, Manuela Ortlieb, Malte Helmert, and Pascal Bercher. Pattern database heuristics for fully observable nondeterministic planning, 2010.

[9] S. Mohajerani, R. Malik, and M. Fabian. A Framework for Compositional Synthesis of Modular Nonblocking Supervisors. *IEEE Tran. on Automatic Control*, 59(1):150–162, 2014.

[10] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., 2004.

[11] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proc. of the Symp. on Principles of Programming Languages*, POPL, pages 179–190, 1989.

[12] P. J. Ramadge and W. M. Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal on Control and Optimization*, 25, 1987.

[13] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *Proc. IEEE*, 77, 1989.

[14] S. A. Zudaire, M. Garrett, and S. Uchite. Iterator-based temporal logic task planning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11472–11478, 2020.