



UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

TESIS

Directed Controller Synthesis for Non-Maximal Blocking Requirements

Tesis de Licenciatura en Ciencias de la Computación

Matias Duran, Florencia Zanollo

Director: Sebasitán Uchitel

Codirector: ???

Buenos Aires, 2020

SINTESIS DE CONTROLADORES DIRIGIDA

El presente proyecto de tesis consistió en un estudio y extensión del método previamente propuesto por Daniel Ciolek en su tesis de doctorado [2]. Más precisamente, se trató de analizar carencias del algoritmo de exploración on-the-fly para problemas de Supervisory Control, cuya propiedad central era de tipo Non-blocking y, posteriormente analizados los problemas, afrontarlos con una nueva especificación e implementación del algoritmo. La funcionalidad fue incorporada al software MTSA¹. Finalmente, se adaptó el algoritmo para construir directores en lugar de supervisores maximales.

Palabras claves: Discrete Event Systems, Supervisory Control (no menos de 5!!).

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

DIRECTED CONTROLLER SYNTHESIS

El abstract pero en ingles? (aprox. 200 palabras).

Keywords: blabla (no menos de 5).

AGRADECIMIENTOS

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

Índice general

1..	Introducción	1
1.1.	Caso de estudio	1
1.2.	Estructura de los capítulos	3
2..	Antecedentes	5
2.1.	Controlador objetivo	5
2.2.	Director	6
2.3.	Algoritmo monolítico	7
2.4.	Exploración on-the-fly	8
2.4.1.	marcado explícito errores	9
2.4.2.	Propagación local vs por conjuntos	9
2.4.3.	Complejidad de exploración	10
2.4.4.	Correcta detección de loops ganadores	10
2.4.5.	Agnosticismo a la heurística	11
3..	Nuevo Directed Controller Synthesis	13
3.1.	Propuesta de nuevo algoritmo	13
3.2.	Demostración de correctitud y completitud	17
3.3.	Nuestro enfoque	19
3.4.	Demostración de Lemas	21
3.5.	Complejidad computacional	26
4..	Implementación	29
4.1.	MTSA	29
4.2.	Heurísticas adicionales	30
4.2.1.	Dummy	30
4.2.2.	BFS	30
4.3.	Testing	31
5..	Performance	33
5.1.	Comparación con versión previa de DCS	34
5.2.	Comparación con otros programas	35
6..	Conclusiones	37

1. INTRODUCCIÓN

El problema de síntesis (construir automáticamente un controlador en base a una especificación) fue estudiado dentro de distintas áreas como: Discrete Event Control [REFS], Reactive Synthesis [REFS] y Automated Planning [REFS]. El problema puede modelarse con un Sistema de Eventos Discretos (DES) con un subconjunto de sus estados marcados. Un factor clave de estos problemas es que el DES se presenta de forma modular tal que la composición paralela de múltiples componentes den lugar al DES de interés. Desarrollaremos en mayor detalle las definiciones del problema en el capítulo 2.

La motivación para analizar dichos problemas surge de la necesidad de verificar software, hoy en día utilizado en prácticamente todo emprendimiento humano. Si bien puede irse ganando confianza sobre la correctitud de un algoritmo a través de una batería de tests, éstos no proveen una garantía sino una seguridad cada vez mayor.

Un método alternativo es el de la verificación de la implementación del algoritmo con un modelo formal que cumpla los objetivos y requerimientos deseados del programa. Con esta visión en mente, el área de ‘Controller Synthesis’ va un paso más allá y busca la generación automática de un controlador que dado un modelo (en forma de DES) cumpla siempre en toda ejecución posible los requisitos del problema.

Podemos trazar una comparación con el método de “Machine Learning”, en auge desde hace unos años. En este paradigma, el programa tiene como input una multitud de casos de ejemplo del comportamiento buscado, si ponemos como ejemplo ganar un juego de ajedrez, tomaría una biblioteca de partidas jugadas de las cuales aprender. Como resultado, presentaría un programa que sabe jugar al ajedrez “muy bien”, es decir, es muy probable que dada una partida, la gane, pero no está garantizado. Pueden verse hoy en día muchas aplicaciones de este método con gran éxito, desde la dominación del juego del “go” hasta autos manejados por IA. Sin embargo, ya que no ofrece ninguna garantía, es un método poco apropiado para sistemas críticos.

Como contraste, la Síntesis de controladores toma como input las reglas del juego de ajedrez, por ejemplo, varios componentes (autómatas) separados, siendo cada uno los movimientos posibles para una pieza. Como resultado, daría un controlador, que en cada momento de la partida solo habilita algunas de las transiciones de los autómatas y garantiza que si se le hace caso ganará la partida. Es esencial notar que la técnica de este trabajo busca obtener una garantía muy fuerte, y el problema lo encuentra en la escalabilidad del problema, ya que hoy en día es imposible aplicarla a un juego del tamaño del ajedrez.

1.1. Caso de estudio

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

Se desea armar un servicio 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.

Para cada servicio que se desea sub-contratar presentamos una versión simplificada en la cual se consulta si ese servicio está disponible. En caso de estar disponible queda

reservado hasta la compra definitiva del paquete completo y la cancelación de la reserva si otro servicio no estaba disponible no implica un gasto.

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 5).

Mostramos en la figura 1.1 un LTS (Labeled transition system) para cada uno de los componentes descriptos y el LTS compuesto para el caso en el que se sub-contrata un solo servicio.

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 automáticamente por MTSA¹ 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.

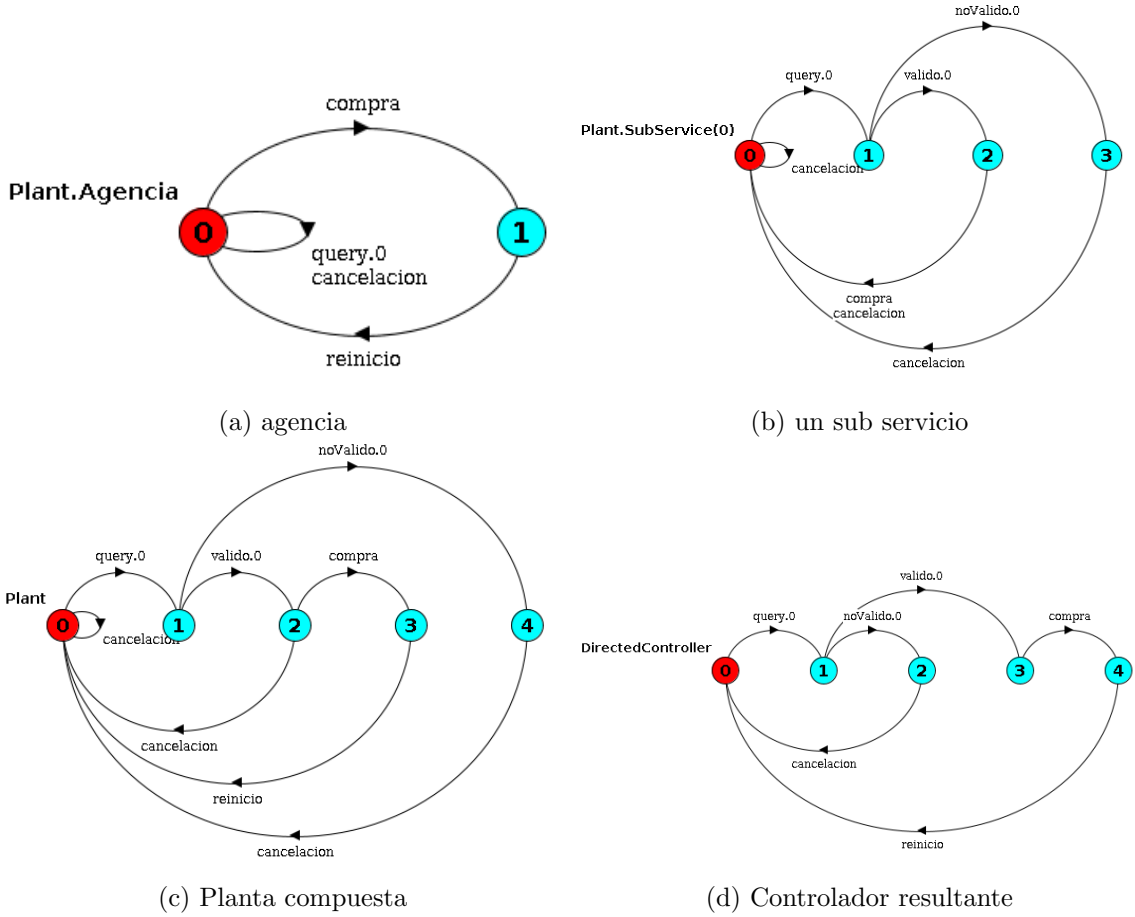


Fig. 1.1: Caso con un solo sub-sevicio

Los algoritmos tradicionales necesitan construir el sistema compuesto entero antes de empezar, es por ésta explosión de estados y transiciones que resulta muy costoso (aveces imposible), por ende los algoritmos de punto fijo pueden manejar hasta cierto tamaño

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

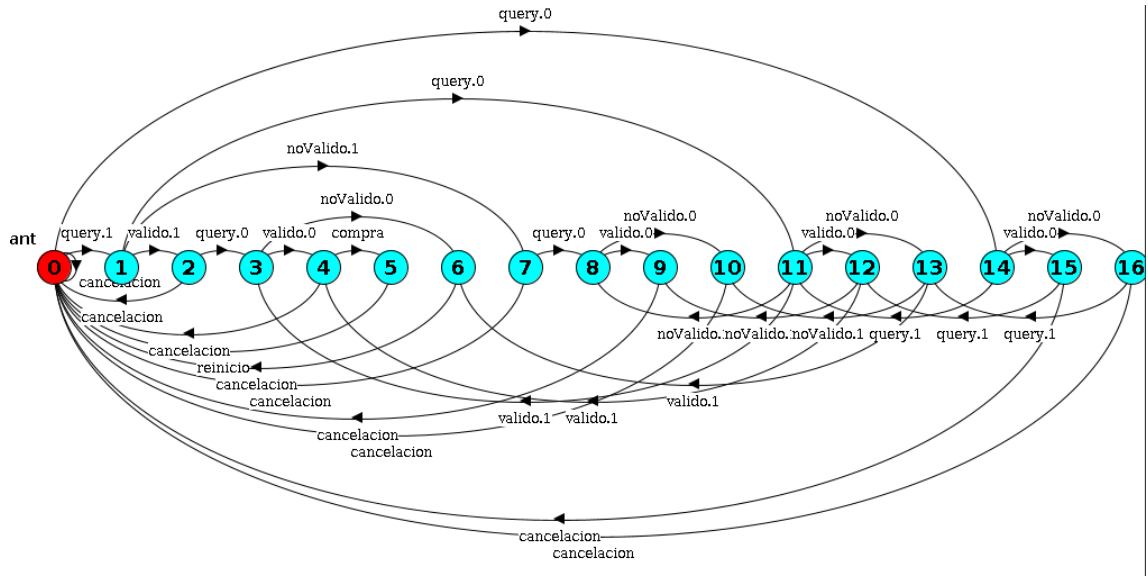


Fig. 1.2: Planta compuesta con 2 sub-servicios

de problemas. En el caso de on-the-fly al ser una exploración dirigida construye sólo lo necesario, sacando conclusiones en función de la información obtenida; en el peor caso puede contruir todo (si es que esto es posible).

1.2. Estructura de los capítulos

En lo siguiente empezamos contando antecedentes, algoritmos tradicionales, la idea básica de exploración on-the-fly y lemas a seguir en orden de conseguir corrección y completitud; ésto puede encontrarse en el capítulo 2.

En el capítulo 3 mostramos nuestra propuesta de algoritmo, demostramos corrección y completitud para el mismo y analizamos la complejidad computacional de su peor caso.

Luego exponemos detalles sobre la implementación en el capítulo 4, MTSA, heurísticas diseñadas para testeo y detalles sobre la batería de test, desarrollada utilizando TDD (Test Driven Development).

Como paso final en el capítulo 5 presentamos el benchmark utilizado y resultados de performance; tanto versus la versión anterior de DCS como versus otros algoritmos del estado del arte.

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 $w = \ell_0, \dots, \ell_k$ 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 $t_0 \xrightarrow{\ell_0 \dots \ell_k}_T t_{k+1}$.

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 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 $\bar{t} \xrightarrow{w}_T t_{k+1}$.

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

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

2.1. Controlador objetivo

Dado un autómata 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 algunas 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 $w = \ell_0, \dots, \ell_k$ una palabra en $\mathcal{L}(T)$, entonces $w \in \mathcal{L}^\sigma(T)$ si y solo si para todo $0 \leq i \leq k$: $\bar{t} \xrightarrow{\ell_0 \dots \ell_i}_T t_{i+1} \wedge \ell_i \in \sigma(\ell_0, \dots, \ell_{i-1})$

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 $\{E_0, \dots, E_n\}$ (podemos abusar la notación y usar $E = (S_E, A_E, \rightarrow_E, \bar{e}, M_E)$ para referirnos a la composición $E_0 \parallel \dots \parallel E_n$), 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).

Podemos pensar en un controlador non-blocking como un jugador optimista. Se encarga de no perder, y mientras tenga un futuro camino posible que lo lleva al destino buscado, considera que está ganando.

Es clave entender que en el problema a tratar, la posición de "tablas" del ajedrez, en la que ambos jugadores repiten sus jugadas 50 veces, se considera ganadora si todavía hay opción de dar un jaque mate. Si repetimos nuestras jugadas y todavía tengo dos torres considero que gané el partido, porque eventualmente mi oponente podría cansarse y dejarme ganar. Si repetimos nuestras jugadas pero solo tengo mi rey, no hay forma de dar mate, no puedo extender esta "palabra", esta partida, de forma de dar mate, y considero que perdí.

Es importante notar que como se busca que cualquier palabra sea extendible a otro estado marcado, lo que se busca es pasar por algún estado marcado infinitas veces. O sea, 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.

De forma más concreta, en nuestro algoritmo, un ciclo del cual el jugador no puede escapar, pero desde el cual existe un camino hacia un estado ganador, se considera ganador.

2.2. Director

En particular, buscamos como solución al problema de control, controladores que sean directores, como en [REFS Huang]. 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 **poner a SUP?** 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 [REF](#).
- La construcción de directores puede requerir una menor exploración de la planta que la construcción de un supervisor. 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 [\[REF 15 paper\]](#) 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 [3] 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.

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 monolítico (que resuelve toda la planta ya compuesta).

```

Algorithm classicalSolver( $E, A_E^C$ ):
   $B = \{s \in S_E \mid \nexists w. s \xrightarrow{w} m \wedge m \in M_E\}$ 
   $B' = \emptyset$ 
  while  $B' \neq B$ :
     $B' = B$ 
     $B = B \cup \{s \in S_E \mid \text{forcedTo}(s, e, E) \wedge e \in B\}$ 
  return  $\bar{e} \notin B$ 

```

Listing 2.1: Algoritmo Monolitico

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 la idea del 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 3.

2.4. 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 un LTS totalmente explorado.

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 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 2.2 mostramos la estructura básica de este método. 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 perdedores o ganadores.

```

Algorithm genericOTF-Exploration( $E, A_E^C$ ):
   $\bar{e} = \langle \bar{e}^0, \dots, \bar{e}^n \rangle$ 
   $ES = initial$  // la parte conocida de la planta
  while  $initial \notin Goals \cup Errors$ :
     $(e, \ell, e') = proxTransicion(ES)$ 
    expandirES( $ES, (e, \ell, e')$ )
    if  $e' \in Errors$ :
      propagarError( $e'$ )
    else if  $e' \in Goals$ :
      propagarGoal( $e'$ )
    else if isLoop( $e, e'$ ):
      if nuevoLoopGanador( $e, e'$ ):
        propagarGoal( $e'$ )
      else if nuevoLoopPerdedor( $e, e'$ ):
        propagarError( $e'$ )

  if  $initial \in Goals$ :
    return armarControlador( $Goals$ )
  else:
    return "UNREALIZABLE"

```

Listing 2.2: Algoritmo Monolitico

Para incrementar las ramas podadas se utiliza una heurística de exploración Best First Search [2] que busca ganar controlablemente o perder no controlablemente, para garantizar con la menor exploración posible que el estado actual es ganador o perdedor.

En el peor caso, 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 3.

En todos los gráficos siguientes vamos a utilizar las letras c , u , para denotar si una

transición es controlable o no, respectivamente; en caso de no especificar letra para una transición es porque su controlabilidad no afecta el resultado del ejemplo.

2.4.1. marcado explícito errores

Gracias a esta suite de tests pudimos encontrar que la exploración fallaba en tres puntos importantes:

- Falencias al encontrar errores
- Propagación local
- Falta de completitud en la exploración en casos donde era necesario seguir.
- detección de estados ganadores poco robusta.

En cuanto a agregar estados al conjunto *Errors* la inadvertencia se debía a que no sacaba conclusión alguna al haber explorado todo un sub-autómata, por ende al propagar información desde otra rama se podría llegar a un resultado erróneo.

Para comprender mejor observar la figura 2.1 donde desde el estado *e* tenemos dos sub-ramas a explorar. 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*.

Cabe destacar que esto era posible debido a que no se requería que un estado hijo tenga conclusión¹ para seguir propagando, es decir, bastaba con que haya sido explorado alguna vez y se asumía lo mejor.

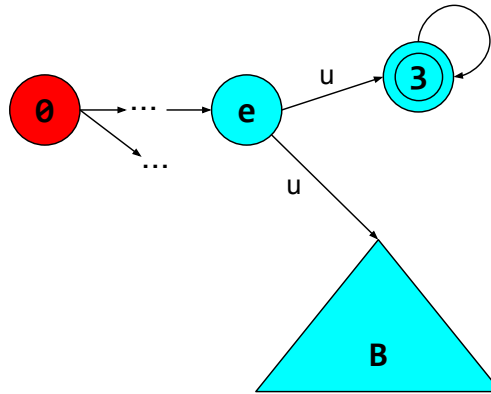


Fig. 2.1: Caso no controlable que anteriormente propagaba Goal

2.4.2. Propagación local vs por conjuntos

Por otro lado, al propagar tenía una mirada local, perdiendo información sobre lo que sucede dentro del conjunto. Es así que no podía reconocer casos donde, por ejemplo: hay un loop controlable entre dos estados, el cual se explora primero, y uno de ellos va

¹ Es decir, esté en el conjunto *Goals* o *Errors*

controlablemente a un error. En este caso es obvio que todo debe ser error pero según la mirada local ambos tienen *una forma de escapar del error*, el otro estado del conjunto. Para una aclaración visual ver la figura 2.2a.

Equivalentemente tampoco funciona reconociendo *Goals*, en la figura 2.2b se puede ver un ejemplo. El estado 2 llega al estado marcado 3, pero no puede forzarlo. Por ser non-blocking esto no nos molestaría y el modelo debería ser controlable. Pero si miramos localmente al propagar *Goal* desde 3, no sabemos dónde nos lleva la transición no controlable, y deberíamos suponer lo peor.

En conclusión, es difícil decidir dónde hacer el corte. Son muchos casos y no se puede, localmente, distinguirlos a todos. Por ende es necesario un algoritmo más inteligente, con una mirada global del conjunto a propagar.

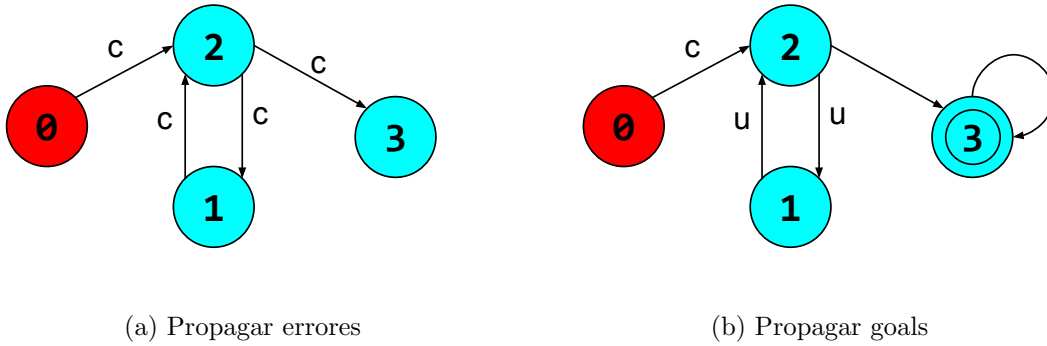


Fig. 2.2: Problemas de propagación local.

2.4.3. Completitud de exploración

Respecto a la falta de completitud, queda claro que teniendo una conclusión para cada uno de los estados hijos del inicial podemos definir si el problema es o no controlable. Lo que sucedía era que al tener conclusiones erróneas y problemas de propagación había ciertos casos donde, según el algoritmo de exploración el problema era controlable pero al llegar al constructor del controlador se daba cuenta que había estados a los cuales les faltaba exploración y que, de hecho, tenían transiciones no controlables a estados sin mirar. Llegado ese punto devolvía que no había controlador, cuando de seguir explorando hubiese visto que lo que faltaba era algo ganador. Un ejemplo particular de esto puede verse en la figura 2.3, al terminar la exploración se concluyó que era controlable, sin embargo al querer construir el controlador se descubre que el estado 1 tiene una no controlable por explorar, devolviendo entonces que no existe controlador.

2.4.4. Correcta detección de loops ganadores

La detección de loops con estados ganadores es una pieza central del problema, y tampoco puede solucionarse con una mirada local. Originalmente, la primera versión del pseudocódigo daba una descripción declarativa de los estados a encontrar (ver listing 2.3). Esta descripción de los estados en un loop ganador es correcta, pero no resulta claro como implementarla. En particular, fue claro al incrementar la batería de tests que la detección de loops ganadores necesitaba una modificación.

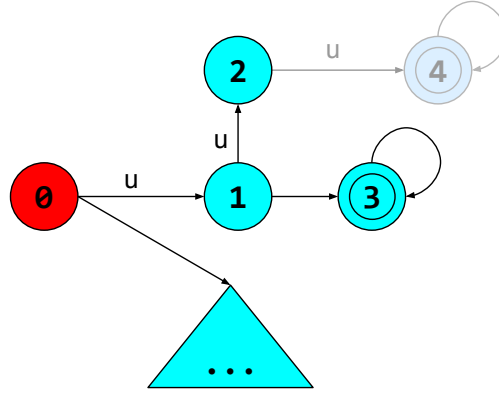


Fig. 2.3: Ejemplo de falta de completitud, los estados en gris son los faltantes por explorar.

Luego de varios intentos más veloces (como ejemplo ver el pseudocódigo de `findNewErrorsIn(loops)`) 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. Como se verá más adelante, corremos el algoritmo clásico sobre una versión optimista que asume que toda transición no explorada es perdedora, y de los estados ya explorados solo tomamos en cuenta un grupo reducido que forma un loop sobre la última transición explorada. Este enfoque otorga la completitud del algoritmo tradicional mientras que sostiene la eficiencia de la exploración on-the-fly.

```

function buildMCCC(e, e'):
  let C such that
    C = {e_i | (e'  $\xrightarrow{w}$  ES e_i  $\xrightarrow{w'}$  ES e  $\vee$  e  $\xrightarrow{w}$  ES e_i  $\xrightarrow{w'}$  ES e')  $\wedge$  extendsCCC(e_i, C  $\cup$  Goals)  $\wedge$ 
      ( $\exists w. e \xrightarrow{w}$  ES e_m  $\wedge$  e_m  $\in$  M_E  $\cap$  (C  $\cup$  Goals))}
  return C

function extendsCCC(e, C):
  return ( $\exists \ell. e \xrightarrow{\ell}_E e' \wedge e' \in C$ )  $\wedge$  ( $\forall \ell_u \in A_U. e \xrightarrow{\ell_u}_E e' \Rightarrow e' \in C$ )

```

Listing 2.3: vieja descripción estados ganadores

2.4.5. 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, por su nombre, no garantiza siempre elegir el mejor camino posible, sino solo la mejor aproximación que encuentre. Es en esa correctitud independiente de la heurística donde nuestro trabajo hizo foco.

El proyecto MTSA inicialmente contaba con dos heurísticas BFS para exploración, *Monotonic Abstraction* y *Ready Abstraction*. Ambas cumplen su función en síntesis de controlador con exploración parcial, pero presentaban un problema.

El algoritmo de exploración había sido desarrollado en conjunto con las heurísticas y si bien esto ayudaba a la eficiencia del mismo, no resultaba agnóstico a las mismas. 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 correctitud ni completitud.

3. NUEVO DIRECTED CONTROLLER SYNTHESIS

En esta sección presentamos el nuevo algoritmo DCS, 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 DCS, demostramos los lemas presentados, detallamos la complejidad de nuestro algoritmo y presentamos las ideas que llevaron a diseñar el algoritmo presentado.

3.1. Propuesta de nuevo algoritmo

```

1  function DCS( $\mathcal{E}=(E, A_E^C)$ , heuristic):
2     $\bar{e} = \langle \bar{e}^0, \dots, \bar{e}^n \rangle$ 
3     $ES = (\{\bar{e}\}, A_E, \emptyset, \bar{e}, M_E \cap \{\bar{e}\})$ 
4    Goals = Errors = Witnesses =  $\emptyset$ 
5    None =  $\{\bar{e}\}$ 
6    if (isDeadlock( $\bar{e}$ )):
7      Errors =  $\{\bar{e}\}$ 
8      None =  $\emptyset$ 
9    while  $\bar{e} \notin \text{Errors} \cup \text{Goals}$ :
10     ( $e, \ell, e'$ ) = expandNext(heuristic)
11      $S_{ES'} = S_{ES} \cup \{e'\}$ 
12      $ES' = (S_{ES'}, A_E, \rightarrow_{ES} \cup \{e \xrightarrow{\ell} e'\}, \bar{e}, M_E \cap S_{ES'})$ 
13     if  $e' \in \text{Errors}$ :
14       propagateError( $\{e'\}$ )
15     else if  $e' \in \text{Goals}$ :
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  $\cup (C \cap M_{ES'})$ 
22         Goals = Goals  $\cup C$ 
23         None = None  $\setminus C$ 
24         propagateGoal(C)
25       else:
26         P = findNewErrorsIn(loops)
27         Errors = Errors  $\cup P$ 
28         None = None  $\setminus P$ 
29         propagateError(P)
30      $ES = ES'$ 
31
32   if  $\bar{e} \in \text{Goals}$ :
33      $r = \text{rankStates}(ES)$ 
34     return  $\lambda w. \{ \ell \mid \bar{e} \xrightarrow{w} e \xrightarrow{\ell} e' \wedge e' \in \text{Goals} \}$ 
35      $\wedge (\ell \in A_E^C \Rightarrow \ell = \text{bestControllable}(s, r, ES))$ 
36   else:
37     return UNREALIZABLE

```

Listing 3.1: On-the-fly Directed Exploration Procedure.

```

function propagateGoal(newGoals):
  C' = ∅; C = ancestorsNone(newGoals)
  while C' ≠ C:
    C' = C
    C = C \ {s ∈ C |
      isForcedToLose(s, C) ∨
      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 |
      (∃e ∈ Errors . forcedTo(s, e, ES'_T)) ∨
      cannotReachGoalIn(s, C)}
  P = P \ C
  Errors = Errors ∪ P
  None = None \ P

```

Listing 3.2: Status propagation procedures.

```

function findNewGoalsIn(loops):
  C = loops; C' = ∅
  while C' ≠ C:
    C' = C; C'' = ∅
    while C'' ≠ C:
      C'' = C
      C = C \ {s ∈ C |
        isForcedToLose(s, C) ∨
        cannotBeReached(s, C)}
    C = C \ {s ∈ C | cannotReachGoalOrMarkedIn(s, C)}
  return C

function findNewErrorsIn(loops):
  if (∃s ∈ loops . s  $\xrightarrow{e'}$  ES'_T s' ∧ (s' ∉ loops ∧ s' ∉ Errors)):
    return ∅
  else:
    return loops

```

Listing 3.3: Status confirmation.


```

procedure expandNext(heuristic):
  let (e, ℓ, e') . e ∈ SES ∧ e  $\xrightarrow{\ell}_E$  e' ∧ ¬e  $\xrightarrow{\ell}_{ES}$  e' ∧
    (∀s, ℓ', s') . s ∈ SES ∧ s  $\xrightarrow{\ell}_E$  s' ∧ ¬s  $\xrightarrow{\ell}_{ES}$  s' ⇒
      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  $\xrightarrow{w}$  e' ∧
    ∄s ∈ w(e) . s ≠ e' ∧ s ∈ Goals ∪ Errors}

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

function getMaxLoop(e, e'):
  return {s | ∃w, w' . e  $\xrightarrow{w}$  s ∧ s  $\xrightarrow{w'}$  e' ∧
    ∄s' . (s' ∈ w(e) ∨ s' ∈ w'(s)) ∧ s' ≠ e' ∧ s' ∈ Goals ∪ Errors}

function forcedTo(s, e, Z):
  return (∃ℓu ∈ AZU . s  $\xrightarrow{\ell_u}$  Z e) ∨
    (∀ℓc ∈ AZC . s  $\xrightarrow{\ell_c}$  Z e' ⇒ e' = e)

function isForcedToLose(s, C):
  return ∃e . forcedTo(s, e, ES⊥) ∧ e ∉ (C ∪ Goals)

function cannotBeReached(s, C):
  return ∄s' ∈ C, ∃ℓ . s'  $\xrightarrow{\ell}_{ES'}$  s

function cannotReachGoalOrMarkedIn(s, C)
  return ∄w . s  $\xrightarrow{w}_C$  s' ∧ s' ∈ C ∧
    (canReachInOneStep(s', ES, Goals)
    ∨ s' ∈ MES')

function cannotReachGoalIn(s, C)
  return ∄w . s  $\xrightarrow{w}_C$  s' ∧ s' ∈ C ∧
    canReachInOneStep(s', ES, Goals)

function canReach(s', s)
  return ∃w . s'  $\xrightarrow{w}_{ES'}$  s

function canReachInOneStep(s, Targets)
  return ∃ℓ . s  $\xrightarrow{\ell}_{ES'}$  s' ∧ s' ∈ Targets

function isDeadlock(s)
  return ∄ℓ . s  $\xrightarrow{\ell}_E$  s'

```

Listing 3.4: auxiliary procedures.

```

function rankStates( $ES$ )
   $r = 0$ ;  $W' = \text{Witnesses}$ ;  $W = \emptyset$ 
  while  $W' \neq W$  :
     $\forall w \in W', \text{rank}(w) = r$ 
     $W = W \cup W'$ 
     $W' = \{s \in (\text{Goals} \setminus W) \mid \exists s' \in W . s \rightarrow_{ES} s'\}$ 
     $r = r + 1$ 
  return rank

function bestControllable( $e, r, ES$ )
  return  $\ell \in A_E^C . e \xrightarrow{\ell}_{ES} e' \wedge \nexists \ell' \in A_E^C .$ 
            $e \xrightarrow{\ell'}_{ES} e'', r(e'') \leq r(e')$ 

```

Listing 3.5: ranking procedures.

3.2. Demostración de correctitud y completitud

Notación 2: Decimos que un estado s es ganador[”winning”] (resp. perdedor[”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} .

El algoritmo (ver Listing 3.1) explora incrementalmente el espacio de estados de E utilizando una estructura de exploración parcial (ES), añadiéndole una transición por vez.

Notación 3: 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 $\exists w_0 \cdot \exists w_1 \cdot w = w_0.w_1$ y $s' \xrightarrow{w_0} \dots \xrightarrow{w_1} s$

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 $S_{ES} \subset S_E$.

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 (\perp) asumiendo que las transiciones no exploradas llevan a estados perdedores.

Definición 6 (Problemas de Control \top y \perp): 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 $\rightarrow_\top = \rightarrow_{ES} \cup \{(s, \ell, \top) \mid \exists s' \cdot (s, \ell, s') \in (\rightarrow_E \setminus \rightarrow_{ES})\} \cup \{(\top, \ell, \top) \mid \ell \in A_E\}$

Definimos \mathcal{E}_\perp como (ES_\perp, A_E^C) donde $ES_\perp = (S_{ES} \cup \{\perp\}, A_E, \rightarrow_\perp, \bar{e}, M_E \cap S_{ES})$ y $\rightarrow_\perp = \rightarrow_{ES} \cup \{(s, \ell, \perp) \mid \exists s' \cdot (s, \ell, s') \in (\rightarrow_E \setminus \rightarrow_{ES})\}$

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

Propiedad 1 (Invariante): El loop principal del Algorithm 3.1 tiene el siguiente invariante:
 $ES \subseteq E \wedge \forall s \in ES . (s \in Goals \Leftrightarrow s \in W_{ES\perp}) \wedge (s \in Errors \Leftrightarrow s \in L_{ES\top}) \wedge s \in Errors \uplus Goals \uplus None$

La explicación del Algorithm 3.1 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 $e \xrightarrow{E} e'$ garantizando que e ya se encontraba en ES y $e \in None$. 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 $ES\perp$ (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 $ES'\perp$ o perdedores en $ES'\top$. Tanto `propagateGoal` como `propagateError` realizan un punto fijo estándar [?] sobre $ES\perp$ 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 $e \xrightarrow{ES} e'$ la única diferencia entre dos exploraciones parciales, ES y ES' , de E . Si $s \notin (W_{ES\perp} \cup L_{ES\top})$ y $s \in (W_{ES'\perp} \cup L_{ES'\top})$, entonces hay $s_0, \dots, s_n \notin (W_{ES\perp} \cup L_{ES\top})$ tal que $s = s_0 \wedge s_0 \xrightarrow{ES} \dots s_n \xrightarrow{ES} e'$.

Ya en la línea 17 sabemos que e' no es ganador en $ES\perp$ ni perdedor en $ES\top$, determinamos si $e \xrightarrow{ES} e'$ 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, $e' \notin (W_{ES'\perp} \cup L_{ES'\top})$ ya que cualquier controlador para e' en $ES'\perp$ (resp. $ES'\top$) es también un controlador en $ES\perp$ (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 $e \xrightarrow{ES} e'$ la única diferencia entre dos exploraciones parciales, ES y ES' , y $e' \notin L_{ES\top} \cup W_{ES\perp}$. Si $W_{ES'\perp} \neq W_{ES\perp}$ entonces $e' \in W_{ES'\perp} \setminus W_{ES\perp}$, 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 $e' \in W_{ES'\perp} \uplus L_{ES'\top}$, y por Lemma 2 propagar cualquier información nueva de e' a sus predecesores.

En la línea 18 computamos *loops*, el conjunto de estados que pertenecen a un loop que pasa por $e \xrightarrow{ES} e'$ y nunca por $W_{ES\perp} \cup L_{ES\top}$. 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 línea 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: $e' \in W_{ES'\perp}$ o $e' \in L_{ES'\top}$ (ver Lemma 4).

Lema 4: (*Condición necesaria/suficiente para ganar/perder*) Sea $e \xrightarrow{ES} e'$ 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 $\text{canBeWinningLoop}(\text{loops})$. Además, si $\text{canBeWinningLoop}(\text{loops})$ entonces $e' \notin L_{ES'_\top}$.

Si $\text{canBeWinningLoop}()$ retorna true, en la línea 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 $\text{canBeWinningLoop}()$ 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 $\text{loops} = \text{getMaxLoop}(e, e') \wedge \neg \text{canBeWinningLoop}(\text{loops})$ y $P = \text{findNewErrorsIn}(\text{loops})$ 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 líneas 33 a 35 extraen un director del conjunto de estados ganadores (*Goals*). Usamos el mismo punto fijo descrito en el Algorithm 4 de [?] para computar un ranking (línea 33) y una función determinística (línea 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 DCS 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 [?] y el hecho de que en el peor caso todas las transiciones son agregadas a la estructura de exploración. Entonces, $E = ES = ES_\perp = ES_\top$.

3.3. Nuestro enfoque

En esta sección presentaremos las ideas y conceptos que tuvieron un rol central en el diseño del nuevo algoritmo, con la intención de dar una intuición sobre el enfoque elegido, más allá de las demostraciones técnicas.

En función de obtener un algoritmo correcto, completo y agnóstico a la heurística, atacamos los problemas mencionados en el capítulo anterior y agregamos un *invariante* que mantenemos a lo largo de la exploración.

Propagación no-local: Al momento de propagar resultados, tanto estados *Goals* como *Errors* recurrimos a un punto fijo. 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.

Completitud de la exploración: Nos pareció esencial resolver toda la clasificación necesaria de ganadores y perdedores durante la exploración. Llegado el momento en el que se intenta comenzar a construir el controlador queremos asegurar que no es necesario volver a explorar más transiciones. También queremos asegurar que cuando concluimos que el estado inicial es ganador vamos a poder construir un controlador. Es decir, al momento de la construcción no podemos encontrar información nueva que nos haga cambiar de opinión.

Para esto, los lemas detallados en la próxima sección aseguran que un estado marcado como ganador o perdedor lo es en la planta compuesta totalmente explorada, y nunca puede cambiar su estado. También probamos que al terminar de explorar, el estado inicial está marcado como ganador o perdedor; nunca podemos terminar la exploración porque no hay más transiciones a explorar pero no llegamos a una conclusión sobre el estado inicial.

Encontrar errores: La detección de errores tiene tres grandes casos, de los cuales dos son sencillos pero uno trae grandes problemas que lo ponen al mismo nivel de la detección de ganadores. El caso de un *deadlock*, un estado sin transiciones salientes es trivial. Luego la propagación de errores, la clasificación de s como perdedor porque no puede escapar de caer en estados perdedores cuando toma una transición no trae dificultades. El problema radica en detectar los ciclos de estados que pueden extender sus palabras infinitamente pero nunca alcanzarán un estado marcado infinitas veces.

Sólo podemos concluir que un ciclo es error cuando este ha sido explorado en su completitud y los descendientes que salen del ciclo han sido clasificados. Esto es así debido a la naturaleza optimista de los problemas non-blocking. Si se tiene un ciclo con ningún estado marcado, y todos tienen transiciones controlables a perdedores, pero un solo estado s tiene un camino para llegar a un estado ganador, todos los estados del ciclo son ganadores.

Fue esta necesidad y dificultad para marcar errores lo que nos llevó a incluir el siguiente invariante para la síntesis on-the-fly.

Invariante: Intentando resolver el problema del marcado de errores, buscamos una separación fuerte entre los estados $error = L_E$ y los estados para los cuales no tenemos suficiente información para clasificar en este momento *None*.

Como se verá en la propiedad 1, un estado s solo puede seguir sin clasificar (siendo *None*) si, con los explorados hasta el momento, y siendo totalmente optimistas sobre las transiciones desconocidas no podemos asegurar que s está condenado a ser perdedor (y tampoco podemos concluir que es ganador).

Al momento de haber explorado todos los descendientes de s , incluso si no se exploró toda la planta total, es claro que no importa si se es optimista (\top) o pesimista (\perp) para saber si s es un estado ganador o perdedor, por lo que nos forzamos por nuestro invariante a clasificarlo. Con esto evitamos las ramas totalmente exploradas pero sin clasificar que traían complicaciones en la figura 2.1 y solo permitimos que un estado sea *None* si tiene un camino a una transición no explorada.

findGoals en planta reducida: Para detectar los loops ganadores, el enfoque con mejores resultados fue el de un punto fijo clásico. La diferencia consiste en el uso del

problema pesimista y en el uso de los lemas para solo analizar los estados que forman loops de estados NONE con la última transición explorada.

3.4. 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 $w_0.\ell \in \mathcal{L}^{\sigma}(Z_{\perp})$.

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 válido 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 éstos 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 $\text{canBeWinningLoop}(\text{loops})$, asumimos

$\neg \text{canBeWinningLoop}(\text{loops})$ y mostramos que $e' \notin W_{ES'_\perp} \setminus W_{ES_\perp}$. Para esto, basta con ver que si $\neg \text{canBeWinningLoop}(\text{loops})$ entonces para alcanzar un estado marcado desde e' se debe salir de loops a un estado $s \notin \text{loops} \cup W_{ES_\perp}$ 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 $\text{canBeWinningLoop}(\text{loops})$ 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 $\text{loops} \cap L_{ES_\top} = \emptyset$. 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 $e' \in W_{ES'_\perp} \setminus W_{ES_\perp}$ pero $\neg \text{canBeWinningLoop}(\text{loops})$. Existe un controlador σ para e' en ES'_\perp , esto significa que debe existir un camino w desde e' hasta un estado marcado m . Dado que $\neg \text{canBeWinningLoop}(\text{loops})$, 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 $L_{ES_\top} \cup \text{None}$, y no tiene un camino de estados *None* hasta e' entonces según Lemma 2 s va a seguir sin cambiar su estado. Dado que $s \notin W_{ES'_\perp}$, s no tiene un controlador en ES'_\perp , pero σ acepta corridas que llevan a s , llegamos a un absurdo.

Asumiendo $\text{canBeWinningLoop}(\text{loops})$ 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 $s' \notin \text{loops} \wedge s' \notin L_{ES_\top}$ y $e' \xrightarrow{w_0}_{ES'_\top} s'$. En otro caso $e' \xrightarrow{w}_{ES'_\top} p \wedge p \in \text{loops}$ entonces para toda ℓ' controlable, $\ell' \in \sigma^4(w)$ si y solo si $\exists p'. p \xrightarrow{\ell'} p' \wedge p' \notin L_{ES_\top}$.

Usamos los controladores $\sigma_{s'}$ donde s' es tal que existe $s \in \text{loops}$ y $s \xrightarrow{\ell'}_{ES'_\top} s' \wedge s' \notin \text{loops} \wedge s' \notin L_{ES_\top}$. 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 $s' \notin \text{loops} \wedge s' \notin L_{ES_\top}$ 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 $p' \notin \text{loops}$ entonces $\sigma^4(w.\ell') = \sigma_{p'}(\lambda)$ y, como antes, sabemos que $\sigma_{p'}$ es *non – blocking* entonces $w.\ell'$ puede extenderse para llegar a un estado marcado.

Si $p' \in \text{loops}$, entonces $w.\ell'$ puede extenderse para llegar a cualquier s en loops . Sabemos que o existe un estado marcado en loops o algún estado en W_{ES_\perp} es alcanzable en un paso desde loops , de cualquier forma podemos extender $w.\ell'$ 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 $\text{loops} \cap L_{ES_\top} = \emptyset$. 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 $P = \text{loops} \subseteq L_{ES'_\top}$. Alcanzamos una contradicción asumiendo que $s \in \text{loops} \setminus L_{ES'_\top}$: 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 $s' \notin \text{loops}$. 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 $s' \notin \text{loops}$ tal que $\exists s \in \text{loops} . s \xrightarrow{l} s'$, puede o ser y seguir siendo un estado perdedor ($s' \in LES \wedge s' \in LESS$) 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 $s' \notin \text{loops} \wedge s' \notin L_{ES_\top}$ 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 $\nexists s \in \text{loops} . s \xrightarrow{\ell'}_{ES'_\top} s' \wedge (s' \notin \text{loops} \wedge s' \notin \text{Errors})$ probamos que $\forall s \in \text{loops}, s \in L_{ES'_\top}$.

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 $s \xrightarrow{w_0}_{ES'_\top} s' \wedge s' \notin \text{loops}$. Dado que $s' \in \text{Errors} \Rightarrow s' \in L_{ES_\top}$ 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 $\forall s \in \text{loops}, s \in L_{ES'_\top}$.

□

3.5. Complejidad computacional

La complejidad del algoritmo en el Listing 3.1 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`, `propagateGoal`, `canReach`, `getMaxLoop`, `canBeWinningLoop`, `findNewGoalsIn`, and `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. $O(\text{findNewGoalsIn}(\text{loop})) \leq O(|S_{ES}| \times |T_{ES}|)$: 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. $O(\text{findNewErrorsIn}(\text{loops})) \leq O(|S_{ES}| \times |A_E|)$: 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 , $|hijos(s)| \leq |A_E|$ (la cantidad de eventos). Además, la cantidad de estados del `loops` está acotada por S_{ES} .
3. $O(\text{propagateGoals}(\text{newGoals})) \leq O(|S_{ES}| \times |T_{ES}|)$: 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 $O(|S_{ES}| \times O(\text{ iteración}))$.

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 conclusión $O(\text{ iteración}) = |T_{ES}|$ y $O(\text{propagateGoals}(\text{newGoals})) \leq O(|S_{ES}| \times |T_{ES}|)$.

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

5. $O(\text{canReach}(e, e')) = O(|T_{ES}|)$: Simplemente requiere una búsqueda en anchura sobre las transiciones salientes.
6. $O(\text{getMaxLoop}(e, e')) = O(|T_{ES}|)$: 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. $O(\text{canBeWinningLoop}(loops)) = O(|T_{ES}|)$: 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, [?] presentó una solución para construir un director optimal en $O(|X| \times |\Sigma|(|X_m - X_t| + 1))$ (donde $X = S_E$, $\Sigma = A_E$ y $X_m - X_t$ es el número de estados marcados que no son terminales. Luego, comparando [?] 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 [?]. 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 DCS el loop es ejecuta hasta $|T_E|$ veces, cada vez que a ES se le agrega una transición. Éste 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.

4. IMPLEMENTACIÓN

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

4.1. 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 4.1 volvemos al caso de estudio presentado en 1.1 esta vez escrito en la herramienta 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 $i \in \text{Servicio}$.

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 4.1: Ejemplo de LTS y composición

En el listing 4.2 vemos un ejemplo de cómo definir un controlador, en este caso lo llamamos *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 DCS, 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 4.2: Ejemplo de Controller y DCS

4.2. Heurísticas adicionales

4.2.1. Dummy

Como dijimos en el capítulo 2, el algoritmo de DCS debe ser agnóstico a la heurística. Al comenzar nuestro trabajo en el proyecto y una vez que pudimos generar cierto conocimiento sobre el pseudocódigo nos percatamos de casos borde que no iban a 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.

4.2.2. BFS

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 tempranamente a alguna conclusión sobre un estado ésta información se puede propagar, cortando más ramas y/o más grandes.

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

La segunda heurística que desarrollamos es un simple BFS. La razón es para mostrar qué tanto se pueden mejorar los tiempos del algoritmo con una buena heurística. Esto se puede ver en detalle en el capítulo 5, donde mostramos resultados de un mismo benchmark para las diferentes heurísticas.

4.3. 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. Si se desea ver la especificación de cada uno de los tests puede encontrarse en el repositorio de MTSA².

En total tenemos una batería de 51 tests. Los cuáles no fueron desarrollados uno detrás del otro sino que utilizamos una técnica llamada TDD (Test Driven Development). Al comienzo se nos fue presentado el algoritmo original y, luego de mirar en detalle tanto el pseudocódigo como la implementación, notamos algunos casos que quizás resolvería mal; fue entonces que iniciamos la batería de tests. 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.

Como recordatorio resumimos los problemas encontrados en tres grandes grupos.

- Falencias al encontrar errores
- Propagación local
- Falta de completitud

Es necesario entrar un poco en detalle sobre la implementación del algoritmo. Para manejar la exploración de manera ordenada contamos con una cola de estados “open”, en ella colocamos los estados que están esperando para ser explorados. Notar que un estado puede salir y entrar de “open” varias veces, ya que podríamos estar explorando otra transición, pero sólo una “cantidad de transiciones” de veces. Muchos de los problemas intermedios fueron a la hora de reabrir estados, para que puedan seguir siendo explorados, esto se traducía en una finalización temprana de la exploración llegando a una conclusión incorrecta.

Dentro de esta batería los siguientes casos son de especial interés.

Test 1 Fig 4.1 Ataca el problema de falta de completitud. Luego de encontrar el primer loop no controlable (u12, u21), al ver el nodo inicial nuevamente, propaga mal lo que en la versión anterior era conocido como WEAK, es decir, sabemos que está activo pero no tenemos datos suficientes para cerrarlo. El problema es que no vuelve a poner al nodo inicial en la cola de exploración, por ende no ve el otro camino, que lleva al estado marcado (y finalmente Goal).

Test 8 Fig 4.2 Es un problema de falencias al encontrar errores ya que no señala el estado 3 como error, porque no detecta que es un livelock. Notar que esto sería lo mismo que si tenemos una rama totalmente explorada a partir de u33.

TEST 12, este puede ser otro problema? porque daba no controlable, en realidad lo es pero mepa que no detectaba que podía cortar la controlable porque está marcada?

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

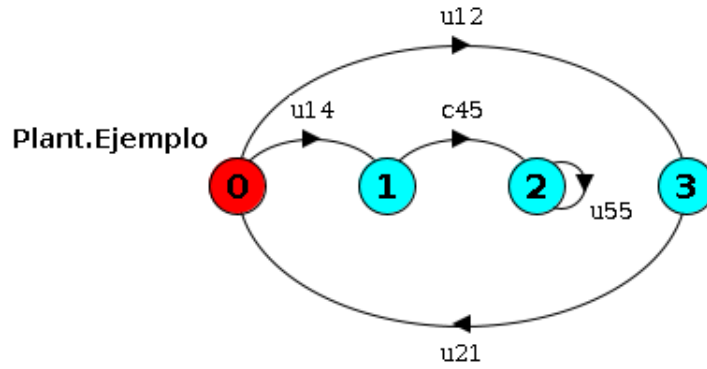


Fig. 4.1: LTS del test 1

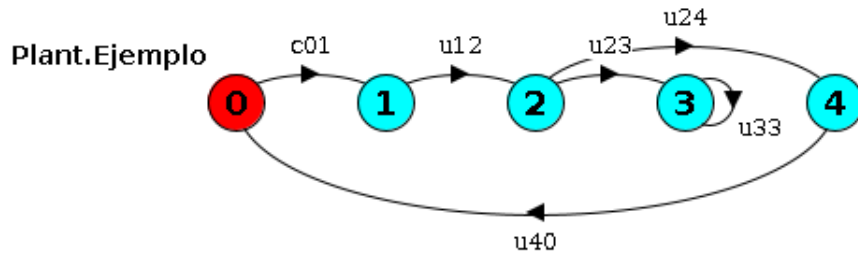


Fig. 4.2: LTS del test 8

Test 26? este puede ser interesante porque son dos loops no controlables pero ganas en ambos y dice “Caso para intentar mostrar que ver solo loops minimales no sirve”.

TEST 30, ACÁ ESTÁ LA PAPA. otro caso de “propagación local no sirve”. 2 y 3 son goal, pero al querer propagar a 0 dice “uh esta no controlable me lleva a algo de estatus desconocido” entonces no marca nada y termina diciendo que no es controlable cuando sí.

Test 35 El tema es que arma un CCC y no chequea que sea válido bien, porque ya no hay loop con marcado. Entonces el build Controller se queja y da que no hay controlador. Pero si revisaba para el otro lado había un CCC válido

```

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 4.3: Test 1 a modo de ejemplo

5. PERFORMANCE

Para realizar las pruebas de performance decidimos usar el mismo conjunto de problemas que el utilizado para examinar el algoritmo original de DCS 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 re-evaluación 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.

5.1. Comparación con versión previa de dcs

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 DCS.

La comparación se realizó para las dos heurísticas desarrolladas en [2], Monotonic Abstraction y Ready Abstraction. Agregamos además una de las heurísticas naïf, 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.

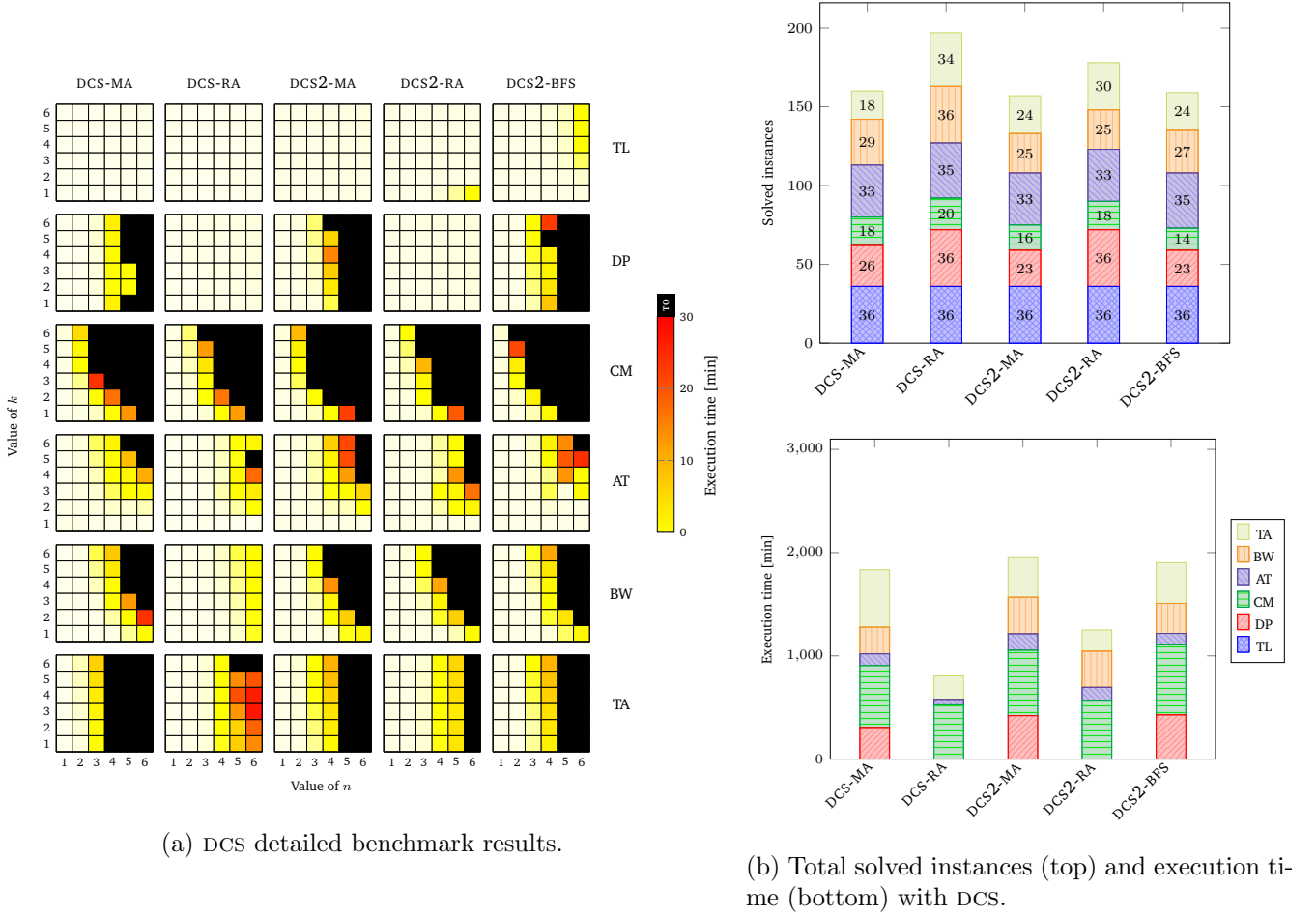


Fig. 5.1: Benchmark results.

En la figura 5.1 se pueden observar los resultados con respecto a DCS anterior. 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 naïf termina compitiendo contra MA. Esto tiene su explicación en el cambio de objetivo, ya que MA fue presentada en [REF reachability en paper] para *reachability*.

Nuestra versión (dcs2) 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 vemos que en general tienen problemas con las combinaciones más grandes (salvo MYND). Nuestra suposición es que en la versión anterior se están cerrando estados como *Goals* de manera anticipada y posiblemente errónea, quizás casos como el de ???. Esto puede hacer que se llegue a una conclusión apurada que posiblemente devuelva un controlador incorrecto.

5.2. Comparación con otros programas

Si bien nuestra exploración parece perder un poco de escalabilidad en ciertos casos, en la figura 5.2 podemos ver que seguimos 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 DCS.

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

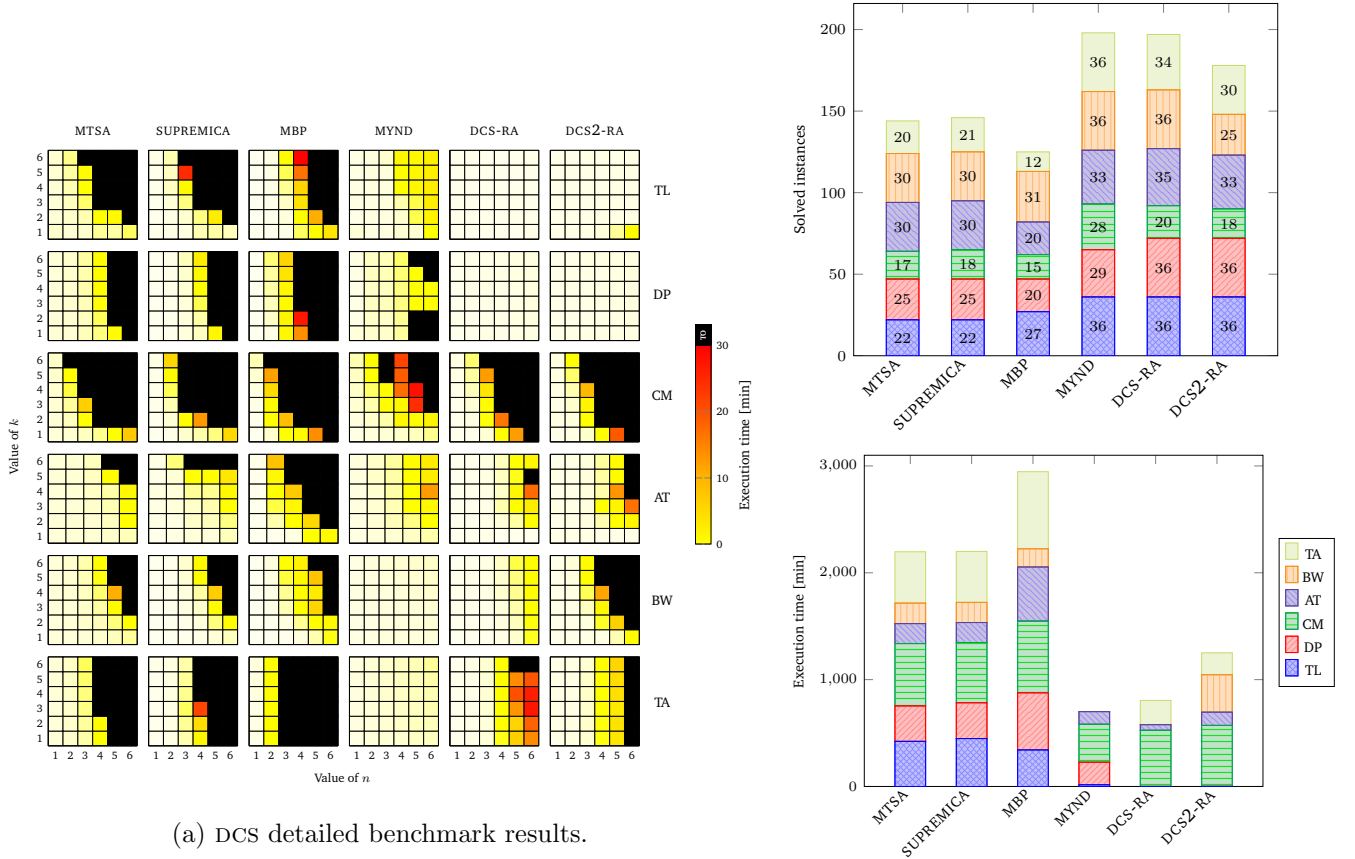


Fig. 5.2: Benchmark results.

6. 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:

- 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.
- Un nuevo algoritmo de exploración, cuya correctitud es agnóstica a la heurística utilizada.
- Una prueba de la correctitud y completitud del algoritmo presentado.
- Resultados experimentales para comprobar que las modificaciones a la exploración siguen manteniendo la buena performance de la técnica.

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] 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.