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

Poner aca el abstract (aprox. 200 palabras).

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

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. Control Supervisado

Entiende el problema y tendrás la solución

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.

Un problema de Control Supervisado, dentro del área de AI Planning, consiste en un Sistema de Eventos Discreto (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 a la 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 bateria de tests, éstos no proveen una garantía si no 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.

1.2. Caso de estudio

A continuación se presenta un ejemplo sobre el cual aplicaremos nuestro algoritmo a lo largo de la presentación.

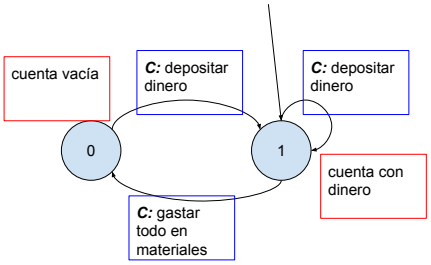
Imaginemos un negocio del cual modelamos 3 componentes: La cuenta de dinero, la estación de ventas de productos terminados, y la estación de compra de materia prima y ensamblaje de productos.

La cuenta de dinero puede estar vacía o con dinero, no contamos cuanto dinero hay, pero sabemos que en caso de estar vacía no se pueden comprar nuevas materias primas. También podemos elegir al momento de comprar nuevos materiales si queremos gastar todo el dinero que hay en la cuenta o solo gasta una parte.

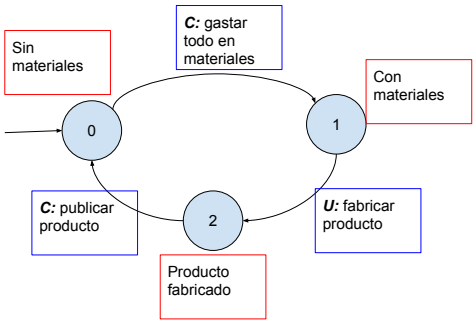
La estación de ensamble puede necesitar comprar nuevos materiales, y una vez que los recibe necesita un tiempo no controlable hasta finalizar el próximo producto, momento en el que el mismo se transfiere a la estación de ventas.

La estación de ventas, cuando tiene un producto para vender, tarda un tiempo no controlable hasta que llega un cliente que compra el producto, momento en el que la cuenta del negocio guarda la plata y seguramente no está vacía.

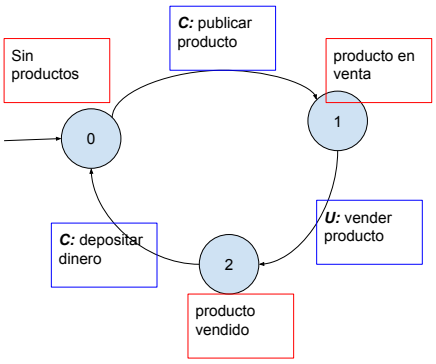
Mostramos en la figura 1.1 un LTS (Labeled transition system) para cada uno de los componentes descriptos.



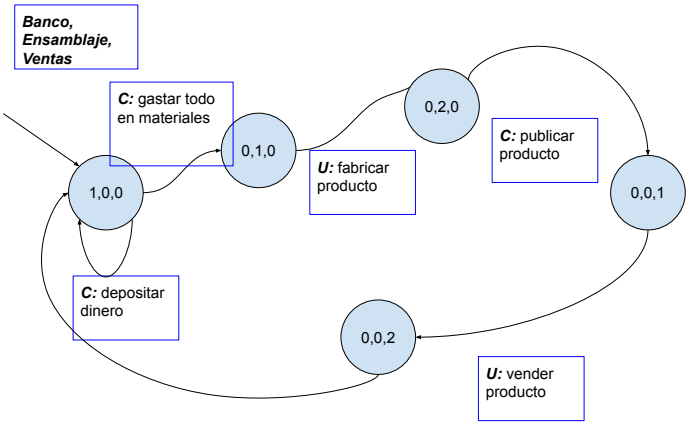
(a) Cuenta bancaria



(b) Estacion de ensamblaje



(c) Estacion de ventas



(d) Composición de los componentes

Fig. 1.1: Modelo de ejemplo

2. ANTECEDENTES

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

Definición 1 (Automata Deterministico): Un *automata deterministico* 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 $t_0 \overset{w}{\dashrightarrow}_T t_{k+1}$.

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

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):

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

2.1. Controlador objetivo

Definición 3 (Problema de Control Supervisado): Un *Problema de Control Supervisado* 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
- *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 $\bar{e} \overset{ww'}{\dashrightarrow}_E e_m$ con $e_m \in M_E$ (i.e., un estado marcado de E).

El problema a tratar consiste en encontrar para la planta de entrada, un controlador, es decir, un autómata con las siguientes características:

- sub-autómata de la planta: todos los estados y transiciones del controlador existen en la planta compuesta.

- controlable: todas las transiciones no controlables de la planta se encuentran en el controlador.

- non-blocking: cada palabra válida para el controlador puede ser extendida por otra palabra no vacía para que su concatenación alcance un estado marcado.

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 "loops", ya que los primeros estados ganadores son aquellos que están en un loop controlable con un estado marcado dentro. Luego señalizamos como ganadores también a cualquier estado que controlablemente alcanza un estado ganador.

También los "loops"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 "loop"del cual el jugador no puede escapar, pero desde el cual existe un camino hacia un estado ganador, se considera ganador.

2.2. Algoritmo monolítico

Una solución a este problema (o problemas similares), ampliamente estudiada[refs paper funcional, tesis dipi, buchiGames] se basa en un doble punto fijo. Ya que en la solución clásica se conoce toda la planta compuesta, se parte de la base de todos los estados marcados como objetivos, y a partir de ellos se desde que estados un controlador puede forzar a llegar a los estados marcados al menos una vez.

De los estados que no pueden forzar a los marcados, ya sabiendo que son estados donde gana el ambiente, se pueden retirar del punto fijo, y también a cualquier estado

Falta la descripción y el pseudo pero me maree

2.3. 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 un LTS totalmente explorado.

```
Algorithm classicalSolver(E, A_E^C):
        C = E
        C' = ∅
        while C' ≠ C :
                C' = C
                R = playerCanForce(C, C ∩ M_E)
                Tr = C \ R
                W = ambientCanForce(C, Tr ∪ (S_E \ (C ∪ Goals)))
                C = C \ W
        return C
```

Listing 2.1: Status confirmation.

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.

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, 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.

2.3.1. 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, 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` por el momento cuenta con dos heurísticas `BFS` para exploración, *Monotonic Abstraction* y *Ready Abstraction*. Ambas son muy buenas y logran gran eficiencia 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. PROBLEMAS ENCONTRADOS

3.1. Problemas encontrados

ADEMAS EL PROPAGATE NO PODIA SER LOCAL, EXPLICAR.

El anterior algoritmo de exploración tenía falencias en cuanto a agregar estados al conjunto *Errors*. Esto se debía a que no sacaba conclusión alguna al haber explorado todo un subgrafo, por ende al propagar información desde otra rama se podría llegar a un resultado erroneo. Para comprender mejor observar la figura (INGRESE FIGURA) donde desde el estado e tenemos dos sub-ramas a explorar. Si se mira primero la de abajo y no lo marcamos como error entonces al mirar la de arriba diremos que es goal y propagaremos dicha información, equivocadamente, más allá de e.

EXPLICAR QUE NUESTRO ALGORITMO ES AGNÓSTICO A LA HEURÍSICA (COMO DEBERÍA)

4. 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 una solución al problema composicional de "supervisory control". También discutimos la correctitud y completitud del nuevo algoritmo DCS.

4.1. Nuestro enfoque

Cabe aclarar que sólo podemos concluir que un loop es error cuando este ha sido explorado en su completitud. Esto es así debido a la naturaleza optimista de los problemas non-blocking.

Fue a causa de esta necesidad de marcar errores que decidimos diseñar un nuevo algoritmo con un invariante que consideramos clave para síntesis on-the-fly: Si con la información de lo explorado hasta el momento es posible concluir que un estado es ganador o perdedor en la planta totalmente explorada, debemos marcar ese estado antes de seguir explorando.

4.2. Propuesta de nuevo algoritmo

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

Listing 4.1: On-the-fly Directed Exploration Procedure.

```
function propagateGoal(newGoals):
  A = ancestorsNone(newGoals)
  C = playerCanForce(A, Goals)
  Goals = Goals ∪ C
  None = None \ C

procedure propagateError(newErrors):
  A = ancestorsNone(newErrors)
  P = ambientCanForce(A, Errors)
  Errors = Errors ∪ P
  None = None \ P
```

Listing 4.2: Status propagation procedures.

```
function findNewGoalsIn(loops):
  C = loops
  C' = ∅
  while C' ≠ C :
    C' = C
    R = playerCanForce(C, (C ∩ M_E) ∪ Goals)
    Tr = C \ R
    W = ambientCanForce(C, Tr ∪ (S_E \ (C ∪ Goals)))
    C = C \ W
  return C

function playerCanForce(C,Target): //Attr1
  K_0 = ∅
  K_1 = Target
  i = 1
  while K_i ≠ K_{i-1}:
    K_{i+1} = K_i
    for each k in K_i \ K_{i-1}:
      for each v in (In(k) ∩ C):
        if ∄ℓ_u . v --ℓ_u--> h ∧ h ∉ K_i then K_{i+1} = K_{i+1} ∪ {v}
return K_i

function ambientCanForce(C,Target): //Attr2
  K_0 = ∅
  K_1 = Target
  i = 1
  while K_i ≠ K_{i-1}:
    K_{i+1} = K_i
    for each k in K_i \ K_{i-1}:
      for each v in (In(k) ∩ C):
        if (∃ℓ_u . v --ℓ_u--> h ∧ h ∈ K_i) ∨ (∀ℓ_c . v --ℓ_c--> h ⇒ h ∈ K_i)
        then K_{i+1} = K_{i+1} ∪ {v}
return K_i

function findNewErrorsIn(loops):
  C = loops
  C' = ∅
  while C' ≠ C :
    C' = C
    R = playerCanForce(C, (C ∩ M_E) ∪ (S_E \ (C ∪ Errors)))
    Tr = C \ R
    W = ambientCanForce(C, Tr ∪ Errors)
    C = C \ W
  P = loops \ C
  return P
```

Listing 4.3: Status confirmation.

```
procedure expandNext(heuristic):
  let (e, ℓ, e') . e ∈ S_ES ∧ e -ℓ->_E e' ∧ ¬e -ℓ->_ES e' ∧
          (∀s, ℓ', s') . s ∈ S_ES ∧ s -ℓ->_E s' ∧ ¬s -ℓ->_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 -w->*_ES e' ∧
          ∄s ∈ w . s ∈ 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' ∈ w, w' . s' ≠ e' ∧ s' ∈ Goals ∪ Errors}

function forcedTo(s, e, Z):
  return (∃ℓ_u ∈ A^U_Z . s -ℓ_u->_Z e) ∨
          (∀ℓ_c ∈ A^C_Z . s -ℓ_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 . ∄w . s' -w->*_ES' s ∧ |w| > 0

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)
```

Listing 4.4: auxiliary procedures.

4.3. Demostración de corectitud y completitud

Notación 2: Decimos que un estado s es ganador["winning"] (resp. perdedor["losing"]) en el problema $\mathcal{E} = (E, A_E^C)$ siendo s el estado inicial de E si hay una (resp. no hay una) solución para $\mathcal{E}$. 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 4.1) explora incrementalmente el espacio de estados de E utilizando una estructura de exploración parcial (ES), añadiéndole una transición por vez.

Definición 4 (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 ($\bot$) asumiendo que las transiciones no exploradas llevan a estados perdedores.

Definición 5 (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 $ES_\top$. Lemma 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 $L_{ES_\top} \subseteq L_{ES'_\top}$.

Demostración 1: Para probar $W_{ES_\bot} \subseteq W_{ES'_\bot}$ mostramos que un supervisor para un estado s en $W_{ES_\bot}$ puede ser usado como un supervisor para s en $W_{ES'_\bot}$. 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 supervisor que s debe tener en $ES'_\top$ es también un supervisor para s en $ES_\top$. □

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 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 Algorithm 4.1 que detallamos a continuación sirve también como un esquema de demostración para Property 1.

Para empezar, notar que la función `expandNext` (line 7) 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 12) o perdedor en $ES_\top$ (line 10) 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 [**?**] sobre $ES_\bot$ y $ES_\top$ pero solo sobre predecesores de e' que están en *None*. Lemma 2 asegura la completitud de esta propagación restringida.

Lema 2: ***(Ganadores/Perdedores nuevos pueden alcanzar transiciones nuevas a traves de estados-None)*** 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 $s = s_0 \wedge s_0 \xrightarrow{\ell_0}_{ES} \ldots s_n \xrightarrow{\ell}_{ES} e'$.

Demostración 2: Si s no es un predecesor de e', como $e \xrightarrow{l}_{ES'} e'$ es la única diferencia entre ES y ES', entonces los decendientes de s son los mismos, por lo tanto sus posibles supervisores en $ES'_\top$ y $ES'_\bot$ no cambiaron. Entonces, $s \notin W_{ES'_\bot} \cup L_{ES'_\top}$ 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 $s' \in (W_{ES_\bot} \cup L_{ES_\top})$. Un supervisor σ de s en $ES_\top$ no va a alcanzar estados en $L_{ES_\top}$, por lo tanto todo s' que alcance va a tener un supervisor $\sigma_{s'}$ para $ES_\bot$. Usamos σ y $\sigma_{s'}$ para construir un supervisor para s en $ES'_\top$ para mostrar que $s \notin L_{ES'_\top}$. Un supervisor para s en $ES'_\bot$ no puede existir porque de otra forma podríamos usarlo para construir un supervisor para s en $ES_\bot$ usando un razonamiento similar al anterior. Esto significa que $s \in W_{ES_\bot}$ contradiciendo la hipótesis. □

Ya en la linea 14 sabemos que e' no es ganador en $ES_\bot$ ni perdedor en $ES_\top$, chequeamos si $e \xrightarrow{\ell}_{ES} e'$ cierra un nuevo loop. Si no es el caso, entonces no hay nada que hacer ya que e' alcanza las mismas transiciones en ES' que en ES. Entonces, $e' \notin (W_{ES'_\bot} \cup L_{ES'_\top})$ ya que cualquier supervisor para e' en $ES'_\bot$ (resp. $ES'_\top$) es también un supervisor en $ES_\bot$ (resp. $ES_\top$) y vice versa. 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'. Si $W_{ES'_\bot} \neq W_{ES_\bot} \Rightarrow e' \in W_{ES'_\bot} \setminus W_{ES_\bot}$, y si $L_{ES'_\top} \neq L_{ES_\top} \Rightarrow e' \in L_{ES'_\top} \setminus L_{ES_\top}$. ENTONCES?

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

Si asumimos $e' \notin L_{ES'_\top}$. 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 supervisor σ desde e' en $ES'_\top$. Como $s \notin L_{ES_\top}$ también debe haber un supervisor σ' en $ES_\top$. Construimos un nuevo supervisor para $ES'_\top$ desde s que funciona exactamente como σ' pero cuando alcanza $e \xrightarrow{\ell} e'$ se comporta como σ. Este nuevo supervisor prueba que $s \in L_{ES'_\top}$ lo cual es una contradicción. □

Si se cerró un nuevo loop(line 14), 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 15 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 supervisor 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 **??** 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 eploraciones parciales, ES y ES'. Sea *loops* = `getMaxLoop(e, e')`. Si $e' \in W_{ES'_\bot} \setminus W_{ES_\bot}$ entonces `canBeWinningLoop(loops)`. Además, si `canBeWinningLoop(loops)` entonces $e' \notin L_{ES'_\top}$

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

Para probar que `canBeWinningLoop`(*loops*) implica $e' \notin L_{ES'_\top}$ construimos un supervisor 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 supervisor 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 supervisor que necesitamos. □

Si `canBeWinningLoop()` retorna true, en la linea 16, sabemos que si e' cambia su estado es porque $e' \in W_{ES'_\bot}$. Para ver si este cambio se produce, se realiza una computación estándar de punto fijo. 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 $loops = \texttt{getMaxLoop}(e, e')$ $\wedge$
$\neg \texttt{canBeWinningLoop}(loops)$ y
$P = \texttt{findNewErrorsIn}(loops)$ entonces
$(e' \in L_{ES'_\top} \Rightarrow e' \in P \subseteq L_{ES'_\top}) \wedge (e' \notin L_{ES'_\top} \Rightarrow P = \emptyset)$

Demostración 5: 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 supervisor 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 supervisor 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 supervisor.

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 supervisor 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 supervisor. □

Por motivos de eficiencia, `findNewGoalsIn` y `findNewErrorsIn` no solo verifican si $e' \in W_{ES'_\bot} / e' \in L_{ES'_\top}$ 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 le siguen de forma natural.

Primero, notar que el algoritmo termina cuando logra determinar que $\bar{e}$ está en $L_{ES'_\top}$ o $W_{ES'_\bot}$. En el segundo caso, es simple construir un supervisor basándose en la estructura de exploración ES. □

Teorema 1 (Correctitud y Completitud): Sea $\mathcal{E} = (E, A^C_E)$ un problema de control composicional según Definition 3. Existe una solución para $\mathcal{E}$ si y solo si el algoritmo DCS retorna un supervisor para $\mathcal{E}$.

Demostración 6: El teorema se desprende del invariante de ciclo del algoritmo (Definition 1), el Lemma 1, y el hecho de que en el peor caso todas las transiciones son agregadas a la estructura de exploración. Entonces, $E = ES = ES_\bot = ES_\top$.

4.4. Demostración de Lemas

Proof Lemma 1 **Proof Sketch** To prove $W_{ES_\bot} \subseteq W_{ES'_\bot}$ we show that a supervisor for a state s in $W_{ES_\bot}$ can be used as a supervisor from s in $W_{ES'_\bot}$. For $L_{ES_\top} \subseteq L_{ES'_\top}$, we assume there is a state $s \in L_{ES_\top} \setminus L_{ES'_\top}$. We reach a contradiction by showing that the supervisor that s must have in $ES'_\top$ is also a supervisor for s in $ES_\top$. **End Proof Sketch**

If $s \in W_{ES_\perp}$ then there exists a supervisor σ for the control problem $ES_\perp$. Let Z such that $ES \subseteq Z$. we will show that σ is a supervisor for $Z_\perp$. This requires showing two conditions as per Definition 3. The first, that σ is controllable, is trivial as the set of controllable and uncontrollable events have not changed.

For the second, nonblocking, we first prove that $\mathcal{L}^\sigma(Z_\perp) = \mathcal{L}^\sigma(ES_\perp)$.

Assume that $\mathcal{L}^\sigma(Z_\perp) \not\subseteq \mathcal{L}^\sigma(ES_\perp)$. If $w \in \mathcal{L}^\sigma(Z_\perp) \setminus \mathcal{L}^\sigma(ES_\perp)$. The run that witnesses w must either always be in Z or eventually reach a deadlock state in $Z_\perp$. In either case, let w_0 be the longest prefix in ES. We know w_0 is a proper prefix of w. Let ℓ such that $w_0.\ell$ is a prefix of w. By definition of $ES_\perp$, $w_0.\ell$ reaches a deadlock state in $ES_\perp$. This is a contradiction as σ is a supervisor for $ES_\perp$.

To show that $\mathcal{L}^\sigma(Z_\perp) \supseteq \mathcal{L}^\sigma(ES_\perp)$, assume $w \in \mathcal{L}^\sigma(ES_\perp)$. If w is also in $\mathcal{L}^\sigma(ES)$ it is also in $\mathcal{L}^\sigma(Z)$ and $\mathcal{L}^\sigma(Z_\perp)$. Otherwise, $w = w_0.\ell$ hits a deadlock state in $ES_\perp$. As w_0 in $\mathcal{L}^\sigma(ES)$, it is also in $\mathcal{L}^\sigma(Z)$. Consider the state s reached by w_0 in E, it must have a transition labelled ℓ to justify its addition to $ES_\perp$. In, Z state s either has the transition and thus $w_0.\ell \in \mathcal{L}^\sigma(Z) \subseteq \mathcal{L}^\sigma(Z_\perp)$, or it does not have the transition but then the state in $Z_\perp$) has an ℓ transition to a deadlock state, hence $w_0.\ell \in L^\sigma(Z_\perp)$.

Now, assume a word $w \in \mathcal{L}^\sigma(Z_\perp)$ that cannot be extended with w' such that $w.w'$ is in $L^\sigma(Z_\perp)$ and reaches a marked state of $Z_\perp$. As w is also in $L^\sigma(ES_\perp)$ then, as σ is a supervisor for $ES_\perp$, there is a w' such that $w.w' \in L^\sigma(ES_\perp) = L^\sigma(Z_\perp)$ and hits a marked state. Note that the run for $w.w'$ is always in ES, which means that the run is also in $Z_\perp$. Thus we reach a contradiction.

FALTA BIEN LA PARTE DE $L_{ES_\top}$

$s \in L_{ES_\top} \Rightarrow s$ can't controllably reach any of the transitions that differ between ES and $ES_\top$ since they lead directly to a marked state with a self-loop. If it could, then s would be winning instead of losing. Also, s can be forced by the environment to losing states within ES, and this will still be true for any $Z \mid ES \subseteq Z \subseteq E$. Then, since this works for any $ES \subseteq Z \subseteq E$, in particular it is valid $Z = E$. Finally, if $s \in L_E$ then s is loser in $E_\top$, and $E_\top = E$ because there are no transitions in $E \setminus E$, so s is loser in E.

□

Proof Lemma 2 **Begin Proof Sketch**

We first show there is a path from s to e' by assuming there is none. If $s \notin W_{ES_\perp} \cup L_{ES_\top}$ then it has a supervisor σ in $ES_\top$ but it does not have one in $ES_\perp$. This depends entirely on the descendants of s, since those are the only states that supervisors can reach in ES. If s is not a predecessor of e', as $e \xrightarrow{l}_{ES'} e'$ is the only difference between ES and ES', then the descendants of s are the same, thus its possible supervisors in $ES'_\top$ and $ES'_\perp$ are unchanged. Thus, $s \notin W_{ES'_\perp} \cup L_{ES'_\top}$ which is a contradiction.

We then prove that there is at least one path from s to e' via *None* states by contradiction assuming that all paths to e' in ES' cross a state $s' \in (W_{ES_\perp} \cup L_{ES_\top})$. A supervisor σ from s in $ES_\top$ will not reach states in $L_{ES_\top}$, thus for all s' it crosses they will have a supervisors $\sigma_{s'}$ for $ES_\perp$. We use σ and $\sigma_{s'}$ to build a supervisor for s in $ES'_\top$ to show that $s \notin L_{ES'_\top}$. A supervisor for s in $ES'_\perp$ cannot exist because otherwise we can use it to build a supervisor for s in $ES_\perp$ using a similar reasoning as before. This means that $s \in W_{ES_\perp}$ which contradicts the hypothesis.

End Proof Sketch

If a state s is not in $W_{ES_\perp} \cup L_{ES_\top}$ it is because it has a supervisor σ in $ES_\top$ but it doesn't have one in $ES_\perp$. This depends entirely on the descendants of s, since those are

the only states that σ can reach. If s is not a predecessor of e', and $e \xrightarrow{l}_{ES'} e'$ is the only difference between ES and ES' then the descendants of s are the same, thus its possible supervisors haven't changed, and s is still NONE.

What is not so clear, is that s has no new possible supervisors if it has a path to reach e' but only passing through states in $W_{ES_\bot} \cup L_{ES_\top}$. Assuming it must pass through states in $W_{ES_\bot} \cup L_{ES_\top}$ we show that:

- knowing s had a supervisor σ in $ES_\top$, we show s has a valid supervisor σ' in $ES'_\top$:

  $\sigma'(w) = \sigma(w)$ if there is no w_0 suffix of w such that $s \xrightarrow{w_0}_{ES'} s_i \wedge s_i \in L_{ES_\top} \cup W_{ES_\bot}$.

  $\sigma'(w) = \sigma_{s_i}(w_1)$ where w_0 is the shortest suffix of $w = w_0.w_1$ such that $s \xrightarrow{w_0}_{ES'} s_i \wedge s_i \in W_{ES_\bot}$. σ_{s_i} is the supervisor we know s_i had in $ES_\bot$ because $s_i \in W_{ES_\bot}$, and every supervisor valid in $ES_\bot$ is also valid in $ES_\top$.

  Since σ is a valid supervisor, we know it will not reach states in $L_{ES_\top}$.

  Finally it is clear that σ' is a valid supervisor for s in $ES'_\top$. Note that σ' does not depend of the new transition.

- Knowing s had no supervisor in $ES_\bot$, we show that s has no supervisor in $ES'_\bot$ by assuming there is one and reaching a contradiction:

  Suppose there is a supervisor σ' for s in $ES'_\bot$, and that $e \xrightarrow{l}_{ES'} e'$ is the only difference between ES and ES'.

  With σ' we will build σ, a supervisor for s in $ES_\bot$.

  $\sigma(w) = \sigma'(w)$ if there is no w_0 which is a prefix of w and $s \xrightarrow{w_0}_{ES} s_i \wedge s_i \in W_{ES_\bot} \cup L_{ES_\top}$. Since σ' is a valid supervisor, we know it will not reach states in $L_{ES'_\top}$. Note that w can't reach $e \xrightarrow{\ell}_{ES'} e'$ because s has no path of *None* states to e'.

  If $w = w_0.w_1$ and $s \xrightarrow{w_0}_{ES} s' \wedge s' \in W_{ES_\bot}$ then $\sigma(w_0.w_1) = \sigma_{s'}(w_1)$ where $\sigma_{s'}$ is the supervisor for s' in $ES_\bot$. Note that once we reach s' we always follow $\sigma_{s'}$.

  Since σ never reaches the new transition we know that σ is valid in $ES_\bot$.

  We see then that assuming there is a valid supervisor σ' for s in $ES'_\bot$ implies that there is a valid supervisor σ for s in $ES_\bot$. ABS!

Proof Lemma 3 **Begin Sketch** Assuming $e' \notin W_{ES'_\bot}$, we use a witness s to $W_{ES'_\bot} \neq W_{ES_\bot}$ to reach a contradiction. State s must have a supervisor in $W_{ES'_\bot}$ that avoids $e \xrightarrow{\ell} e'$ which is the only difference between $ES_\bot$ and $ES'_\bot$. This supervisor is then also a supervisor for s in $W_{ES_\bot}$ reaching a contradiction.

If we assume $e' \notin L_{ES'_\top}$. We use a witness s to $L_{ES'_\top} \neq L_{ES_\top}$ to reach a contradiction. Note that as $e' \notin L_{ES'_\top}$, there is a supervisor σ from e' in $ES'_\top$. As $s \notin L_{ES_\top}$ it also must have a supervisor σ' in $ES_\top$. We build a new supervisor for $ES'_\top$ from s that works just like σ' but when it reaches $e \xrightarrow{\ell} e'$ it behaves as σ. This new supervisor proves that $s \in L_{ES'_\top}$ which is a contradiction. **End Sketch**

We prove both implications by contradiction.

First we assume $e' \notin W_{ES'_\bot} \setminus W_{ES_\bot}$. Note that as $e' \notin W_{ES_\bot}$ then $e' \notin W_{ES'_\bot}$. As $W_{ES'_\bot} \neq W_{ES_\bot}$ by monotonicity (Lemma1) there must be a state s such that $s \in W_{ES'_\bot} \setminus$

$W_{ES_\perp}$, so s must have a supervisor in $W_{ES'_\perp}$. This supervisor cannot reach e' because if it did, then there would be a supervisor for e' and we had assumed $e' \notin W_{ES'_\perp}$. Furthermore, if the supervisor reaches e, then ℓ must be controllable (it if were uncontrollable, the supervisor would reach e' which we established is not possible). Thus, the supervisor avoids $e \xrightarrow{\ell} e'$ altogether which means that it is also a supervisor for s in $ES_\perp$ (i.e., $s \in W_{ES_\perp}$) reaching a contradiction.

Now we assume $e' \notin L_{ES'_\top} \setminus L_{ES_\top}$. Note that as $e' \notin L_{ES_\top}$ then $e' \notin L_{ES'_\top}$. Let $s \in L_{ES'_\top}$ and $s \notin L_{ES_\top}$. We know that from s there is a supervisor σ' for $ES_\top$. This supervisor either is also a supervisor for ES or reaches the $\top$ state in $ES_\top$. In the first case, it is also a supervisor ES' and in $ES'_\top$, reaching a contradiction. In the second case, it either uses a transition that is neither in ES' nor ES, which means that in $ES'_\top$ it will lead to a winning $\top$ state; or it uses the transition that is in ES' but not ES which leads to e'. Since we know e' is not in $L_{ES'_\top}$, then it has a supervisor for it in $ES'_\top$, then we know that there exists a supervisor σ'' that includes both σ' and the supervisor for e'. Finally, σ'' is a supervisor for s in $ES'_\top$, but $s \notin L_{ES'_\top}$, ABS!

Proof Lemma 4 **Begin Proof Sketch** To prove $e' \in W_{ES'_\perp} \setminus W_{ES_\perp}$ implies `canBeWinningLoop(`*loops*`)`, we assume $\neg$`canBeWinningLoop(`*loops*`)` and show that $e' \notin W_{ES'_\perp} \setminus W_{ES_\perp}$. For this, it suffices to see that if $\neg$`canBeWinningLoop(`*loops*`)` then to reach a marked state from e' it must exit *loops* to a state $s \notin$ *loops*. This state must be such that $s \notin W_{ES'_\perp}$. As s has no supervisor in $ES'_\perp$, it is impossible for e' to have one.

To prove that `canBeWinningLoop(`*loops*`)` implies $e' \notin L_{ES'_\top}$ we build a supervisor σ' for e' in $ES'_\top$ as follows: For a trace that stays within *loops*, we only choose controllable successors that are not in $L_{ES_\top}$. Note that there cannot be uncontrollable successors in $L_{ES_\top}$ because $loops \cap L_{ES_\top} = \emptyset$. As soon as a trace exits *loops* at a state s' we use the supervisor for s' in $ES'_\top$. As s' cannot reach $e \xrightarrow{\ell}_{ES'} e'$ using *None* states, by Lemma 2, s' must have such a supervisor.

End Proof Sketch

Supose $e' \in W_{ES'_\perp} \setminus W_{ES_\perp}$ but $\neg$`canBeWinningLoop(`*loops*`)`. There exists a supervisor σ for e' in $ES'_\perp$, this means there must be a path w from e' to a marked state m. Since $\neg$`canBeWinningLoop(`*loops*`)`, there are no marked state in *loops*, w must leave *loops*. Let s be the first state reached by w not in *loops*, s is in $L_{ES_\top} \cup None$, and doesn't have a *None* path to e' then by Lemma 2 s will still not change it status. Since $s \notin W_{ES'_\perp}$, s doesn't have a supervisor in $ES'_\perp$, but σ accepts runs that lead to s, absurd.

Assuming `canBeWinningLoop(`*loops*`)` we simply build a supervisor σ' (MISMO SUPERVISOR QUE EN PROXIMA DEMO) for e' in $ES'_\top$ to prove that $e' \notin L_{ES'_\top}$

We define σ' so that it enables all uncontrollable transitions (to be *controllable*).

$\sigma'(w_0.w_1) = \sigma_{s'}(w_1)$ if w_0 is the shortest word such that there is an $s' \notin loops \wedge s' \notin L_{ES_\top}$ and $e' \xrightarrow{w_0}_{ES'_\top} s'$. Else if $e' \xrightarrow{w}_{ES'_\top} p \wedge p \in loops$ then for all controllable ℓ', $\ell' \in \sigma'(w)$ iff $\exists p' \,.\, p \xrightarrow{l'} p' \wedge p' \notin L_{ES_\top}$.

We use the supervisors $\sigma_{s'}$ where s' is such that exists $s \in loops$ and $s \xrightarrow{\ell'}_{ES'_\top} s' \wedge s' \notin loops \wedge s' \notin L_{ES_\top}$. If none such s' exists, we know there is a marked state in *loops* and σ' never leaves *loops* since all reachable states from *loops* are in $L_{ES_\top}$.

We will prove that σ' is *controllable* and $non-blocking$.

Since we enabled all uncontrollable transitions, σ' is trivially *controllable*.

For $non-blocking$, assume w compatible with σ', we will show that it can be extended. If $w = w_0.w_1$ where w_0 is the shortest word such that there is an $s' \notin loops \wedge s' \notin L_{ES_\top}$ and $e' \xrightarrow{w_0}_{ES'_\top} s'$. Then by definition of σ' we have that $\sigma'(w_0.w_1.w_2) = \sigma_{s'}(w_1.w_2)$ for all w_2. As $\sigma_{s'}$ is $non-blocking$, there is a w_2 such that $\sigma_{s'}(w_1.w_2)$ hits a marked state. So $\sigma'(w_0.w_1)$ can be extended to hit the marked state.

Otherwise, w never exits $loops$. We want to prove that for every ℓ' such that $w.\ell'$ is consistent with σ', $w.\ell'$ is extendable with a w' to reach a marked state. Let p' be such that $e' \xrightarrow{w.\ell'} p'$. If $p' \notin loops$ then $\sigma'(w.\ell') = \sigma_{p'}(\lambda)$ and, as before, we know $\sigma_{p'}$ is $non-blocking$ so $w.l'$ is extendable to reach a marked state.
If $p' \in loops$, then $w.l'$ can be extended to reach any state s in the $loops$. We know that either there is a marked state in $loops$ or a state in $W_{ES_\perp}$ reachable in one step from $loops$, either way we can extend $w.l'$ to reach a marked state.

Proof Lemma 5 **Begin Proof Sketch** We split the proof based on the if/then/else structure of `findSomeNewErrors`. In the case of `if` being true, it suffices to prove that $e' \notin L_{ES'_\top}$. For this, we build a supervisor σ' for e' in $ES'_\top$ as follows: For a trace that stays within $loops$, we only choose controllable successors that are not in $L_{ES_\top}$. Note that there cannot be uncontrollable successors in $L_{ES_\top}$ because $loops \cap L_{ES_\top} = \emptyset$. As soon as a trace exits $loops$ at a state s' we use the supervisor for s' in $ES'_\top$. As s' cannot reach $e \xrightarrow{\ell}_{ES'} e'$ using $None$ states, by Lemma 2, s' must have such a supervisor.

When the `if` is false, it suffices to prove $P = loops \subseteq L_{ES'_\top}$. We do this by assuming $s \in loops \setminus L_{ES'_\top}$ and reaching a contradiction. If $s \notin LESS$ it has a supervisor σ which admits a trace to a marked state. As there are no marked states in $loops$, the trace reaches a state $s' \not\in loops$ As the `if` was false, $s' \in L_{ES'_\top}$ which means σ is not a supervisor. **End Proof Sketch**

First, we know that every state $s' \notin loops$ such that $\exists s \in loops \,.\, s \xrightarrow{l} s'$, either is and will be a losing state ($s' \in LES \wedge s' \in LESS$) or is and will be $None$ (because s isn't a $NONE-$Predecessor of any state in $loops$, otherwise s would be in $loops$).
This means that any state $s' \notin loops \wedge s' \notin L_{ES_\top}$ can not be forced to a state in $L_{ES'_\top}$. Thus, if we reach a $None$ state we know it has a supervisor $\sigma_{s'}$ in $ES'_\top$.

In the case that the `if` statement is true, we prove that $e' \notin L_{ES'_\top}$:

We will build a supervisor for e' in $ES'_\top$ using the supervisors $\sigma_{s'}$ where s' is such that exists $s \in loops$ and $s \xrightarrow{\ell'}_{ES'_\top} s' \wedge s' \notin loops \wedge s' \notin L_{ES_\top}$.

We define σ' so that it enables all uncontrollable transitions (to be $controllable$).

$\sigma'(w_0.w_1) = \sigma_{s'}(w_1)$ if w_0 is the shortest word such that there is an $s' \notin loops \wedge s' \notin L_{ES_\top}$ and $e' \xrightarrow{w_0}_{ES'_\top} s'$.

Else if $e' \xrightarrow{w}_{ES'_\top} p \wedge p \in loops$ then for all controllable ℓ', $\ell' \in \sigma'(w)$ iff $\exists p' \,.\, p \xrightarrow{l'} p' \wedge p' \notin L_{ES_\top}$.

We will prove that σ' is $controllable$ and $non-blocking$.

Since we enabled all uncontrollable transitions, σ' is trivially $controllable$.

For $non-blocking$, assume w compatible with σ', we will show that it can be extended. If $w = w_0.w_1$ where w_0 is the shortest word such that there is an $s' \notin loops \wedge s' \notin L_{ES_\top}$ and $e' \xrightarrow{w_0}_{ES'_\top} s'$. Then by definition of σ' we have that $\sigma'(w_0.w_1.w_2) = \sigma_{s'}(w_1.w_2)$ for all w_2. As $\sigma_{s'}$ is $non-blocking$, there is a w_2 such that $\sigma_{s'}(w_1.w_2)$ hits a marked state. So $\sigma'(w_0.w_1)$ can be extended to hit the marked state.

Otherwise, w never exits *loops*. We want to prove that for every ℓ' such that $w.\ell'$ is consistent with σ', $w.\ell'$ is extendable with a w' to reach a marked state. Let p' be such that $e' \xrightarrow{w.\ell'} p'$. If $p' \notin loops$ then $\sigma'(w.\ell') = \sigma_{p'}(\lambda)$ and, as before, we know $\sigma_{p'}$ is $non-blocking$ so $w.l'$ is extendable to reach a marked state.
If $p' \in loops$, then $w.l'$ can be extended to reach any state s in the *loops*. The if statement guarantees that there exists an s such that $s \in loops\ .\ s \xrightarrow{\ell'}_{ES'_\top} s' \wedge (s' \notin loops \wedge s' \notin Errors)$, thus we can extend $w.l'$ to reach that s' and use, as before, $\sigma_{s'}$, to prove that $w.l'$ can be extended to hit a marked state.

Otherwise we enter the `else` block:

If $\nexists s \in loops\ .\ s \xrightarrow{\ell'}_{ES'_\top} s' \wedge (s' \notin loops \wedge s' \notin Errors)$ we prove that $\forall s \in loops$, $s \in L_{ES'_\top}$

Suppose there is a supervisor σ' for s in $ES'_\top$ then $\exists w'$ such that $s \xrightarrow{\lambda.w'}_{ES'_\top} e_m \wedge e_m \in M_{ES'_\top}$. Since there's no marked state in *loops* then eventually s following w' leaves *loops*. Let $w' = w_0.w_1$ such that w_0 is the shortest word so that $s \xrightarrow{w_0}_{ES'_\top} s' \wedge s' \notin loops$. But since $s' \in Errors \Rightarrow s' \in L_{ES_\top}$ then it is not possible for a valid supervisor σ' to allow reaching that state. ABS! Then there's no supervisor for s in $ES'_\top$ implying $\forall s \in loops$, $s \in L_{ES'_\top}$.

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 programa utilizado viene 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 la figura (agregar ref) podemos ver un ejemplo donde declaramos un LTS llamado $MiLTS$, el mismo tiene como estado inicial $A0$. Este estado puede ir por $c01$ a $A1$, $A1$ se declara en la linea siguiente. Tenemos la información entonces de cada estado y sus transiciones. Además se pueden componer las distintas LTSs con el comando `||`.

```
MiLTS = A0,
A0 = (c01 -> A1),
A1 = (u12 -> A2 | u13 -> A3),
A2 = (u23 -> A3),
A3 = (u33 -> A3).

OtroLTS = B0,
B0 = (c01 -> A1),
B1 = (u12 -> A2 | u13 -> A3),
B2 = (u23 -> A3 | inicial -> A1).

|| Compuesto = (MiLTS || OtroLTS).
```

Listing 5.1: Ejemplo de LTS y composición

QUIZAS CAMBIAR EL EJEMPLO POR UNO MENOS HORRIBLE

AGREGAR DEFINICION DEL CONTROLLER, QUÉ COSAS SE LE PUEDEN PASAR Y CÓMO LLAMAR A DCS.

5.2. Testing

Luego de haber presentado la sintaxis con la cuál desarrollamos nuestros tests vamos a hablar un poco de ellos y qué es lo que se esperaba en cada uno. AGREGAR SOBRETODO LOS TESTS QUE NOS HICIERON DAR CUENTA QUE EL PROPAGATE NO PODIA SER LOCAL Y HABIA QUE MARCAR ERRORES AL TERMINAR DE EXPLORAR ALGO

```
Ejemplo = A0,
A0 = (c01 -> A1),
A1 = (u12 -> A2 | u13 -> A3),
A2 = (u23 -> A3),
A3 = (u33 -> A3).

||Plant = Ejemplo.

controllerSpec Goal = {
controllable = {c01}
marking = {u33}
nonblocking
```

```
}

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

Listing 5.2: Ejemplo de test

6. PERFORMANCE

Para realizar las pruebas de performance decidimos utilizar el mismo conjunto de problemas creado y utilizado en [2] para examinar el algoritmo original de DCS, ya que nuestra intención era principalmente compararnos contra la versión anterior. En esta sección presentamos los resultados de la comparación versus dicha versión y, además, contra diversos programas del estado del arte de resolución de problemas de síntesis.

Todos los casos de estudios fueron escritos de manera de poder modificar el número de componentes y estados, con la intención de probar escalabilidad dentro de cada tipo de problema.

Transfer Line Automatización de una fabrica, 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 cumpplir 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 adjacente 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 concenso, 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 concenso. 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 osquestar los servicios de manera de obtener un paquete de vacaciones completo de ser posible, evitando pagar por paquetes incompletos.

6.1. Comparación con versión previa de DCS

Como ya dijimos el principal foco del trabajo fue de brindar una mayor seguridad sobre la correctitud y completitud del approach novedoso de la exploración on the fly. Esto debía hacerse sin perder la buena performance que aportaba la técnica, con el foco de poder 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 dos heurísticas distintas, *MonotonicAbstraction* y *ReadyAbstraction*, desarrolladas en [2], para observar la diferencia de performance en distintas condiciones. Puede notarse que la segunda heurística supera ampliamente a la primera, pero que para ambas las diferencias entre los algoritmos de exploración es mínima.

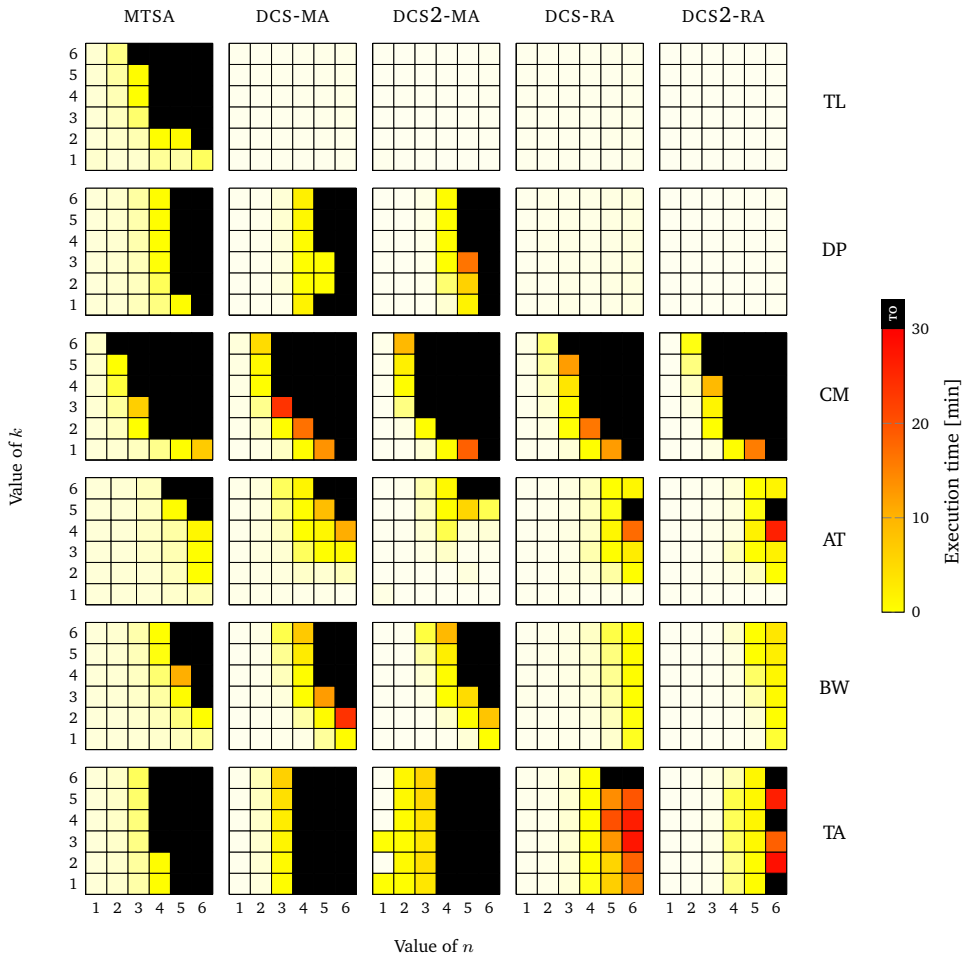


Fig. 6.1: dcs2 = performance nuevo algoritmo

En la figura 6.1 puede verse la comparación y que en la mayoría de los problemas los resultados son similares. En los problemas CM y TA, la nueva versión del algoritmo $DCS2$ no logra resolver algunas instancias antes del *timeout* que previamente podían ser resueltas. No consideramos eso como una señal mayor de mala performance en el algoritmo, ya que hay otros problemas como AT y BW donde los resultados para la heurística $MonotonicAbstraction$ no solo no empeoraron sino que mejoraron.

Es esperable que en un proceso exploratorio guiado de forma heurística, cambios en la poda de ramas (por la clasificación o falta de la misma de ciertos estados) van a llevar a explorar caminos más o menos fructíferos según la estructura del problema. Lo relevante de la comparación es que no se observa una diferencia generalizada en los desempeños de las distintas versiones del algoritmo de exploración.

De estos resultados concluímos que las modificaciones al algoritmo no afectan de forma significativa su performance.

6.2. Comparación con otros programas

En función de la completitud del capítulo decidimos agregar los gráficos de comparación entre DCS2 y las herramientas utilizadas en [2]. Al igual que su versión anterior DCS2 supera ampliamente en muchas de las instancias a las demás herramientas del estado del arte, esto se puede apreciar en la figura 6.2. Además la figura 6.3 muestra un resúmen visual de total de instancias resultas y tiempo de ejecución, respectivamente.

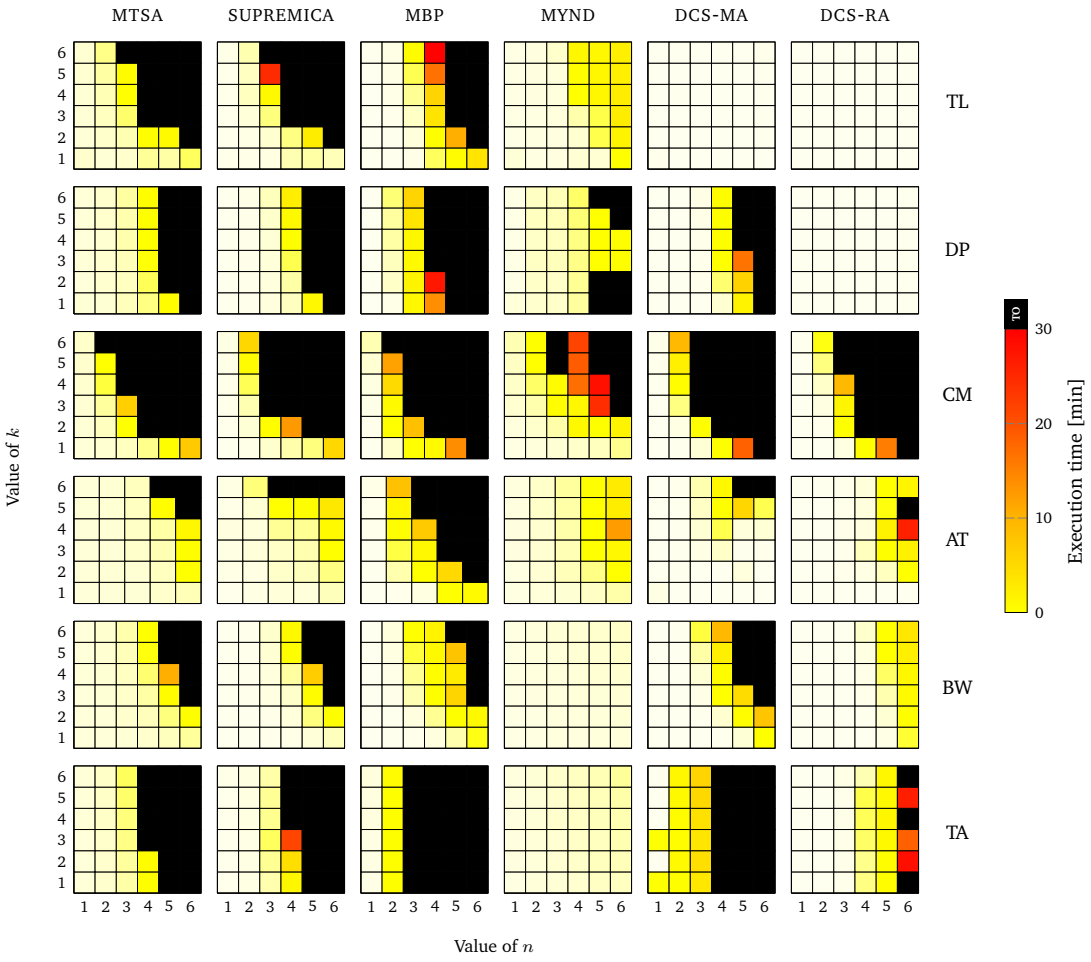


Fig. 6.2: comparación detallada

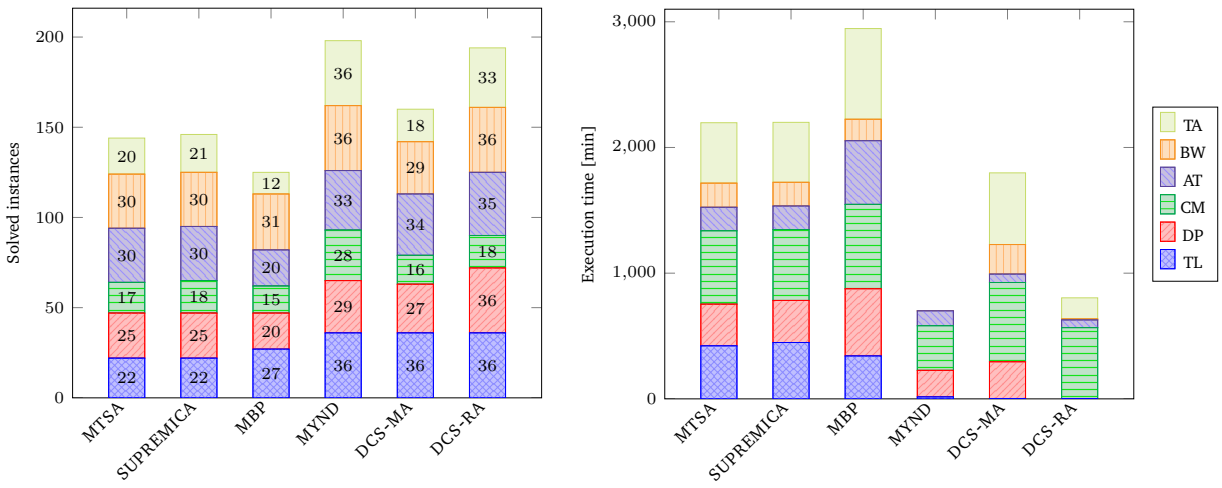


Fig. 6.3: Cantidad total de instancias resueltas (izquierda) y tiempo de ejecución (derecha) con DCS y otras herramientas.

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. Éste 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.