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

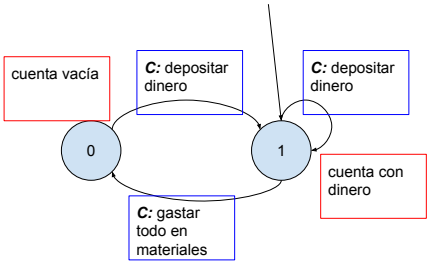
A continuación se presenta un ejemplo para comprender el problema a resolver. Se trata de una versión simplificada del problema *TravelAgency* utilizado para medir la performance del algoritmo.

Se desea armar un sevicio de venta online de paquetes vacacionales que reservará de forma automática una variedad de servicios (alquiler de auto, hotel, pasaje de avion, etc.) asegurando que no se perderá nada de dinero a menos que se reserve el paquete completo.

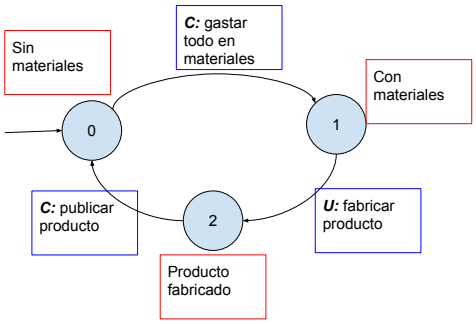
Para cada servicio que se desea sub-contratar presentamos una versión simplificada en la cual se consulta si ese servicio está disponible. En caso de estar disponible queda reservado hasta la compra definitiva del paquete completo y la cancelación de la reserva si otro servicio no estaba disponible no 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 6).

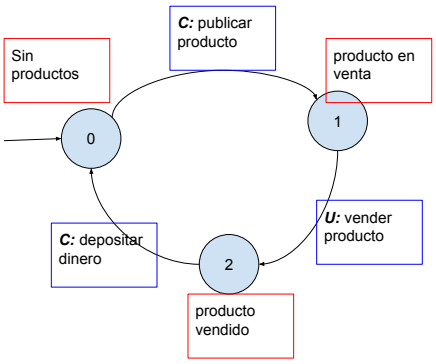
Mostramos en la figura 1.1 un LTS (Labeled transition system) para cada uno de los componentes descriptos y el LTS compuesto para el caso en el que se sub-contrata un solo servicio.



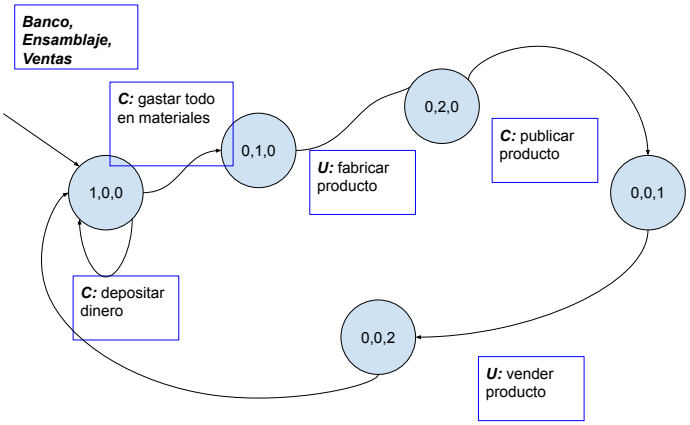
(a) Cuenta bancaria



(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. Heurística de debugging

Como dijimos en el capítulo anterior, el algoritmo de DCS debe ser agnóstico a la heurística. Al comenzar nuestro trabajo en el proyecto y una vez que pudimos generar un conocimiento sobre el pseudocódigo nos percatamos de ciertos casos borde que no iban a ser bien resueltos, o esto suponíamos. Sin embargo, al correr dichos casos el resultado era correcto, esto se debía a que la heurística era muy buena y llevaba al error directamente; 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 no mira ninguna información sobre distancia a marcados o error. Decidimos dejar el ordenamiento de no controlables primero ya que esto no es heurístico, se sabe perfectamente 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. En la figura AGREGAR REF vemos un gráfico a modo de resúmen sobre cantidad de tests fallados con/sin heurística *Dummy* y en cuántos dan diferentes resultados. Nuestro suite de regresión cuenta con 50 tests, todos de casos sumamente especiales o variaciones pequeñas de los mismos que son interesantes desde un punto de vista implementativo.

3.2. Suite de regresión

A continuación mostraremos algunos tests dignos de mención y explicaremos qué problemática ataca. En el capítulo 5 se puede encontrar más información sobre ellos, junto con el código del modelo.

3.3. Puntos a resolver

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.

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 erroneo. Para comprender mejor observar la figura (INGRESE FIGURA) donde desde el estado e tenemos dos subramas 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. Hay que tener en cuenta que esto podía pasar debido a que no se

requería que un estado hijo tenga conclusión para seguir propagando, es decir, bastaba con que haya sido explorado alguna vez y se asumía lo mejor.

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 no controlable entre dos estados y uno de ellos va no controlablemente a un error. En este caso es obvio que ambos deben ser errores pero según la mirada local tienen *una forma de escapar del error*, el otro estado del conjunto. Para una aclaración visual ver la fugura INGRESE OTRA FIGURA.

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 erroneas y problemas de propagación había ciertos casos donde según el algoritmo de exploración el problema era controlable; pero que al llegar al constructor del controlador se daba cuenta que había estados a los cuáles 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.

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.

Expandir esta sección

Poner que nunca queremos que el open quede vacío, siempre queremos concluir que el inicial es W o L.

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       if canBeWinningLoop(loops):
17         C = findNewGoalsIn(loops)
18         Goals = Goals ∪ C
19         None = None \ C
20         propagateGoal(C)
21       else:
22         P = findNewErrorsIn(loops)
23         Errors = Errors ∪ loops
24         None = None \ loops
25         propagateError(P)
26     ES = ES'
27
28   if ē ∈ Goals:
29     return (λw . { ℓ | ē -w->_ES e -ℓ->_ES e' ∧ e' ∈ Goals })
30   else:
31     return UNREALIZABLE
```

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

```
function propagateGoal(newGoals):
  C = ancestorsNone(newGoals)
  C' = ∅
  while C_Viejo ≠ 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 . forcedTo(s, e, ES'_⊤) ∧ e ∈ Errors) ∨
        cannotReachGoalIn(s, C)}
  P = P \ C
  Errors = Errors ∪ P
  None = None \ P
```

Listing 4.2: Status propagation procedures.

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

function findNewErrorsIn(loops):
  if (∃s ∈ loops . s -ℓ'→_{ES'⊤} s' ∧ (s' ∉ loops ∧ s' ∈ None)):
    P = ∅
  else:
    P = loops
  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 $L_{ES_\top} \subseteq L_{ES'_\top}$.

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: $ES \subseteq E \wedge \forall s \in ES \,.\, (s \in Goals \Leftrightarrow s \in W_{ES_\bot}) \wedge (s \in Errors \Leftrightarrow s \in L_{ES_\top}) \wedge s \in Errors \uplus Goals \uplus None$

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

Ya en la linea 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?

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 16 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 $e' \notin L_{ES'_\top}$

Si `canBeWinningLoop()` retorna true, en la linea 17, 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
$(e' \in L_{ES'_\top} \Rightarrow e' \in P \subseteq L_{ES'_\top}) \wedge (e' \notin L_{ES'_\top} \Rightarrow P = \emptyset)$

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 $\mathcal{E} = (E, A_E^C)$ 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 (Correctitud y completitud): 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

Demostración Lemma 1: (Idea: 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 $ES_\top$.)

Si $s \in W_{ES_\bot}$ entonces existe un supervisor σ para el problema de control $ES_\bot$. Sea Z tal que $ES \subseteq Z$. Demostraremos que σ es un supervisor para $Z_\bot$. 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 $\mathcal{L}^\sigma(Z_\bot) = \mathcal{L}^\sigma(ES_\bot)$.

Si asumimos que $\mathcal{L}^\sigma(Z_\bot) \not\subseteq \mathcal{L}^\sigma(ES_\bot)$ y $w \in \mathcal{L}^\sigma(Z_\bot) \setminus \mathcal{L}^\sigma(ES_\bot)$, la corrida que verifica w debe permanecer siempre en Z o alcanzar eventualmente un estado *deadlock* en $Z_\bot$. 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 supervisor 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 $\mathcal{L}^\sigma(ES)$, pertenecer también a $\mathcal{L}^\sigma(Z)$. Consideramos el estado s alcanzado por w_0 en E, debe tener una transición etiquetada como ℓ para justificar su inclusión en $ES_\perp$. En Z, el estado s o tiene la transición y por lo tanto $w_0.\ell \in \mathcal{L}^\sigma(Z) \subseteq \mathcal{L}^\sigma(Z_\perp)$, o no tiene la transición, pero el estado en $Z_\perp$) tiene una transición ℓ a un estado *deadlock*, por lo tanto $w_0.\ell \in L^\sigma(Z_\perp)$.

Ahora, sabiendo que $\mathcal{L}^\sigma(Z_\perp) = \mathcal{L}^\sigma(ES_\perp)$, procedemos a *nonblocking*. Sea una palabra $w \in \mathcal{L}^\sigma(Z_\perp)$ que no puede ser extendida con w' tal que $w.w'$ se encuentra en $L^\sigma(Z_\perp)$ y alcanza un estado marcado de $Z_\perp$. Como w también se encuentra en $L^\sigma(ES_\perp)$ entonces, como σ es un supervisor 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 supervisor σ en $ES'_\top$, pero $s \in L_{ES_\top}$ por lo que no puede existitr un supervisor válido σ' para s en $ES_\top$. Esto es falso, más aún, mostraremos que si σ es un supervisor para s en $ES'_\top$, entonces hay un supervisor válido σ' para s en cualquier $Z_\top$ si $Z \subseteq ES$.

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

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

□

Demostración Lemma 2: (Idea: Si s no es un predecesor de e', como $e \xrightarrow{l}_{ES'} e'$ es la única diferencia entre ES y ES', entonces los decendientes de s son los mismos, por lo tanto sus posibles supervisores 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 $s' \in (W_{ES_\perp} \cup L_{ES_\top})$. 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 $ES_\perp$. 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 $ES'_\perp$ no puede existir porque de otra forma podríamos usarlo para construir un supervisor 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 supervisor 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 decendientes de s son los mismos, por lo que los posibles supervisores no tuvieron ningún cambio, y s sigue siendo NONE.

Lo que no es tan claro, es que s no tiene nuevos supervisores 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 supervisor σ en $ES_\top$, mostramos que s tiene un supervisor 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 supervisor que sabemos que s_i tiene en $ES_\perp$ ya que $s_i \in W_{ES_\perp}$, y que cada supervisor válido en $ES_\perp$ es también válido en $ES_\top$.

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

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

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

  Suponemos que existe un supervisor 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 supervisor 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 supervisor 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 supervisor 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 supervisor válido σ' para s en $ES'_\perp$ estamos implicando la existencia de un supervisor 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 supervisor en $W_{ES'_\perp}$ que evita $e \xrightarrow{\ell} e'$, la única diferencia entre $ES_\perp$ y $ES'_\perp$. Este supervisor entonces es también un supervisor para s en $W_{ES_\perp}$ 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.)

Probamos ambas implicaciones por contradicción.

Primero asumimos que $e' \notin W_{ES'_\bot} \setminus W_{ES_\bot}$. Notar que como $e' \notin W_{ES_\bot}$ entonces $e' \notin W_{ES'_\bot}$. Como $W_{ES'_\bot} \neq W_{ES_\bot}$ y por la monotonicidad del (Lemma1) debe existir un estado s tal que $s \in W_{ES'_\bot} \setminus W_{ES_\bot}$, entonces s debe tener un supervisor en $W_{ES'_\bot}$. Este supervisor no puede alcanzar e' porque si lo hiciera, debería haber un supervisor para e' y comenzamos asumiendo que $e' \notin W_{ES'_\bot}$. Más aún, si el supervisor alcanzara e, entonces ℓ debe ser controlable (si fuera no controlable, el supervisor alcanzaría e' lo cual ya establecimos no es posible). Entonces, el supervisor evita $e \xrightarrow{\ell} e'$ lo que significa que debe ser también un supervisor para s en $ES_\bot$ (i.e., $s \in W_{ES_\bot}$) 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 supervisor σ' para $ES_\top$. Este supervisor puede o ser también un supervisor para ES o alcanzar el estado $\top$ en $ES_\top$. En el primer caso, es también un supervisor 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 supervisor en $ES'_\top$, entonces sabemos que existe un supervisor σ'' que incluye tanto a σ' como al supervisor para e'. Finalmente, σ'' es un supervisor para s en $ES'_\top$, pero $s \notin L_{ES'_\top}$, ABS!

□

Demostración Lemma 4: (Idea: Para probar que $e' \in W_{ES'_\bot} \setminus W_{ES_\bot}$ implica `canBeWinningLoop(`*loops*`)`, asumimos
$\neg$`canBeWinningLoop(`*loops*`)` y mostramos que $e' \notin W_{ES'_\bot} \setminus W_{ES_\bot}$. Para esto, basta con ver que si $\neg$`canBeWinningLoop(`*loops*`)` entonces para alcanzar un estado marcado desde e' se debe salir de *loops* a un estado $s \notin loops \cup W_{ES_\bot}$ 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.)

Supose $e' \in W_{ES'_\bot} \setminus W_{ES_\bot}$ but $\neg$`canBeWinningLoop(`*loops*`)`. There exists a supervisor σ for e' in $ES'_\bot$, 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 $s \notin W_{ES'_\bot}$, s doesn't have a supervisor in $ES'_\bot$, but σ accepts runs that lead to s, absurd.

Assuming `canBeWinningLoop(`*loops*`)` we simply build a supervisor σ^4 for e' in $ES'_\top$ to prove that $e' \notin L_{ES'_\top}$

We define σ^4 so that it enables all uncontrollable transitions (to be *controllable*).

$\sigma^4(w_0.w_1) = \sigma_{s'}(w_1)$ if w_0 is the shortest word such that there is an $s' \notin loops \wedge s' \notin L_{ES_\top}$ and $e' \overset{w_0}{\dashrightarrow}_{ES'_\top} s'$. Else if $e' \overset{w}{\dashrightarrow}_{ES'_\top} p \wedge p \in loops$ then for all controllable ℓ', $\ell' \in \sigma^4(w)$ 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^4 never leaves *loops* since all reachable states from *loops* are in $L_{ES_\top}$.

We will prove that σ^4 is *controllable* and $non-blocking$.

Since we enabled all uncontrollable transitions, σ^4 is trivially *controllable*.

For $non-blocking$, assume w compatible with σ^4, 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 $e' \overset{w_0}{\dashrightarrow}_{ES'_\top} s'$. Then by definition of σ^4 we have that $\sigma^4(w_0.w_1.w_2) = \sigma_{s'}(w_1.w_2)$ 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 $\sigma^4(w_0.w_1)$ 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^4, $w.\ell'$ is extendable with a w' to reach a marked state. Let p' be such that $e' \overset{w.\ell'}{\dashrightarrow} p'$. If $p' \notin loops$ then $\sigma^4(w.\ell') = \sigma_{p'}(\lambda)$ 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.

□

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 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.)

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 use the same σ^4 as in Proof Lemma 4. We already know σ^4 is both *controllable* and $non-blocking$ in this situation because of the previos Lemma.

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 software utilizado cuenta con una gran cantidad de funcionalidad. Principalmente nos interesa la forma de escribir Labelled Transition Systems (LTS), esto se puede hacer mediante Finite State Process (FSP). En el listing 5.1 podemos ver un ejemplo donde se declara 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 LOS EJEMPLOS POR UNOS MENOS HORRIBLES

En el listing 5.2 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 $u12$ como marcada, por ende el estado $A2$ estará marcado. Luego definimos el conjunto de transiciones controlables, en este caso solo $c01$. Luego debemos agregar la palabra clave `nonblocking`, en caso contrario se intentará, por defecto, resolver *blocking*.

En la última línea utilizamos la palabra clave `heuristic` para aclarar que queremos utilizar el algoritmo de DCS, luego le damos el nombre de *DirectedController* y lo definimos como el LTS *Compuesto* dirigido bajo el controller *Goal*.

```
...
controllerSpec Goal = {
    marking = {u12}
    controllable = {c01}
    nonblocking
}

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

Listing 5.2: Ejemplo de Controller y 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.

TO ADD

- TEST 1, Cuando encuentra el primer loop no controlable, propaga mal WEAK, cuando vuelve a ver el nodo inicial. El problema es que no vuelve a poner al nodo inicial en open, entonces no ve el otro camino, que lleva al goal.

- TEST 7, caso donde encontraba un loop que pensaba era goal pero luego, recién al querer crear controloador, fallaba porque tenía cosas por explorar. Esto era doblemente malo porque se puede tener una conclusión erronea y no se sigue explorando.

- TEST 19, 22, daban que no habia controlador y son controlables

- TEST 26? este puede ser interesante porque son dos loops no controlables pero ganas en ambos

- TEST 35 El tema es que arma un CCC y no chequea que sea valido 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 = 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 5.3: 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.

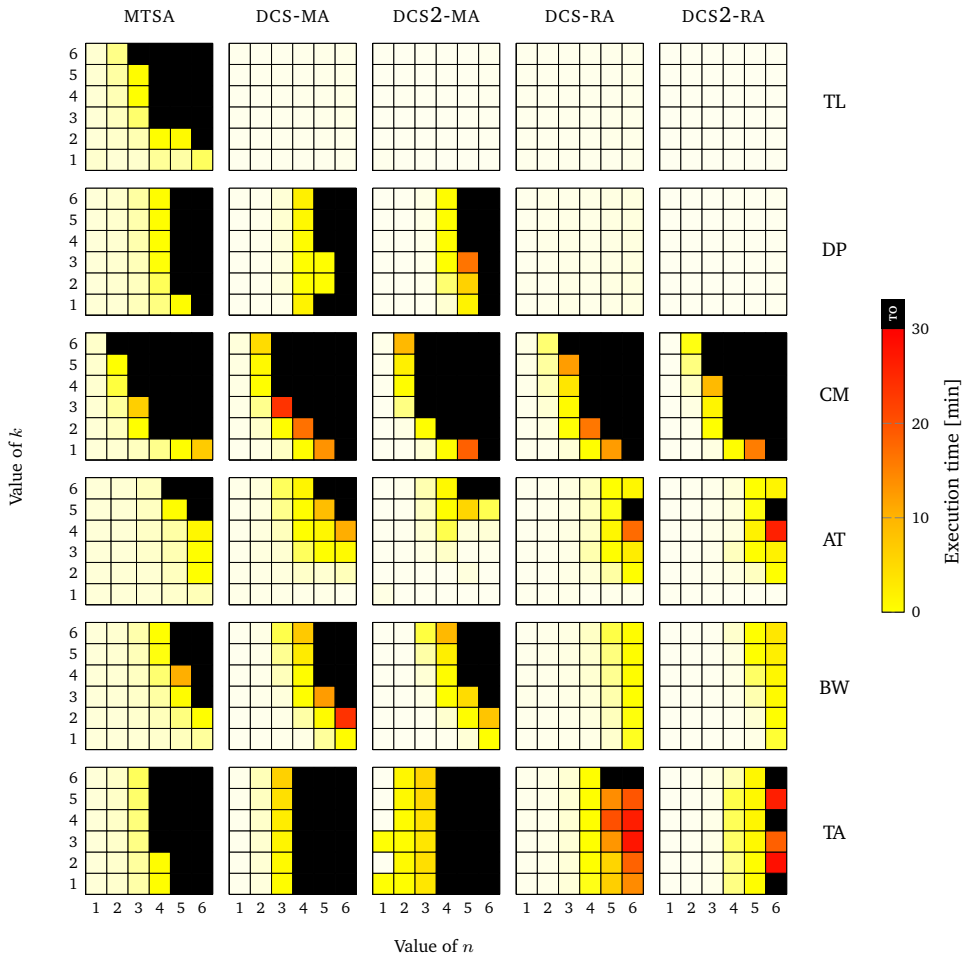


Fig. 6.1: dcs2 = performance nuevo algoritmo

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.

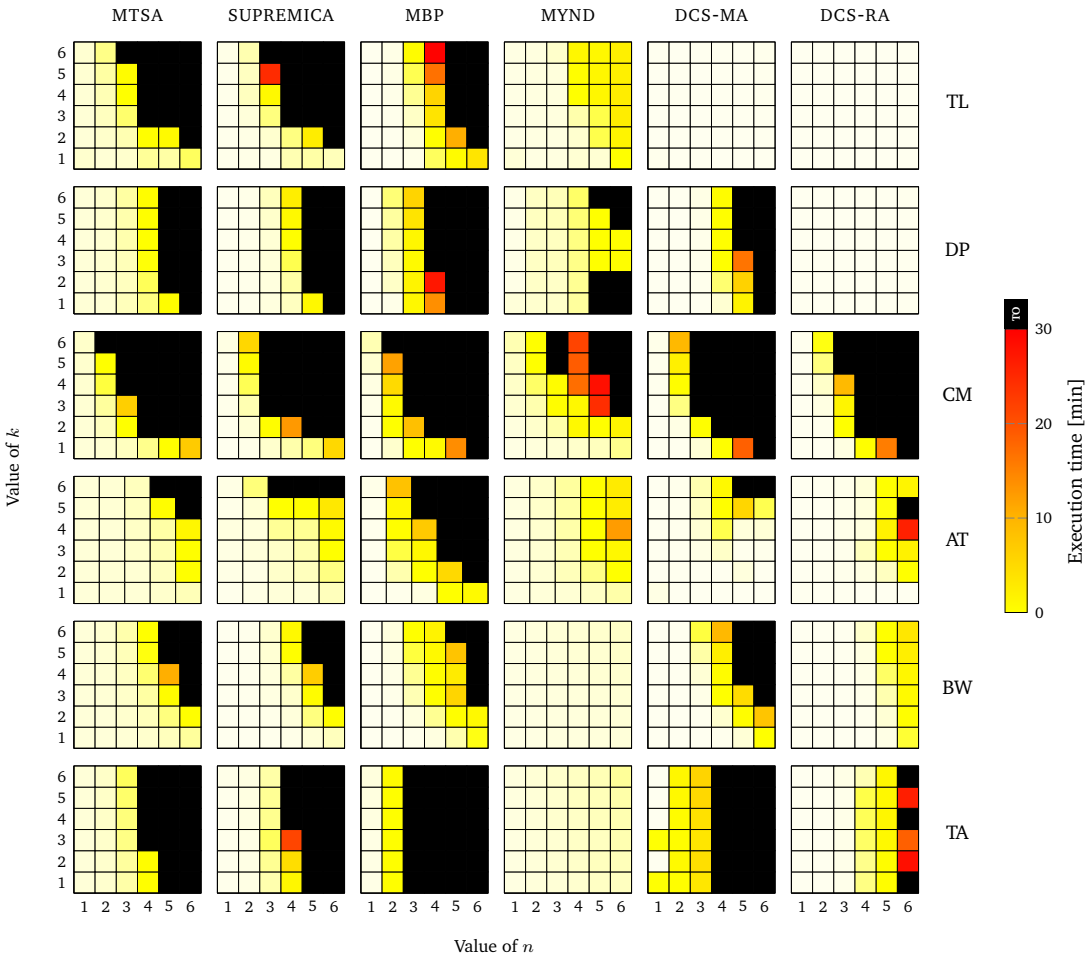


Fig. 6.2: comparación detallada

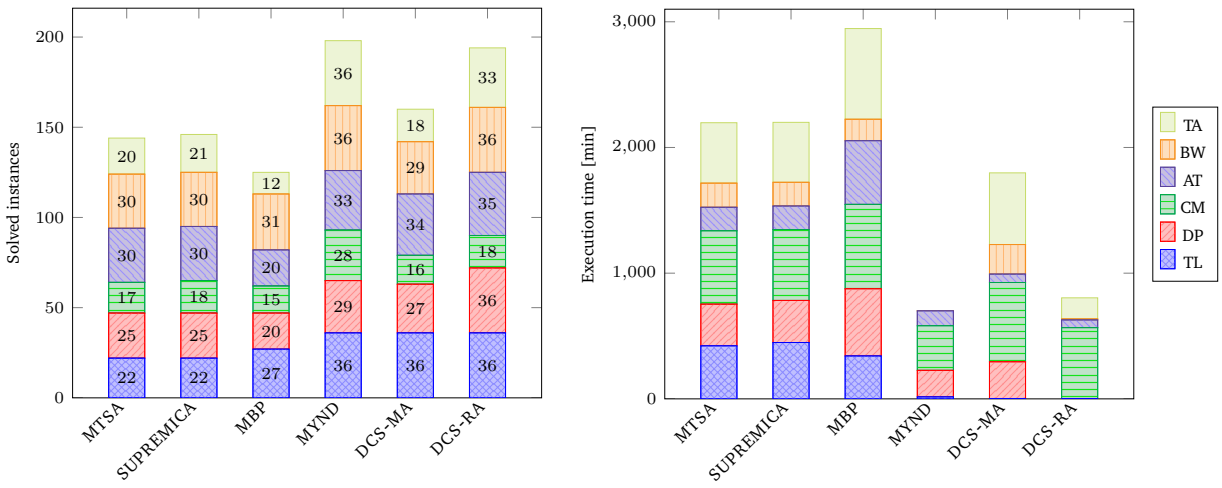


Fig. 6.3: Cantidad total de instancias resueltas (izquierda) y tiempo de ejecución (derecha) con DCS y otras herramientas.

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.