Reparación Automática de Planes de Producción usando SOAR-RL

Juan Cruz Barsce[1], Jorge Palombarini[1,2], Ernesto Martinez[3]

[1] DEPARTAMENTO DE SISTEMAS (UTN) - Av. Universidad 450, X5900 HLR, Villa María, Argentina.
jbarsce@frvm.utn.edu.ar
[2] GISIQ(UTN) - Av. Universidad 450, X5900 HLR, Villa María, Argentina.
jpalombarini@frvm.utn.edu.ar
[3] INGAR(CONICET-UTN), Avellaneda 3657, S3002 GJC, Santa Fe, Argentina.
ecmarti@santafe-conicet.gob.ar

Abstract. *Asegurar una manufactura de productos flexible y eficiente en un ambiente cada vez más dinámico y turbulento sin sacrificar costo-beneficio, calidad del producto y entrega a tiempo se ha vuelto una cuestión fundamental para la mayoría de las industrias. Un enfoque promisorio para enfrentar este desafío sugiere la integración de capacidades cognitivas en sistemas y procesos, buscando expandir la base de conocimiento usada para realizar tareas administrativas y operacionales. En este trabajo, se propone un enfoque novedoso de rescheduling en tiempo real, el cual permite obtener considerables mejoras en flexibilidad y adaptabilidad de los sistemas de producción por medio de la integración de capacidades cognitivas artificiales, involucrando percepción, razonamiento, aprendizaje y habilidades de planeamiento. Por otra parte, se discute un ejemplo industrial donde las capacidades cognitivas proporcionadas por el entorno de programación SOAR son integradas en un prototipo de software, mostrando cómo el enfoque permite al sistema de rescheduling responder a un evento disruptivo de forma autonómica y adquirir experiencia a través de simulación intensiva mientras realiza tareas de reparación.*

Palabras clave. Rescheduling, Arquitectura Cognitiva, Sistemas de Manufactura, Aprendizaje por Refuerzo, SOAR.

1 Introducción

En los últimos años, el control efectivo de los sistemas de producción se ha vuelto cada vez más complejo, debido a los crecientes requerimientos de flexibilidad y productividad, sumados a la decreciente predictibilidad de las condiciones del entorno en las plantas de producción.

Esta tendencia ha sido acompañada por altos niveles de incertidumbre presente en las industrias en términos de una creciente cantidad de productos, sus variantes (cada una con una configuración específica), fluctuaciones a gran escala de la demanda y despacho prioritario de órdenes. Los sistemas de manufactura actuales deben poder hacer frente a la competencia a nivel global, las masivas demandas de personalización y la cobertura de los pequeños nichos de mercado. Para ello, deben ser principalmente colaborativos, flexibles y responsivos [1] sin sacrificar costo-beneficio, calidad de producto y cumplimiento a tiempo de la entrega ante la presencia de eventos imprevistos como fallas en el equipamiento, re-procesamiento de operaciones demandadas por test de calidad, pedidos urgentes, retrasos en los arribos de materiales y arribo de nuevas órdenes [2]. En este contexto, los componentes de rescheduling reactivo se han vuelto un elemento clave para cualquier estrategia de gestión de disrupciones, puesto que las condiciones antes mencionadas hacen que la programación y los planes de producción se vuelvan inefectivos tras poco tiempo en la planta de producción. Los mencionados eventos además crean oportunidades para mejorar el rendimiento en planta, basadas en la situación encontrada antes del imprevisto [2].

La mayoría de los enfoques modernos para hacer frente al problema de rescheduling involucran algún tipo de programación matemática para explotar las peculiaridades de la estructura específica del problema, teniendo en mente la eficiencia del plan [3, 4] o la estabilidad del mismo [5]. Más recientemente, Gersmann y Hammer desarrollaron una mejora sobre una estrategia interactiva de reparación

utilizando máquinas de soporte vectorial [6]. Sin embargo, la representación basada en *features* resulta muy ineficiente para generalizar a estados no visitados, no presenta una lógica de reparación clara al usuario final, y la transferencia de conocimiento hacia dominios desconocidos de scheduling no es factible [7]. De esta manera, algunos investigadores han identificado la necesidad de desarrollar metodologías de rescheduling interactivas para poder alcanzar mayores grados de flexibilidad, adaptabilidad y autonomía en los sistemas de manufactura [1, 2, 8]. Estos enfoques requieren la integración de capacidades cognitivas cercanas al nivel humano, junto con habilidades de aprendizaje, razonamiento e inteligencia en componentes de rescheduling. Las mismas les permiten razonar utilizando cantidades sustanciales de conocimiento apropiadamente representado, aprender de su experiencia para mejorar su rendimiento con el paso del tiempo, explicarse y recibir órdenes, ser conscientes de sus propias capacidades para poder reflejarlas en su comportamiento, y responder robustamente a imprevistos [9]. Al incorporar continuamente información en tiempo real del ambiente de la planta de producción, el sistema de manufactura y la configuración individual del producto (a través de sensores abstractos), el componente de rescheduling va ganando gradualmente experiencia para así actuar y decidir de forma autónoma para contrarrestar cambios abruptos y situaciones inesperadas por medio de actuadores abstractos [10, 11].

En el presente trabajo se presenta un novedoso enfoque de rescheduling en tiempo real que recurre a las capacidades generales de una arquitectura cognitiva e integra representaciones simbólicas de los estados del schedule por medio de operadores de reparación abstractos. Para aprender una política de reschedule cercana a la óptima usando transiciones simuladas en los estados del schedule, se propone una estrategia interactiva basada en reparación, teniendo en cuenta las distintas metas y escenarios. Para este fin, el conocimiento específico del dominio para el scheduling reactivo es desarrollado e integrado con los mecanismos de aprendizaje de la arquitectura cognitiva SOAR, como el *chunking* y el aprendizaje por refuerzo, utilizando memorias de largo plazo [12]. Por último, se discute un ejemplo industrial mostrando cómo el enfoque le permite al sistema de scheduling evaluar su rango de operación de forma autonómica y adquirir experiencia a través de simulación intensiva mientras realiza tareas de reparación en los schedules de producción.

2 Rescheduling en Tiempo Real empleando SOAR-RL

En este enfoque, el conocimiento sobre heurísticas de rescheduling basado en reparación para lidiar con imprevistos y perturbaciones es generado y representado recurriendo al uso de un simulador de estados del schedule conectado a la arquitectura cognitiva SOAR [13]. En el ambiente de simulación, una instancia del schedule es modificada interactivamente por el sistema usando una secuencia de operadores de reparación sugeridos por SOAR. Esto es realizado hasta que el objetivo de reparación es alcanzado o bien se acepta la imposibilidad de reparar el schedule. SOAR resuelve el problema de generar y codificar conocimiento de rescheduling usando Teoría General de Computación, que se basa en objetivos, espacios de problema, estados y operadores, los cuales serán explicados en detalle más adelante. La figura 1 presenta una vista estructural de SOAR, con sus memorias primitivas representadas por rectángulos de bordes cuadrados, sus procesos por rectángulos redondeados y sus conexiones por flechas.

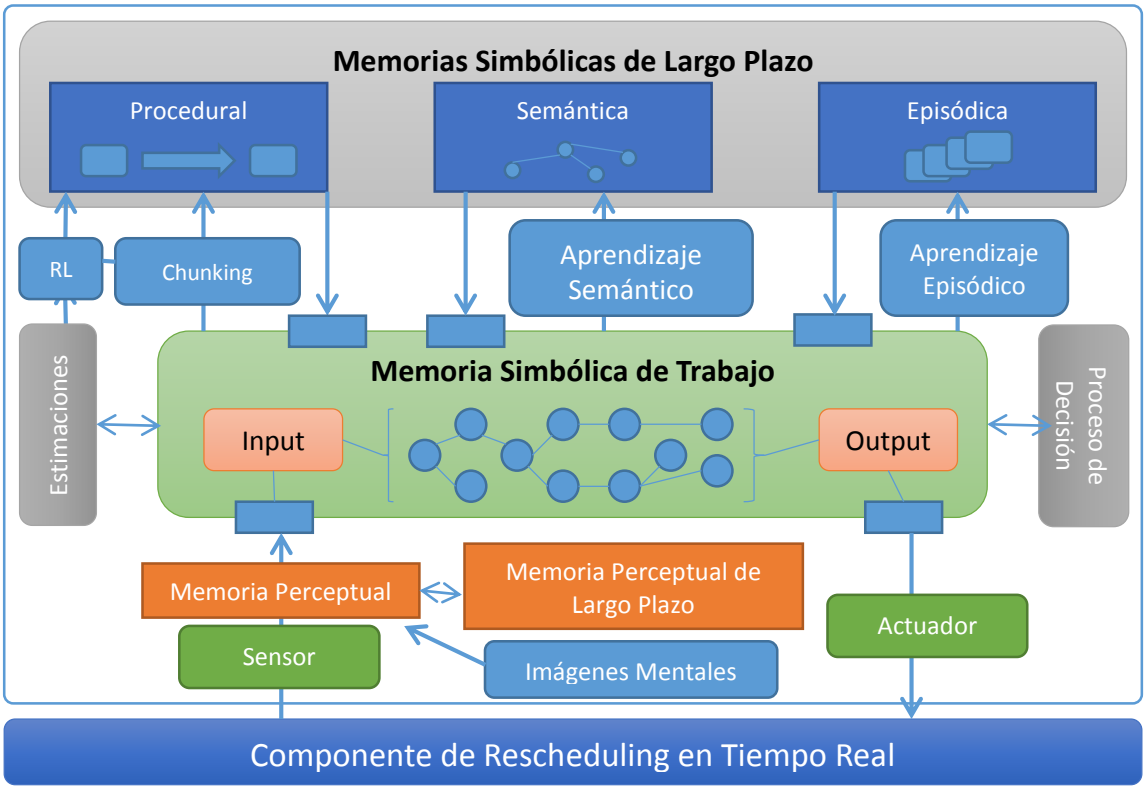


Fig. 1. Diagrama de bloques que representa la arquitectura cognitiva SOAR

Para implementar el enfoque propuesto, la arquitectura cognitiva se conecta con el componente de rescheduling por medio de una interfaz, que a su vez se comunica con .NET. La información sobre el estado actual del schedule se recupera en el módulo de percepción, el cual se relaciona con una estructura *InputLink*, y es almacenado en la memoria perceptual a corto plazo. Desde allí se extraen estructuras de estado de primer orden simbólicas en forma de valores id-atributo las cuales posteriormente se agregan a la memoria de trabajo de SOAR. La misma actúa como una memoria global a corto plazo que consulta el conocimiento de rescheduling almacenado en la memoria de largo plazo y sirve como base para iniciar las acciones de reparación del schedule. Las tres memorias simbólicas de largo plazo son independientes entre sí, y usan mecanismos separados de aprendizaje. La memoria procedural de largo plazo es responsable de consultar el conocimiento que controla el procesamiento, representado como reglas si-entonces conocidas como reglas de producción. Las mismas comparan las condiciones con los contenidos de la memoria de trabajo, ejecutando sus acciones en paralelo. Las reglas de producción pueden modificar la memoria de trabajo (y, por lo tanto, el estado del schedule). Para controlar el comportamiento de rescheduling, las reglas generan preferencias que son usadas por los procedimientos de decisión para elegir un operador de reparación del schedule. Los operadores son un componente clave en este enfoque, porque pueden ser aplicados causando cambios persistentes en la memoria de trabajo, y tienen un espacio reservado de memoria que es monitoreado a su vez por otras memorias y procesos. Es por eso que un cambio en la memoria de trabajo puede desencadenar consultas desde las memorias semántica y episódica o iniciar acciones de reparación del schedule a través del actuador abstracto en el ambiente. La memoria semántica guarda estructuras generales de primer orden que pueden ser empleadas para resolver nuevas situaciones. Por ejemplo, si en un estado del schedule `s1` las siguientes relaciones `precede (tarea1, tarea2)`

y `precede (tarea2, tarea3)` (las cuales comparten el parámetro "tarea2") son verificadas, la memoria semántica puede agregar la relación abstracta `precede (A, B)` y `precede (B, C)` para generalizar ese conocimiento. Por otra parte, la memoria episódica almacena la experiencia en forma de cadenas `estado – operador … estado – operador`. Dicho conocimiento puede ser usado para predecir comportamiento o dinámicas ambientales en situaciones similares, o bien prever el estado del schedule más allá de la percepción inmediata, utilizando su experiencia para predecir los valores asociados a los posibles cursos de acción de reparación del schedule. Además, este enfoque utiliza dos mecanismos de aprendizaje específicos asociados con la memoria procedural de SOAR: chunking y aprendizaje por refuerzo [14].

El chunking continúa el proceso de aprendizaje con nuevas reglas de producción al tiempo que el proceso de reparación es realizado como un procedimiento automático que convierte la búsqueda del espacio del problema en un estado de schedule en conocimiento que puede ser accedido posteriormente mediante búsqueda. El aprendizaje por refuerzo, por su parte, ajusta las acciones de reparación de las reglas que crean preferencias para la selección de operadores (explicados en detalle más adelante). Además, la memoria semántica almacena hechos generales, mientras que la memoria episódica guarda instantáneas de la memoria de trabajo, así ambas pueden ser accedidas creando consultas desde la misma. Por último, los operadores de reparación sugeridos por el proceso de decisión de SOAR afectan el estado del schedule y se conectan con los componentes de rescheduling en tiempo real a través de una estructura de *OutputLink*.

3 Representación de Estados del Schedule, Operadores de Reparación y Objetivos de Rescheduling en SOAR

La memoria de trabajo de SOAR mantiene el estado actual del schedule, y se encuentra organizada como una estructura de grafo conexo (red semántica), anclada a un símbolo que representa el estado. Los nodos no terminales de la gráfica se llaman identificadores, los arcos se llaman atributos, y los valores son los otros nodos. Los arcos que comparten el mismo nodo se denominan objetos, los cuales consisten en todas las propiedades y relaciones de un identificador. El estado es un objeto que contiene a todos los demás objetos, los cuales directa o indirectamente son sub-estructuras del mismo. En la parte superior de la Figura 2, se muestra un ejemplo de un estado en la memoria de trabajo de SOAR llamado `<s>`. En esta figura, algunos detalles, como algunas sub-estructuras o arcos, son omitidos por motivos de claridad.

En la situación mostrada en la Figura 2, podemos ver que en la memoria de trabajo el estado actual se llama `<s>`, y contiene atributos como avgTard con un valor de 2,5 horas, initTardiness con un valor de 28,5 horas y totalWIP con 46,83, entre otros. Además, existe un atributo resource, cuyo valor es otro identificador, por lo tanto conecta el estado `<s>` con otros objetos llamados `<r1>`, `<r2>` y `<r3>`. `<r1>` se corresponde con un recurso del tipo extruder, puede procesar productos de tipo A y B con una tardanza acumulada en el recurso de 17 horas y tiene tres tareas asignadas a él: `<t1>`, `<t11>` y `<t3>`. Estas tareas a su vez también son objetos de forma que, por ejemplo, `<t1>` tiene atributos propios que la describen como productType, previous, next, quantity, duration, entre otros.

Al igual que para las tareas, los recursos tienen un atributo importante llamado processingRate que determina la cantidad total de un tipo de producto que el recurso puede procesar. Por último, un atributo del estado llamado *calculations* almacena la información relevante al cálculo de la tardanza total, tardanza por recurso, estado de las operaciones y otros valores similares.

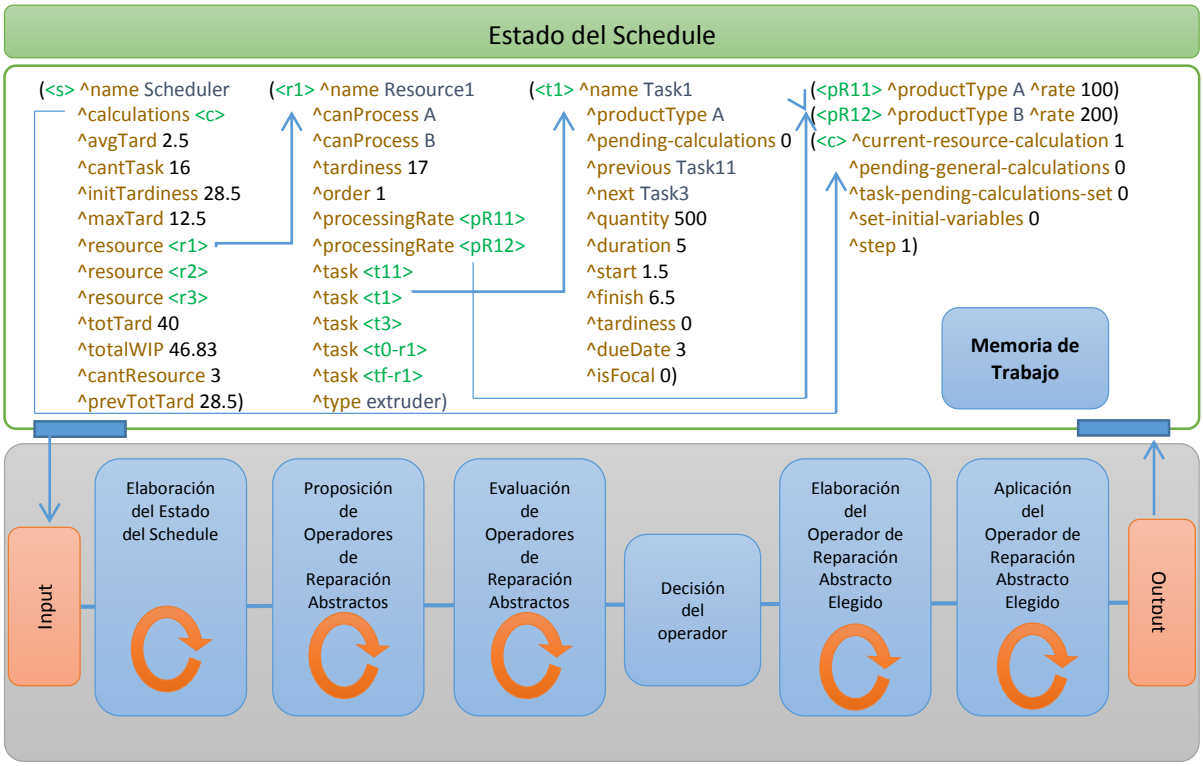


Fig. 2. Representación simbólica del estado del schedule (arriba) y elaboración del ciclo de reparación (abajo).

El estado es un componente clave en SOAR para resolver instancias de problemas de rescheduling, debido a que los programas SOAR están implícitamente organizados en términos de Modelos Computacionales de Espacios de Problema [15]. Las condiciones para la proposición de operadores de reparación restringen el espacio de operadores para considerarlos sólo cuando estos sean relevantes, y así definir el espacio de posibles estados que podrían ser considerados para cumplir con los objetivos de reparación. En este enfoque, el espacio de problema se compone por todos los posibles schedules que pueden ser generados con las tareas actuales, y todos los operadores de reparación que dan lugar a que el schedule pase de un estado a otro. No obstante, la arquitectura no genera explícitamente en cada proceso de rescheduling todos los estados de forma exhaustiva, sino que SOAR se encuentra en un estado específico del schedule en un tiempo determinado (representado por la memoria de trabajo). En cada estado intenta seleccionar un operador con el que tratará de mover el schedule a un estado mejorado. El proceso continúa recursivamente hacia un estado objetivo (por ejemplo, cuando la tardanza total del schedule es menor que la inicial). La parte inferior de la Figura 2 muestra la ejecución del proceso de reparación del schedule, que procede a través de un número de ciclos. Cada ciclo es aplicado en fases. En la fase de entrada, la memoria de trabajo obtiene nuevos datos provenientes de sensores. En la fase de elaboración se disparan nuevas reglas de producción (y se retractan otras) para interpretar los nuevos datos y generar conocimientos derivados. En la fase de proposición la arquitectura propone operadores de reparación para el schedule actual utilizando preferencias, para luego ser comparados en la fase de evaluación. Todas las producciones que hagan *matching* se disparan en paralelo (al igual que todas las retracciones) hasta que no haya más reglas que hagan matching completamente o haya retracciones de reglas de producción (estado conocido como estancamiento o

quiescence). Por lo tanto, un proceso de decisión selecciona un nuevo operador basado en las preferencias numéricas provistas por las reglas de aprendizaje por refuerzo. Una vez que un operador de reparación es seleccionado, ejecuta su fase de aplicación y se disparan las reglas de aplicación específicas del operador. Las acciones de estas producciones dan lugar a más *matches* o retractaciones; tal como en la fase de proposición, las producciones se disparan y retractan en paralelo hasta encontrarse con el estado de *quiescence*. Por último, los comandos de salida son enviados al componente de rescheduling de tiempo real. El ciclo continúa hasta que el programa recibe una orden de detención desde el programa SOAR (como la acción de una regla de producción).

3.1 Diseño e Implementación de Reparadores de Operación y Objetivos de Rescheduling

Tal como fue explicado previamente, los operadores de reparación son la forma en la que el schedule pasa de un estado a otro hasta alcanzar el objetivo del rescheduling. Tal situación es monitoreada por una regla de elaboración, como se muestra en el siguiente ejemplo:

```
Ejemplo 1. sp
{Scheduler*elaborate*done
              (state <s> ^name
Scheduler
           ^calculations.pending
           -general-calculations
           0    ^initTardiness
           <initTard>
             ^totTard <tottard>
< <initTard>)
--> (halt)}
```

La regla de elaboración establece que si en el estado `<s>` el nombre del problema es Scheduler, no hay cálculos generales numéricos pendientes (relacionados con valores en las tareas como la duración, tardanza total, etc.), la tardanza inicial existe y es un valor ligado a la variable `<initTard>`, la tardanza total existe y es un valor ligado a la variable `<tottard>`, y `<tottard>` es menor que `<initTard>`, entonces el proceso asociado con la reparación del schedule debe finalizar.

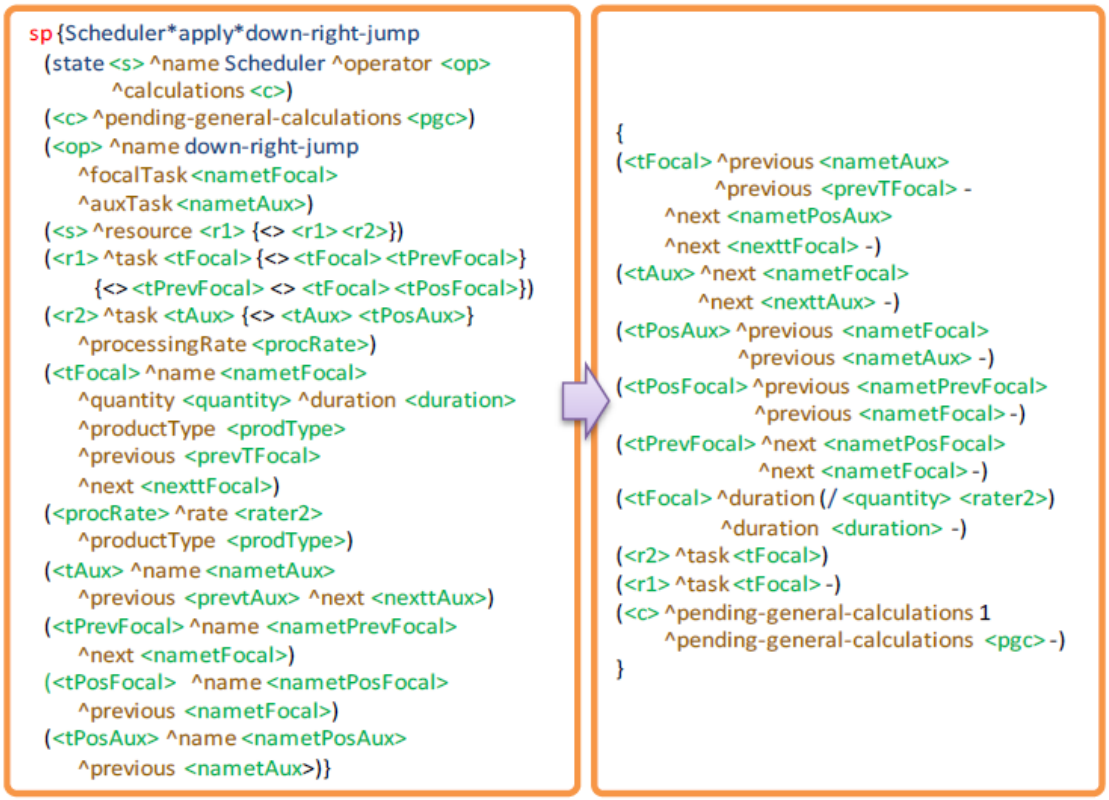


Fig. 3. Aplicación de la regla Down-right-jump

Por otra parte, los operadores de reparación deícticos han sido diseñados para mover o intercambiar una tarea focal con otras tareas en el schedule (las cuales pueden estar asignadas a otro recurso), con el fin de alcanzar un estado objetivo [11]. Cada operador toma dos argumentos: la tarea focal y la tarea auxiliar. La tarea focal es tomada como punto de anclaje de la reparación, y la tarea auxiliar sirve para especificar la acción de reparación y evaluar su efectividad. Por ejemplo, si el operador propuesto es down-right-jump, la idea es que la tarea focal sea movida a un recurso alternativo e insertada después de la tarea auxiliar. Si es así, las condiciones de las reglas de proposición del operador down-right-jump deben asegurar que la tarea auxiliar tenga un tiempo de inicio programado mayor que el tiempo de inicio antes de haber sido movido. Es importante destacar que en el recurso alternativo puede existir más de una tarea que cumpla la condición; en tal caso, el operador es propuesto en paralelo, parametrizado con tareas auxiliares diferentes. Para ejemplificar este razonamiento, la Figura 3 muestra la regla de aplicación del operador down-right-jump (*left hand side* a la izquierda de la figura y *right hand side* a la derecha).

El *left hand side* de la regla en la Figura 3 establece las condiciones que deben ser cumplidas por el estado del schedule para que el operador pueda ser aplicado. A su vez, el right hand side define cómo el estado del schedule cambia como consecuencia de la aplicación del operador de reparación. Todos los símbolos encerrados en "<>" representan variables, y las variables con el mismo nombre se refieren al mismo objeto en ambas partes de la regla. Por lo tanto, la regla en la figura 3 puede ser semánticamente expresada como: "si en el estado `<s>` del schedule, el nombre del espacio del problema es Scheduler, el valor de pending-general-calculation es `<pgc>`, existe un operador propuesto llamado down-right-jump que toma como argumento una tarea focal llamada `<nametFocal>` y una tarea auxiliar `<nametAux>`; también, existe un recurso `<r1>` que tiene asignadas las tareas `<tFocal>`, `<tPrevFocal>` y `<tPosFocal>`, las cuales son distintas entre sí, existe un recurso `<r2>` con un ratio de procesamiento `<rater2>` del tipo de producto `<prodType>`, el cual tiene asignadas las distintas tareas `<tAux>` y `<tPosAux>`. La tarea `<tFocal>` es la tarea focal del operador de reparación y sus atributos toman los valores `<quantity>`, `<duration>` y `<prodType>`, la tarea previamente programada en el recurso es `<prevTFocal>` y la próxima es `<nexttFocal>`. La tarea `<tAux>` tiene como nombre `<nametAux>` y es la tarea auxiliar en el operador de reparación y tiene una tarea previamente programada llamada `<prevtAux>` y una tarea siguiente `<nexttAux>`. Además, la tareas `<tPrevFocal>` y `<tPosFocal>` tienen como nombres `<nametPrecFocal>` y `<namePosFocal>`, respectivamente, y como resultado de la aplicación del operador el estado del schedule cambia de la siguiente manera: la nueva tarea previa de `<tAux>` es `<nametFocal>`, la nueva tarea siguiente de `<tAux>` es `<nametFocal>`, la nueva tarea previa de `<tPosAux>` es `<nametFocal>` y la nueva tarea previa de `<tPrevFocal>` es `<nametPosFocal>`. Además, el nuevo valor de la duración de la tarea focal es `(/ <quantity> <rater2>)`, la tarea focal es movida al recurso `<r2>` y removida de `<r1>`, y se actualiza el valor de pending-general-calculations a 1. Como resultado de la aplicación de la regla, la tarea focal ha sido movida a un recurso alternativo, su duración ha sido recalculada (debido al cambio de ratio de procesamiento al pasar al recurso alternativo), y ha sido insertada después de la tarea auxiliar y la tarea originalmente programada para empezar antes. El valor de 1 en pending-general-calculations dispara nuevos operadores con el objeto de re-calcular la tardanza de cada tarea, cada recurso, y otros valores numéricos. Una consideración importante es que, una vez

que el operador de reparación y sus argumentos han sido obtenidos, el resto de los valores variables pueden ser totalmente definidos porque se relacionan entre sí, permitiendo una aplicación efectiva del operador de reparación. Otra ventaja de este enfoque se basa en el uso de variables en el cuerpo de la regla, las cuales pueden actuar como un cuantificador universal, y de esa forma la definición del operador de reparación y su aplicación pueden hacer match en situaciones y tipos de schedule totalmente diferentes únicamente cumpliendo con las restricciones relacionales establecidas en el left hand side de las reglas.

Por último, después de que los cambios en el schedule hayan sido realizados por la regla de aplicación del operador, se disparan las reglas SOAR de aprendizaje por refuerzo para que la arquitectura pueda aprender en forma de preferencias numéricas a partir de los resultados de las aplicaciones particulares de operadores de reparación, los cuales son llevados a cabo utilizando una función de reward y el algoritmo SARSA(λ) [16]. La función de reward es definida como la cantidad de tardanza reducida (reward positivo) o aumentada (reward negativo). Por lo tanto, el algoritmo SARSA(λ) actualiza la preferencia numérica del operador usando la bien conocida fórmula ilustrada en la Ecuación (1)

$$Q(s,ro)_{t+1} = Q(s,ro)_t + \alpha[r + \gamma Q(s',ro')_t - Q(s,ro)_t]e(s,ro)_t \qquad (1)$$

donde $Q(s,ro)$ es el valor de aplicar el operador de reparación `ro` en el estado `s` del schedule, mientras que α y γ son los parámetros del algoritmo, r es el valor de reward mientras que $e(s,ro)$ es el valor de la traza de elegibilidad para el operador de reparación `ro` en el estado `s`. Debido a que el espacio del problema puede ser extremadamente largo y los valores de Q son guardados en las reglas de producción que no pueden ser preestablecidas, debe definirse un template de generación de reglas de aprendizaje por refuerzo [12] para poder generar preferencias numéricas actualizables que sigan las especificaciones SOAR y desarrollar el procedimiento de aprendizaje cada vez que se visiten estados del schedule por medio de operadores de reparación.

4 Caso de estudio industrial

Para ilustrar nuestro enfoque para el rescheduling automático de tareas se considera un problema propuesto en [17] por Musier y Evans. El mismo consiste en una planta que procesa por lotes y consta de tres extrusoras semi-continuas que procesan pedidos personalizados para cuatro clases de productos: A, B, C y D. Cada extrusora puede procesar un producto a la vez y posee características propias. Por ejemplo, el ratio de producción para cada tipo de producto puede variar de una extrusora a otra, y no necesariamente cada extrusora puede procesar todo tipo de productos. Cada tarea, por su parte, tiene un tiempo de entrega, una cantidad requerida (medida en Kg.) y un tipo de producto.

Fueron utilizadas tres aplicaciones para implementar y testear el caso de estudio: VisualSoar v4.6.1, SoarDebugger 9.3.2 [18] y Visual Studio 2010 Ultimate, todas bajo Windows 7. Visual Studio 2010 fue usado para desarrollar el componente de scheduling en tiempo real, el cual permite validar los resultados y leer / escribir en los input y output links de SOAR, respectivamente. El entorno VisualSoar se usó para diseñar e implementar la definición del estado del schedule y el conocimiento para la proposición, elaboración y aplicación de los operadores. Para definir la estructura del estado del schedule, se implementó un Datamap en SOAR, a fin de especificar la semántica y relaciones entre los

identificadores, atributos y valores. Esta es una característica importante de SOAR, puesto que le permite al diseñador de conocimiento revisar la consistencia de las reglas y detectar la definición de atributos que no son revisados ni modificados por ninguna regla. Es por eso que resulta una herramienta muy útil para mantener la coherencia interna de hechos lógicos definidos por la elaboración y la estructura de proposición y aplicación de operadores. El entorno VisualSoar también permite escribir y revisar automáticamente todos los tipos de reglas antes mencionados, y establecer los parámetros en los procesos de aprendizaje por refuerzo y chunking. Aparte de los operadores de reparación previamente explicados, fueron diseñados operadores de cálculo para actualizar el estado numérico de variables como la tardanza total, pero sin producir ninguna actualización en relación al aprendizaje de las preferencias numéricas de los operadores de reparación. Además, un operador llamado monitor-calculations-complete está a cargo de detectar cuando los valores numéricos de las variables de estado han sido completamente actualizados; de ser así, la aplicación del operador actualiza el valor de pending-general-calculations, incrementa los pasos de reparación en uno y establece el reward asignado al último operador de reparación. Por su parte, el entorno SoarDebugger fue utilizado para correr las reglas previamente mencionadas y entrenar al agente de rescheduling, como así también para analizar la exactitud de las reglas de proposición/aplicación y las operaciones de cálculo.

Para el espacio del problema de rescheduling, hubo un máximo de diez operadores de reparación propuestos para todo estado en cada paso de reparación. Los mismos se dividen en dos clases de operadores: operadores de movimiento, los cuales mueven la tarea focal hacia otra posición en el mismo recurso o en uno alternativo, y operadores de intercambio, que intercambian la tarea focal con otra tarea en distintos recursos.

Luego de cada paso de reparación, el agente de rescheduling verifica si el schedule ha sido reparado, usando la regla precedente en el Ejemplo 1. En tal caso, la meta ha sido alcanzada y en consecuencia la arquitectura es detenida. De otra manera, el agente propone/aplica un nuevo operador hasta alcanzar la meta, o hasta que hayan transcurrido una excesiva cantidad de episodios sin haberse encontrado la solución del problema de rescheduling. Esta situación suele darse cuando el schedule a ser reparado es muy similar al schedule óptimo, por lo que resulta muy difícil realizar mejoras adicionales. También, por cada aplicación del operador de reparación, se otorga un reward al agente basado en qué tan cerca la tardanza del schedule actual está de la tardanza inicial (en otras palabras, cuán cerca está el schedule actual del estado objetivo).

En este trabajo, el evento disruptivo que ha sido considerado es la llegada de un nuevo pedido. Por lo tanto, en el proceso de aprender a insertar un nuevo pedido, el escenario de rescheduling es descrito por 1) la llegada del pedido con atributos dados que debe ser insertado en un schedule aleatoriamente generado teniendo en cuenta las posibilidades ofrecidas de alcanzar un estado reparado, y 2) los atributos del nuevo pedido que son también generados automáticamente. Para el aprendizaje se utilizó el algoritmo Sarsa (λ) con una política ε-greedy, trazas de elegibilidad, y los siguientes parámetros: $\gamma = 0,9$; $\varepsilon = 0,1$; $\lambda = 0,1$; $\alpha = 0,1$.

Fueron ejecutados 20 episodios en la fase de entrenamiento del agente de rescheduling. En algunos, el agente alcanzó el objetivo de reparación, expresado como “inserte el nuevo pedido reduciendo la tardanza total presente en el schedule previa a dicha inserción”. La intención detrás del objetivo de reparación fue por un lado insertar la nueva orden de forma eficiente, y por otro tomar ventaja de la oportunidad de mejorar la eficiencia del schedule, la cual puede ser causada por la sola ocurrencia del evento disruptivo. También, como

consecuencia de la naturaleza progresiva del proceso de aprendizaje y la convergencia del aprendizaje por refuerzo, en los episodios iniciales el agente ocasionalmente quedó atascado con una tardanza total por encima de las 80 horas al alcanzar el máximo permitido de 400 movimientos. Como resultado de la fase de entrenamiento, se generaron dinámicamente 2520 reglas de aprendizaje por refuerzo. Un ejemplo representativo de tales reglas puede verse debajo en el Ejemplo 2 (algunos componentes del lado izquierdo se omitieron para facilitar la lectura y comprensión semántica).

```
Ejemplo 2. sp
{rl*Scheduler*rl*rules*157
     (state <s1> ^totalWIP 46.83
     ^taskNumber 16 ^maxTard 15
     ^avgTard 2.5 ^totTard 40
     ^initTardiness 28.5 ^name
     Scheduler ^operator <o1> +
     ^focalTask <t1>)
     (<o1> ^auxTask Task10
     ^focalTask Task5 ^name up-
     right-jump)
      --> (<s1> ^operator <o1> =
-0.1498)}
```

La regla del Ejemplo 2 fue automáticamente generada SOAR, y se transfiere a las siguientes operaciones de reparación del schedule, por lo que el uso de las reglas para rescheduling aprendidas puede hacerse de forma reactiva en tiempo real sin deliberaciones extras. La regla del Ejemplo 2 se lee como sigue: si el estado <s1> llamado Schedule tiene una trabajo total en proceso (totalWIP) de 46.83, un número de tarea de 16, una tardanza máxima de 15, una tardanza promedio de 2,5, una tardanza total de 40, una tardanza inicial de 28,5, una tarea focal <t1> y el operador de reparación aplicado es up-right-jump, el cual tiene una tarea auxiliar Task10 y una tarea focal Task5, entonces el valor de Q de la aplicación de esa operación de reparación es de -0,1498. Evaluando esos valores para cada operador, el agente puede determinar cuál es la mejor situación y actuar en consecuencia.

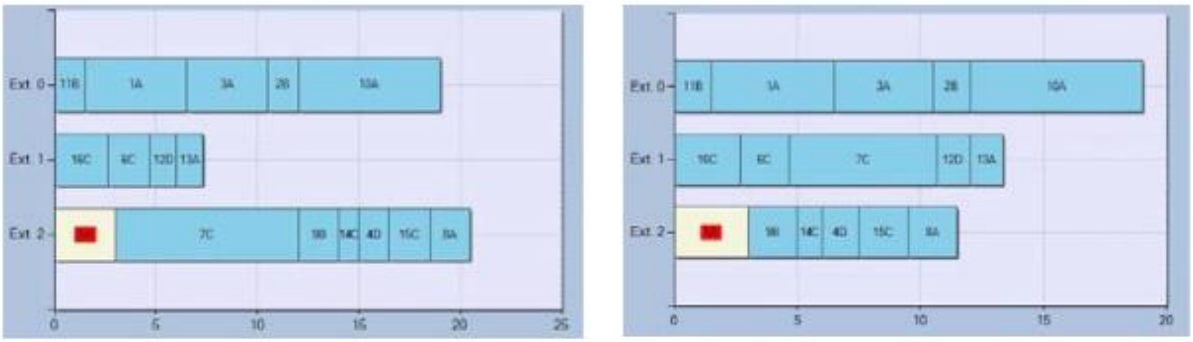
Una a vez que el proceso de aprendizaje ha sido realizado, se genera un nuevo schedule para testear la política de reparación aprendida. Los detalles del pedido son mostrados en la Tabla 1, donde el nuevo pedido es resaltado en amarillo. Para generar el schedule inicial, se asignaron 15 órdenes a 3 recursos, tal como puede verse en la Figura 4. Previo a la inserción de la tarea focal, la tardanza total del schedule era de 28,5 horas (ver Fig. 4ª). Luego de seis pasos de reparación usando la base de conocimiento SOAR-RL, el schedule inicial fue reparado con éxito y la tardanza total fue reducida a 18,5 h.

La secuencia de operadores de reparación aplicada por SOAR fue la siguiente: el primer operador fue up-right-jump al Resource1 (por lo tanto la tarea focal fue insertada entre Task3 y Task2), incrementando la tardanza total a 44 h. Eso fue sucedido por un down-right-jump al Resource2 (detrás de Task16), reduciendo la tardanza total a 28,5 h. El tercer operador aplicado fue un up-right-jump al Resource1 (delante de Task10), incrementando la tardanza total a 46,5 h. El siguiente fue un down-right-jump al Resource3 (entre Task14 y Task4). SOAR luego aplicó un up-left-jump al Resource2 (entre Task6 y Task12). Por último, se aplicó un down-left-swap al Resource3 con Task7, dejando el schedule con una tardanza total de 18,5 h.

Como puede verse en la secuencia de reparación, la política de rescheduling intenta obtener un schedule balanceado usando la tarea focal como base para intercalar las posiciones de los pedidos en secuencia de schedules generados, para así tomar ventaja de las extrusoras con los mejores tiempos de procesamiento. En este caso, el agente de rescheduling intenta re-localizar la tarea focal en un recurso alternativo a fin de hacer un intercambio con Task7 para moverla a la segunda sub-utilizada extrusora. Es importante notar el pequeño número de pasos requeridos para el agente de scheduling para implementar la política aprendida para manejar la inserción de un nuevo pedido.

Orden #	Producto	Peso [Kg]	DD [h]
1	A	500	3
2	B	300	7
3	A	400	12
4	D	300	4
5	A	300	4
6	C	300	10
7	C	900	7
8	A	200	10
9	B	300	14
10	A	700	18
11	B	300	19
12	D	400	15
13	A	200	13
14	C	100	15
15	C	200	16
16	C	400	20

Tabla 1. Datos de los pedidos iniciales del ejemplo



(a). Tardanza inicial: 28,50 h. Tardanza post-inserción: 40,00 h.

(b).Tardanza final tras aplicar la política de reparación: 18,50 h.

Fig. 4. Ejemplo de aplicación de una secuencia óptima de operadores de reparación.

5. Conclusión

Se presentó un nuevo enfoque para el aprendizaje basado en simulación de una política basada en reglas tratando con reparación automática de schedules en tiempo real usando la arquitectura cognitiva SOAR. La política de rescheduling permite generar una secuencia de operadores de reparación local para poder obtener objetivos alternativos de rescheduling que ayuden a hacer frente a eventos no planificados o anormales (tales como la inserción de una orden nueva) con tardanza mínima, basada en una representación simbólica de primer orden de los estados del schedule usando operadores de reparación abstractos. El enfoque propuesto representa eficientemente y usa grandes conjuntos de conocimiento simbólico, porque combina dinámicamente el conocimiento disponible para efectuar la toma de decisiones en forma de reglas de producción con mecanismos de aprendizaje. Puede, además, compilar la solución al problema de rescheduling en reglas de producción y de esa forma, con el tiempo, el proceso de reparación de schedules es reemplazado por la toma de decisiones basada en reglas. Esto puede ser usado reactivamente en tiempo real en forma sencilla. En ese sentido, el uso de operadores de reparación abstractos para matchear

situaciones no predefinidas representando tareas por medio de espacios de problemas y estados del schedule usando una abstracción relacional simbólica no solamente es eficiente, sino que también es potencialmente una elección muy natural para imitar la habilidad cognitiva humana de lidiar con problemas de rescheduling, donde las relaciones entre los puntos focales y los objetos para definir estrategias de reparación son típicamente usadas. Por último, al basarse en un apropiado y bien diseñado conjunto de templates de reglas, el enfoque permite la generación automática a través del aprendizaje por refuerzo y el chunking de heurísticas de rescheduling que pueden ser naturalmente entendidas por el usuario final.

Referencias

1. Zaeh, M., Reinhart, G., Ostgathe, M., Geiger, F., & Lau, C.: A holistic approach for the cognitive control of production systems. Advanced Engineering Informatics, 24, 300–307 (2010).
2. Aytug, H., Lawley, M., McKay, K., Mohan, S., Uzsoy, R.: Executing production schedules in the face of uncertainties: A review and some future directions. European Journal of Operational Research, 161, 86–110 (2005).
3. Adhitya, A., Srinivasan, R., Karimi, I. A.: Heuristic rescheduling of crude oil operations to manage abnormal supply chain events. AIChE J. 53(2), 397-422 (2007).
4. Zhu, G., Bard, J., Yu, G.: Disruption management for resource-constrained project scheduling. Journal of the Operational Research Society, 56, 365-381 (2005).
5. Novas, J. M., Henning, G.: Reactive scheduling framework based on domain knowledge and constraint programming. Comp. and Chemical Engineering, 34, 2129–2148 (2010).
6. Gersmann, K., Hammer, B.: Improving iterative repair strategies for scheduling with the SVM. Neurocomputing, 63, 271–292 (2005).
7. Morales, E. F.: Relational state abstraction for reinforcement learning. Proceedings of the Twenty-first Intl. Conference (ICML 2004), Banff, Alberta, Canada, Julio 4-8 (2004).
8. Henning, G.: Production Scheduling in the Process Industries: Current Trends, Emerging Challenges and Opportunities. Computer-Aided Chemical Engineering, 27, 23 (2009).
9. Brachman R. Systems That Know What They're Doing. IEEE Intelligent Systems, issue 6, 67-71 (2002).
10. Trentesaux, D.: Distributed control of production systems. Engineering Applications of Artificial Intelligence, 22, 971–978 (2009).
11. Palombarini, J., Martínez, E.: SmartGantt – An Intelligent System for Real Time Rescheduling Based on Relational Reinforcement Learning. Expert Systems with Applications vol. 39, pp. 10251- 10268 (2012).
12. Nason, S., Laird, J. E.: Soar-RL: integrating reinforcement learning with Soar. Cognitive Systems Research 6, 51–59 (2005).
13. Laird, J. E.: The Soar Cognitive Architecture, MIT Press, Boston (2012).
14. Nuxoll, A. M., Laird, J. E.: Enhancing intelligent agents with episodic memory. Cognitive Systems Research 17–18, 34–48 (2012).
15. Newell, A., Yost, G. R., Laird, J. E., Rosenbloom, P. S., Altmann, E. M.: Formulating the problem space computational model. In CMU Computer Science: A 25ft. Anniversary Commemorative, ed. R. Rashid. ACM Press/Addison Wesley (1991).
16. Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. MIT Press (1998).
17. Musier, R., Evans, L.: An Approximate Method for the Production Scheduling of Industrial Batch Processes with Parallel Units. Comp. and Chem. Engineering, 13, 229 (1989).
18. SOAR Suite. URL https://code.google.com/p/soar/wiki/Downloads?tm=2. Revisado: 11/05/2013.