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

# Actualización dinámica de controladores de eventos discretos

Tesis presentada para optar al título de Doctor de la Universidad de Buenos Aires en el área Ciencias de la Computación

**Lic. Leandro Ezequiel Nahabedian**

Director de Tesis: Dr. Sebastián Uchitel
Consejero de Estudios: Dr. Nicolás D'Ippolito
Lugar de Trabajo: Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires

Buenos Aires, 2020
Fecha de Defensa: 3 de julio de 2020

**Leandro Ezequiel Nahabedian**
*Actualización dinámica de controladores de eventos discretos*
3 de julio de 2020
Jurado: Dr. Andrés Diaz-Pace, Dr. Hugo Andrés Lopez y Dr. Matías Urbieta
Director: Dr. Sebastián Uchitel
Consejero: Dr. Nicolás D'Ippolito

**Universidad de Buenos Aires**
*Facultad de Ciencias Exactas y Naturales*
Departamento de Computación
Intendente Güiraldes 2160
C1428EGA – Buenos Aires, Argentina

# Actualización dinámica de controladores de eventos discretos

## Resumen

Los sistemas de eventos discretos están en el corazón de muchos sistemas de software que requieren operación continua como los sistemas reactivos. Cambiar estos controladores en tiempo de ejecución, para dar soporte a cambios del ambiente o cambios en los requerimientos, es una problema desafiante y no resuelto hasta ahora. En esta tesis, se plantea formalmente el problema de actualizar dinámicamente sistemas de eventos discretos que controlan sistemas reactivos. Presento aquí un enfoque general para especificar criterios de correctitud para actualizaciones dinámicas y una técnica que computa automáticamente un controlador que maneja la transición desde la vieja especificación hasta la nueva especificación, garantizando que el sistema alcance un estado en el cual esa transición pueda ocurrir correctamente y en la cual la arquitectura del sistema subyacente pueda reconfigurarse. La solución usa síntesis de controladores de eventos discretos para construir automáticamente un controlador que garantiza ambas: progreso sobre la actualización y actualizaciones seguras.

La técnica desarrollada fue aplicada a distintos dominios como sistemas reactivos o sistemas robóticos. Cada uno de ellos comprende distintos desafios entre los cuales se destaca la urgencia por adaptarse a los nuevos requerimientos y las variedades de estrategias que se pueden computar para lograr la actualización.

Otro dominio de aplicación de la técnica fue la reconfiguración de procesos de negocio. Como es esperado, las organizaciones requieren que sus procesos de negocios evolucionen manteniendo el cumplimiento de nuevas políticas, estrategias y regulaciones. La reconfiguración de un proceso de negocio es un problema desafiante ya que no solo se debe idear un nuevo workflow, sino que también, requiere de entender de cómo debe ser la transición entre el viejo workflow y el nuevo. Si bien los procesos de negocio suelen ser más lentos que los sistemas reactivos o los sistemas robóticos, este problema solo fue levemente estudiado, sin poder garantizar un proceso automático que lo resuelva. En esta tesis producimos procesos de reconfiguración que garantiza la evolución de un antiguo workflow a uno nuevo, satisfaciendo los requerimientos de transición definidos por el usuario.

***Palabras clave: Actualización dinámica. Sistemas Reactivos. Procesos de Negocio. Sistemas de Eventos Discretos. Síntesis.***

# Dynamic Update of Discrete Event Controllers

## Abstract

Discrete event controllers are at the heart of many software systems that require continuous operation such as reactive systems. Changing these controllers at runtime to cope with environment or system requirements change is a challenging open problem. In this paper we address the problem of dynamic update of controllers in reactive systems. We present a general approach to specifying correctness criteria for dynamic update and a technique for automatically computing a controller that handles the transition from the old to the new specification, assuring that the system will reach a state in which such a transition can correctly occur and in which the underlying system architecture can reconfigure. Our solution uses discrete event controller synthesis to automatically build a controller that guarantees both progress towards update and safe update.

The developed technique was applied to different domains as reactive systems or robotic systems. Each of them comprises different challenges, among them the urgency to adapt to the new requirements and varieties of strategies that can be computed to achieve the update.

Another domain of application of the technique was the reconfiguration of business processes. As expected, organizations require that their business processes evolve while maintaining compliance with new policies, strategies and regulations. The reconfiguration of a business process is a challenging problem since not only a new workflow must be devised, but it also requires to understand how the transition between the old workflow and the new one should be. While business processes are usually slower than reactive systems or robotic systems, this problem was only slightly studied, without being able to guarantee an automatic process that solves it. In this thesis we produce a reconfiguration of a business process that guarantees the evolution of an old workflow to a new one, satisfying the transition requirements defined by the user.

***Keywords: Dynamic Update. Reactive Systems. Business Processes. Discrete Event Controllers. Synthesis.***

# Agradecimientos

Cuando un doctor explica lo que significa hacer un doctorado, le intenta poner palabras a un período de tiempo, el cual, no es explicable. Al menos no lo es con simples palabras. Tan así de inexplicable es el doctorado, como mi sentimiento de gratitud hacia las personas que siguen a continuación.

En primer lugar, quiero agradecer a Sebastián Uchitel. Él me ha guiado por este camino, mostrándome que era posible y facilitándome todo lo necesario para alcanzar los objetivos. Sebastián es un ejemplo de director (y de persona) que logró explotar al máximo mis habilidades con las que inicié el doctorado y pudo descubrir mis puntos más flojos para mejorarlos. Sin él, todo esto hubiese sido imposible, y por todos estos puntos que remarco, me genera una profunda admiración.

El trabajo aquí realizado es también esfuerzo de otros investigadores a los cuales valoro. Empezando por Nicolás D'ippolito quien me ha guiado en los primeros pasos en el mundo académico, y supo acompañarme muy bien, sobretodo, cuando todo estuvo muy tenebroso. Su forma de ser tan simple y divertida hacen que el día laboral sea mucho más ameno. Luego, Víctor Braberman y Jeff Kramer me han ayudado a mejorar mis trabajos con sus intuiciones. Hasta el día de hoy, sigo impresionado por la manera en que Víctor ve las fortalezas y las debilidades de una idea.

No hubiese podido hacer un doctorado sin el amor infinito que recibo de mis padres. Tanto Daniel como Hebe me dieron la vida y una gran parte de las suyas para sobrepasar cualquier obstáculo que surja. Ellos son para mi una pieza indispensable y hoy soy la persona que soy gracias a ellos. Agradezco también a mi hermano Luciano, a mi cuñada Alejandra y a mi sobrino Nachito, que fueron una gran compañía de mi doctorado. Ellos, junto con Claudia, Ariel, Oli, Bruno, Silvia y Franco, son mi soporte, mi guía y mi cable a tierra.

No quiero dejar de agradecer a todos mis amigos: empezando por el grupo de LaFHIS y destacando a Christian con quien empezamos juntos el doctorado. También a Fer, "El tano", Lucas, Ger, Pablito, Lea, Pablo, Ale y Ricky. Todos ellos logran que cada día de mi vida tenga una razón para reír.

Finalmente, quiero agradecerles a mis abuelos que no pudieron estar conmigo físicamente en esta etapa, pero me han dejado valiosos recuerdos que llevaré en cada etapa de mi vida. Beba y Héctor los llevo en mi corazón.

# Índice general


**1 Introducción** **1**
- 1.1 Motivación . . . 1
- 1.2 Resumen de las contribuciones . . . 6
- 1.3 Estructura de la tesis . . . 9

**2 Definiciones Preliminares** **11**
- 2.1 Sistemas de Transiciones Etiquetados . . . 11
- 2.2 Lógica Lineal Temporal basada en Fluents . . . 16
- 2.3 Problema de Control de Sistemas de Transiciones Etiquetados . . . 19
- 2.4 Grafos Dinámicos de Condición y Respuesta . . . 21

**3 Actualización Dinámica de Controladores de Eventos Discretos** **25**
- 3.1 Ejemplo Motivador: Cadena de Montaje . . . 25
- 3.2 Formalización del problema . . . 31
  - 3.2.1 Hotswap de Controladores . . . 31
  - 3.2.2 Controlando la Reconfiguración del Ambiente . . . 33
  - 3.2.3 Criterio de Correctitud de la Actualización de Controladores . 36
- 3.3 Propuesta de Solución . . . 40
  - 3.3.1 Modelo del Ambiente . . . 40
  - 3.3.2 Resolviendo la Síntesis de ADC con control de LTS . . . 43
  - 3.3.3 Complejidad . . . 46
  - 3.3.4 Correctitud y Completitud . . . 46

**4 Soporte de Herramienta** **55**
- 4.1 Sintetizando un Controlador . . . 55
  - 4.1.1 Modelando el Ambiente . . . 56
  - 4.1.2 Modelando los Requerimientos del Controlador . . . 57
  - 4.1.3 Ejecutando la Síntesis . . . 58
- 4.2 Sintetizando un Controlador Actualizador . . . 59
  - 4.2.1 Modelando un Problema de Actualización Dinámica . . . 59
  - 4.2.2 Solución al Problema de Actualización Dinámica . . . 60

**5 Aplicación a Orquestaciones de Software** **63**
- 5.1 Cadena de Montaje . . . 65
- 5.2 Planta de Energía . . . 68
- 5.3 RailCab . . . 69
- 5.4 Protocolo GSM-oriented . . . 72
- 5.5 MetaSocket . . . 72
- 5.6 Surveillance . . . 73
- 5.7 Tasa de Consumo de Energía Inesperada . . . 74
- 5.8 Monitoreo de Incendios . . . 75

5.9 Resumen de las Experiencias . . . . . 76

**6 Aplicación a Procesos de Negocio 79**
6.1 Ejemplo Motivador: Hospital Oncológico . . . . . 80
6.2 Semántica de DCR como Problema de Control . . . . . 83
6.2.1 Eventos Controlables y Monitoreables . . . . . 84
6.2.2 Modelo del Ambiente . . . . . 85
6.2.3 Objetivos del Controlador . . . . . 86
6.2.4 Síntesis de Workflows . . . . . 88
6.2.5 Correctitud y Maximalidad de la Traducción . . . . . 89
6.3 Controlando la Reconfiguración de Procesos de Negocio . . . . . 95
6.3.1 La Especificación de un Requerimiento de Transición . . . . . 95
6.3.2 Reconfiguración de Workflows . . . . . 96
6.3.3 Construcción Automática del Workflow de Reconfiguración . . 98
6.3.4 Reconfiguración de DCR graphs . . . . . 100
6.3.5 Construyendo una Reconfiguración de DCR . . . . . 101
6.3.6 Correctitud del Algoritmo . . . . . 104
6.4 Casos de Estudio de Procesos de Negocio . . . . . 106
6.4.1 Hospital Oncológico . . . . . 106
6.4.2 Proceso de evaluación de Doctores . . . . . 108
6.4.3 Proceso de Seguro . . . . . 109
6.4.4 Proceso de Reparación de Computadoras . . . . . 109
6.4.5 Procesos de Negocio: Resumen de las Experiencias . . . . . 110

**7 Discusión y Trabajo Relacionado 113**
7.1 Controladores de Eventos Discretos . . . . . 113
7.1.1 Propiedades de Transición Especificadas por el Usuario . . . . 114
7.1.2 Computación Automática de Estrategias de Actualización . . . 115
7.1.3 Sistemas de Eventos Discretos . . . . . 117
7.1.4 Reconfiguración de Arquitecturas de Software . . . . . 118
7.1.5 Síntesis . . . . . 118
7.2 Procesos de Negocio . . . . . 121

**8 Conclusiones 123**
8.1 Contribuciones . . . . . 123
8.2 Limitaciones . . . . . 124
8.3 Trabajo Futuro . . . . . 125

**Bibliografía 127**

# 1 Introducción

> *The beginnings will not be easy; they shall be extremely difficult.*
>
> — **Ernesto "Che" Guevara**
> (Revolucionario argentino)

## 1.1 Motivación

La actualización dinámica de un software es una característica fundamental de sistemas que necesitan estar en ejecución continuamente. Este requerimiento es muy común en muchos dominios como por ejemplo, sistemas de seguridad, sistemas de negocios críticos, entre otros. Cuando pensamos en ejecución continua, estamos de algún modo considerando que los sistemas puedan ser actualizados en tiempo de ejecución para poder adaptarse a cambios en su ambiente de ejecución y en los requerimientos que se espera que los sistemas garanticen. Producir técnicas novedosas para actualizar de manera correcta, sin parar o interrumpir la ejecución es una tarea desafiante y fue bastante estudiado bajo distintos escenarios, incluyendo actualización y reconfiguración dinámica de componentes (e.j. [KM90]), reconfiguración de controladores (e.j. [GR96; NS15; Sam+08]) y muchos más recientes en diseño de sistemas adaptativos [Sea].

¿Cuando es seguro cambiar un sistema que esta ejecutando? Un concepto clave a la hora de actualizar un sistema en ejecución es poder garantizar que la ejecución actual no se vea alterada. Por ejemplo, un criterio para la reconfiguración arquitectónica es poder garantizar que no haya componentes involucrados en las transacciones activas, formalizado inicialmente como "quiescence" [KM90] y más tarde "tranquility" [Van+07]. Luego se han estudiado condiciones relacionadas (incluyendo [AR09; Gup+96]) para cambios en tiempo de ejecución. Sin embargo,

la mayoría de estos trabajos no considera el caso en el que una actualización es requerida debido a un cambio de especificación [BG10], ni tampoco analizan si la actualización es correcta, o no, con respecto a esa especificación, ni mucho menos **cómo** el sistema debería comportarse durante la actualización.

Consideremos el caso de los sistemas orquestados. Estos tienen un componente controlador que ejecuta una estrategia (abstractamente representada por una máquina de estados) que coordina múltiples componentes y servicios para garantizar objetivos de una misión [KGP08]. Si un controlador de estos sistemas necesita ser cambiado en tiempo de ejecución para adaptarse a un cambio de especificación, nos preguntaríamos cuando hacemos el "hotswap" del controlador actual por uno nuevo que garantice la nueva especificación. Llamamos "hotswap" al intercambio de controladores mientras la ejecución sigue su curso. Zhang y Cheng [ZC05] argumentan que las condiciones de actualización *deben ser explícitamente capturadas a nivel de requerimientos* y que las restricciones al momento de cambiar un controlador pueden variar dependiendo del dominio y del sistema en particular. En situaciones de emergencia se requieren actualizaciones que deben ser ejecutadas tan pronto como se puede con muy pocas restricciones que demoren la actualización.

Para actualizaciones planeadas de, por ejemplo, una cadena de montaje automatizada, se puede requerir que las tareas pendientes sean completadas antes del cambio a la nueva especificación. A su vez, cabe destacar, que los sistemas que pueden ser considerados como sistemas orquestados son muy variados. Entre ellos sistemas reactivos, sistemas orientados a servicios y otros más.

De manera similar, los procesos de negocio también apuntan a garantizar ejecución continua. Estos, a diferencia de los sistemas orquestados, apuntan a organizar la fuerza de trabajo de un equipo o de individuos teniendo un impacto final cuantificable (e.g. el trabajo total finaliza más rápido, menor cantidad de horas hombre desperdiciada, etc). Estos sistemas tienen una representación de ejecución definida por un proceso que cumple con los objetivos de la organización (i.e. el workflow). Estos workflows sirven a organizaciones donde las reglas de negocios evolucionan de ma-

nera desconocida desde el momento en el cual el workflow se diseñó (e.g., [VAS00]). Las organizaciones diseñan estos procesos de negocio en un momento determinado, llamado tiempo de diseño, con la premisa de que van a ser ejecutados por un tiempo prolongado, y luego, cuando estas organizaciones evolucionan, sus workflows deben poder garantizar nuevos requerimientos [AH04]. Hay una necesidad, entonces, de modificar en tiempo de ejecución un workflow para permitir cambios en objetivos de negocio, cambios en las interfaces con otros sistemas, incrementos de procesos regulatorios, debilitar suposiciones de actividades potencialmente fraudulentas, entre otras cosas. Por lo tanto, cuando hablemos de la *reconfiguración de procesos de negocio* nos vamos a referir no solo a la creación de un nuevo workflow, sino que también al cambio dinámico desde el viejo workflow al nuevo. Tanto la reconfiguración como el nuevo proceso de negocio no va a ser definido de manera *imperativa*. Es decir, no vamos a definir por extensión cómo actualizar cada caso, sino que daremos un conjunto de reglas *declarativas* cuyo modelo que las satisface es uno que produce la reconfiguración deseada.

En los últimos años, hubo avances en el entendimiento de tipos de propiedades que una actualización dinámica o una reconfiguración de procesos de negocio podría necesitar satisfacer, pero los esfuerzos siguen siendo acotados. Por ejemplo, se han propuesto técnicas de verificación para proveer garantías a estrategias manuales de actualización dinámica, pero, recientemente solo se ha estudiado la síntesis automática de estrategias de actualización de manera limitada [Ghe+12; Gre+13; ZC05; ZC06]. Ya sea sin producir enfoques automáticos o si son automáticos no proveen suficientes garantías.

Las técnicas existentes para actualizar dinámicamente un proceso suponen que este irá ejecutándose hasta eventualmente alcanzar un estado en el cual es seguro realizar una actualización. Por ejemplo, en [KM90], se espera que los componentes en un sistema distribuido hayan sido diseñados para aceptar mensajes de *passivate* y también comunican cuando pasan a estar en estado *quiescent*. Sin embargo, no hay garantías de que el estado "quiescence" sea alcanzado. En [Ghe+12] una

suposición subyacente es que el controlador a ser actualizado regularmente vuelve a su estado inicial, el cual es seguro para actualizar el controlador. Suponer que el sistema naturalmente alcanza un estado seguro representa una suposición muy fuerte que podría no ser cierto en muchos dominios. Además, en muchas situaciones no es deseable tener que esperar a las instancias en ejecución que alcancen un estado seguro. De hecho, Zhang and Cheng [ZC05] reconocen que un sistema puede necesitar ser guiado activamente a un estado seguro actualizable; sin embargo en su estudio, es el usuario el que produce la estrategia para hacer esto. El hecho de que el ambiente del sistema puede no ser cooperativo, complica la tarea de guiar al sistema a un estado seguro.

En cuanto a la reconfiguración de procesos de negocio también se han estudiado diversas alternativas. La pregunta a resolver dentro de este dominio es cómo reconfigurar las instancias vivas de un proceso de negocio. Estas, son ejecuciones de un proceso de negocio que están en mitad de la ejecución al momento de solicitarse la reconfiguración. Por ejemplo, [Ell+95] discute reconfiguración "inmediata" que asegura que la reconfiguración debe ocurrir tan pronto como sea posible pero solo en los estados en los cuales el nuevo workflow tenga un comportamiento consistente con el viejo. Las reconfiguraciones "postergadas" consideran que las instancias vivas deben terminar usando el viejo workflow, mientras que las nuevas instancias son creadas usando el nuevo.

En algunos casos, se solicita *requerimientos de transición específicos de dominio*. Por ejemplo, la reconfiguración puede ser requerida tan pronto como sea posible pero se requiere que para algunas instancias vivas en particular se aplique un tratamiento excepcional, que puede incluir la repetición o la reversión de una actividad.

De hecho, la reconfiguración de procesos de negocio pueden ser extremadamente desafiantes y podría ser muy beneficioso tener técnicas automáticas que I) analicen requerimientos de procesos de negocio y requerimientos de transición, y II) construyan workflows y estrategias de reconfiguración que cumplan esos requerimientos.

Al igual que para actualización de software orquestado, un enfoque posible para la automatización de la reconfiguración de procesos de negocio es *construir y verificar*, en la cual se usan técnicas de verificación formal para obtener una base sólida para el análisis de workflow. Estas técnicas pueden usarse para garantizar el cumplimiento de los requerimientos del workflow. Sin embargo, este enfoque también requiere una construcción de un workflow primero para que luego sea verificado, y, en caso de tener que modificarlo, es necesario una re-verificación. Un enfoque alternativo es producirlos automáticamente (workflows *correctos-por-construcción*) a partir de los requerimientos, y, en caso de necesitar una reconfiguración, deberíamos seguir la misma estrategia.

Aunque la construcción automática de workflows a partir de los requerimientos ha sido estudiado (e.g., [PA06; HM11]), la *síntesis de estrategias de reconfiguración* para requerimientos de transición de dominio especifico definido por el usuario no ha recibido atención.

Finalmente, y a modo de resumen, para que el problema de *actualización dinámica de controladores (ADC)* o el problema de *reconfiguración de un proceso de negocio* soporte cambios de especificación, ya sea de los requerimientos y/o del ambiente, debe ser planteado como una construcción automática de un *controlador actualizador* que sabe realizar el hotswap del proceso actual (que garantiza la vieja especificación) por uno nuevo (que va a garantizar la nueva especificación) mientras, a su vez, satisface cualquier requerimiento adicional que reglamenta cuando un hotswap puede ocurrir y cuando la arquitectura puede ser reconfigurada. El controlador actualizador debe primero tomar control del sistema. Luego, mientras satisface la vieja especificación, debe guiar al sistema (garantizando progreso) hacia un estado en el cual la reconfiguración de la arquitectura y el cambio de la vieja especificación a la nueva puedan ocurrir. Al finalizar la actualización o reconfiguración de un proceso de negocio, este proceso actualizador debe operar garantizando la nueva especificación.

A modo de aclaración destacamos la diferencia entre reconfiguración de la arquitectura vs reconfiguración de procesos de negocio. La primera se refiere a un paso particular de la actualización dinámica de controladores en donde el controlador actualizador determina el momento en el que la reconfiguración de la arquitectura puede llevarse a cabo (e.g. enlazado de componentes), mientras que, la reconfiguración de procesos de negocio es el término utilizado en la literatura para referirse a la actualización de procesos de negocio.

## 1.2 Resumen de las contribuciones

Las contribuciones presentadas en esta tesis puede clasificarse en dos tipos distintos: contribuciones teóricas, generadas por la elaboración de una técnica novedosa, y contribuciones de aplicación, que se generan al aplicar la técnica elaborada en un dominio particular. Habiendo diferenciado ambas, pasaremos a explicar cada una de ellas.

En cuanto las contribuciones teóricas, la primera a destacar es la definición de un criterio de correctitud para actualizaciones dinámicas de procesos en general. Estos procesos pueden comprender procesos de software donde una pieza particular es la encargada de orquestar el sistema, o también, procesos sociales donde se define un orden particular para ejecutar ciertas actividades, el cual debe ser respetado por los distintos grupos de trabajo. A su vez, se presenta una solución posible al problema de actualizar dinámicamente procesos para satisfacer este criterio de correctitud. La solución propuesta se basa en síntesis de controladores de eventos discretos (e.g. [RW89; Pit+06; D'I+13]), ya que por su naturaleza nos garantiza una solución automática y correcta con respecto a los cambios de requerimientos de supuestos del ambiente. Una característica esencial de esta solución es que fue ideada lo suficientemente abstracta como para poder ser aplicada para cualquier proceso que pueda modelarse como un controlador de eventos discretos.

Las contribuciones principales que posee esta técnica abstracta pueden enumerarse de la siguiente manera:

1. Mostramos como los requerimientos para describir la transición entre las especificaciones pueden ser descritos restringiendo cuando el cambio puede ocurrir y cuando una reconfiguración debería ocurrir;
2. Formalizamos el comportamiento de un proceso en donde un controlador es intercambiado por otro que sabe tomar acción ni bien es instalado, guía la ejecución hacia un estado donde es seguro realizar el cambio de especificación y realiza la reconfiguración de su ambiente de ejecución.
3. Definimos un criterio de correctitud para la actualización dinámica de controladores (ADC).
4. Definimos un algoritmo basado en teoría de control de eventos discretos (e.g. [RW89; Pit+06; D'I+13]) que construye el controlador cuyo comportamiento es el descrito en el punto 2.
5. Probamos que el controlador construido automáticamente por el algoritmo del punto 2 es correcto y completo con respecto al criterio de correctitud del punto 3.

Por otro lado, pudimos generar contribuciones en diversos dominios de aplicación. La primer idea fue aplicar la técnica a software orquestado, ya que la lógica del orquestador puede ajustarse muy fácilmente a un controlador de eventos discretos. Otra aplicación exitosa es la aplicación a procesos de negocio. Estos últimos no son fácilmente modelados como controlador de eventos discretos pero pudimos desarrollar una traducción que une las dos áreas.

Una de las contribuciones más importantes que vale la pena mencionar, es que tanto como para la *Actualización Dinamica de Software Orquestado* como para la *Reconfiguración de Procesos de Negocio*, la técnica presentada puede ser utilizada más de una vez si así fuese necesario. Suponiendo que el diseño del proceso original fue malo (i.e., no satisface los requerimientos necesario o satisface ciertos requerimientos que fueron mal elicitados) y fue cambiado por otro, podemos incluso cambiar este

último por una especificación aún mejor. Este escenario puede suceder en dominios de ejecución muy cambiantes como por ejemplo controladores que toman la decisión de comprar o vender activos en mercados financieros, donde las valuaciones de estos varían con mucha frecuencia.

En cuanto a la aplicación de la técnica a procesos de negocio tenemos algunos beneficios que ya describí anteriormente (e.g. produjimos workflows que son correctos-por-construcción, computamos una estrategia que reconfigura un proceso garantizando progreso desde el viejo workflow, etc). Sin embargo, como enunciamos anteriormente, para lograr estos beneficios, debemos realizar una traducción. En la Figura 1.1, mostramos el esquema de reconfiguración que seguiremos para poder adaptar la técnica abstracta a procesos de negocio. Inicialmente, traducimos Diagramas Dinámicos de Condición y Respuesta (DCR graphs [HM11]), un lenguaje declarativo para definir requerimientos de procesos de negocio, a un formalismo basado en Sistema de Transición Etiquetado y Lógica Lineal Temporal [Pnu77]. Vamos a utilizar esta traducción tanto para los requerimientos de DCR viejos como para los nuevos, generando así dos problema de control [D'I+13]. Con estos dos problemas de control más un requerimiento de transición propuesto por el usuario, podemos definir un problema de actualización de controladores [Nah+18] que cuando es resuelto brinda una estrategia de reconfiguración representada con un Sistema de Transiciones Etiquetadas (Definición 2.1.1). Esta estrategia es capaz de reconfigurar cualquier instancia viva del workflow actual desde cualquier estado. El esquema de reconfiguración dinámica finaliza con un proceso de extracción en el cual la historia de una instancia viva del workflow actual es usada para computar, a partir de la estrategia de reconfiguración, un DCR graph cuya semántica representa cómo debe reconfigurarse esa instancia viva particular para que cumpla con los requerimientos de transición y alcance las reglas de negocio nuevas.

Pudimos aplicar este esquema a ejemplos extraídos de BPM Academic Initiative [Bpma] que están modelados como DCR Graphs en la herramienta [Mar+16] y los reconfiguramos alternando varios requerimientos de transición.

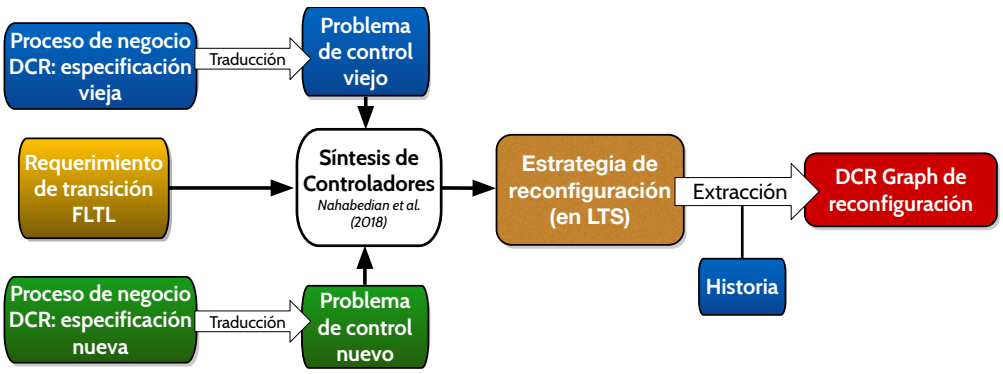


**Fig. 1.1:** Esquema de la reconfiguración de procesos de negocio.

Todas las contribuciones detalladas anteriormente fueron introducidas en publicaciones con referato, ya sea en conferencias o en revistas. La primera de ellas [Nah+16] en la conferencia SEAMS (*International Symposium on Software Engineering for Adaptive and Self-Managing Systems* [Sea]), por la cual, hemos sido galardonados con el **"Best Paper Award"**. Luego, una publicación [Nah17] aceptada en el Doctoral Symposium de la conferencia ICSE (*International Conference on Software Engineering* [Ics]). Posteriormente, hemos realizado una extensión a nuestro primer trabajo, realizando un cambio en el formalismo para obtener mejoras de performance. Esta publicación [Nah+18] también contiene la prueba de correctitud y completitud de la técnica y una sección más detallada de validación experimental. La extensión fue publicada en la revista *Transaction on Software Engineering [Tse]*. Por último, toda la aplicación de la técnica al mundo de Business Process Management fue publicada [Nah+19] en la conferencia *International Conference on Business Process Management [Bpmb]*.

## 1.3 Estructura de la tesis

El resto de la tesis esta estructurada de la siguiente manera:

En el Capítulo 2, introduciré todas las definiciones teóricas necesarias para poder expresar formalmente el problema a resolver y contar la solución propuesta basada en síntesis de controladores discretos y Dynamic Conditions Response Graphs.

Más adelante, el Capítulo 3 está compuesto por tres secciones esenciales de la tesis. La primera presenta un ejemplo motivador para guiar al lector a entender que tipo de problemas son los que solucionamos. Luego, la segunda plantea el problema de actualización dinámica de controladores, llegando a definir un criterio de correctitud que toda solución debería alcanzar para que sea considerada correcta. Finalmente, la tercera, propone una solución posible al problema de actualización dinámica de controladores que está basada en síntesis de controladores discretos. Dicha solución satisface el criterio de correctitud introducido previamente, afirmación que pruebo formalmente.

El Capítulo 4 se encarga de mostrar el uso de la herramienta que produce los valores de salida que se encuentran en el Capítulo 3. De esta manera, podemos decir que es un capítulo clave para poder replicar los resultados o para producir nuevos ejemplos.

Este documento continúa con las aplicaciones de la técnica propuestas. En primera instancia, el Capítulo 5, detalla varios casos de estudio del tipo de software orquestado con su respectivo análisis. Luego, el Capítulo 6, se concentra en explicar cómo aplicar la técnica del Capítulo 3 para resolver el problema de reconfiguración de procesos de negocio. Para esto, es necesario explicar como traducir un proceso de negocio a un problema de control, para luego poder utilizar dos de estas traducciones (una para los requerimientos viejos y otra para las nuevos) como valores de entrada del problema descrito en la Sección 3.3. También se explica como extraer desde ese output un DCR de reconfiguración cuyas trazas extienden una traza de una instancia viva en el viejo workflow.

Finalmente esta tesis termina con dos capítulos tradicionales. El primero de ellos es el capítulo de discusión (Capítulo 7) donde se realiza un trabajo comparativo con otras técnicas elaboradas anteriormente. Aquí no solo se explican las ventajas y desventajas, sino que también detallamos el estado del arte sobre el cual se apoya esta tesis. El último capítulo establece las conclusiones del trabajo, expresando contribuciones, limitaciones y trabajo futuro (Capítulo 8).

# Definiciones Preliminares

# 2

> *If the facts do not fit the theory, change the facts.*
>
> — **Albert Einstein**
> (Físico)

En este capitulo presentamos la teoría de fondo necesaria para poder formalizar el problema de actualización dinámica de controladores (ADC). Además, introducimos todas las definiciones necesarias para realizar las aplicaciones tanto en software orquestado como en procesos de negocio. En particular, es necesario introducir definiciones para explicar la traducción de procesos de negocio a problemas de control. Así mismo, también muestro las pruebas formales de las propiedades presentadas.

## 2.1 Sistemas de Transiciones Etiquetados

Los Sistemas de Transiciones Etiquetadas o LTS por su sigla en ingles (Labelled Transition Systems) son una representación canónica y composicional de sistemas reactivos. Como su nombre lo indica, estos modelos están compuestos por transiciones etiquetadas y estados, que en conjunto denotan un conjunto de trazas posibles. Vamos a utilizar este formalismo mayoritariamente para representar controladores y ambientes. Las trazas del ambiente representa todos los posibles comportamientos del mundo, representando con cada etiqueta, un evento distinto. Similarmente, podemos decir que las trazas de un controlador son las ejecuciones de la estrategia del controlador.

**Definición 2.1.1.** (Sistema de Transiciones Etiquetadas) *Un* Sistema de Transiciones Etiquetadas *(LTS)* $E$ *es una tupla* $(S_E, A_E, \Delta_E, e_0)$*, donde* $S_E$ *es un conjunto finito de estados,* $A_E \subseteq$ Act *es el* alfabeto de comunicación*,* Act *es el universo de todos los*

*eventos,* $\Delta_E \subseteq (S_E \times A_E \times S_E)$ *es una relación de transición, y,* $s_0 \in S_E$ *es el estado inicial.*

**Notación 2.1.1.** *Sea* $E$ *un LTS, para un estado* $e \in S_E$*, notaré* $\Delta_E(e) = \{\ell \mid (e, \ell, e') \in \Delta_E\}$.

**Notación 2.1.2.** *Sea* $E$ *un LTS y* $e \in S_E$ *un estado de* $E$*, notaré el cambio del estado inicial de* $E$ *a* $e$ *como* $E(e)$.

Dos características de interés de los LTS es si son determinísticos o si son libre de deadlock (deadlock-free). Para lo primero necesitamos que para todo estado no haya dos transiciones de salida con la misma etiqueta que vayan a estados distintos. La segunda propiedad se cumple cuando no existe un estado sin transiciones de salida. Formalmente:

**Definición 2.1.2.** (Determinismo) *Sea* $E$ *un LTS,* $E$ *es* ***deterministico*** *si* $(e, \ell, e') \in \Delta_E$ *y* $(e, \ell, e'') \in \Delta_E$*, entonces,* $e' = e''$

**Definición 2.1.3.** (Deadlock-freedom) *Sea* $E$ *un LTS,* $E$ *es* ***deadlock-free*** *si para todo* $e \in S \cdot \exists\ (e, \ell, e') \in \Delta_E$.

**Notación 2.1.3.** *Sea* $E$ *un LTS,* $\ell$ *una etiqueta en el alfabeto de* $E$*, denotamos por* $e \xrightarrow[E]{\ell} \dot{e}$ *a la transición* $(e, \ell, \dot{e}) \in \Delta_E$.

Una ejecución de un controlador puede verse como la secuencia de eventos de un camino del LTS.

**Definición 2.1.4.** (Traza) *Una* ***traza*** *de un LTS* $E$ *es una secuencia de etiquetas* $\pi = \ell_0, \ell_1, \ldots$ *de las cuales existe una secuencia de estados* $s_0, s_1, \ldots$ *tal que* $s_0$ *es el estado inicial de* $E$ *y* $\forall i \geq 0 \cdot \ell_i \in \Delta_E(s_i)$.

**Notación 2.1.4.** *Sea* $E$ *un LTS y* $\pi$ *una traza de* $E$*. Notaré* $\ell \in \pi$ *a una etiqueta en la traza* $\pi$.

A continuación presentaremos diferentes relaciones de equivalencias entre LTS. El isomorfismo, que es la relación más fuerte, exige una equivalencia total tanto de

estados como de transiciones. Mientras que, la bisimulación solo exige que ambos LTS puedan simularse (i.e. copiar la ejecución) entre si.

**Definición 2.1.5.** (Isomorfismo) *Sea* $E = (S_E, A, \Delta_E, s_E)$ *y* $M = (S_M, A, \Delta_M, s_M)$ *LTS. Un isomorfismo es una bijección* $f : S_E \to S_M$ *que preserva transiciones:*

$$(q, \ell, q') \in \Delta_E \quad \textit{si y solo si} \quad (f(q), \ell, f(q')) \in \Delta_M$$

*Para todo* $q, q' \in S_E$. *Si existe un isomorfismo entre* $E$ *y* $M$, *entonces* $E$ *y* $M$ *son isomorficos, denotado* $E = M$.

**Definición 2.1.6.** (Bisimulación) *Sea* $\mathcal{P}$ *el universo de todos los LTS. Una relación binaria* $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ *es una bisimulación si y solo si, siempre que* $(P, Q) \in \mathcal{R}$ *entonces para cada* $a \in Act$ *lo siguiente vale:*

- *si* $(P \xrightarrow{a} P')$, *entonces,* $(\exists Q' \cdot Q \xrightarrow{a} Q' \wedge (P', Q') \in \mathcal{R})$
- *si* $(Q \xrightarrow{a} Q')$, *entonces,* $(\exists P' \cdot P \xrightarrow{a} P' \wedge (P', Q') \in \mathcal{R})$

*donde* $X \xrightarrow{x} X'$ *denota que el LTS* $X$ *puede ejecutar* $x$ *alcanzando al estado* $s$ *y* $X' = X(s)$. *Denoto por* $P \sim Q$ *que* $P$ *y* $Q$ *son bisimilares.*

**Propiedad 2.1.1.** *Sea* $P$ *y* $Q$ *LTS tales que* $P = Q$, *entonces,* $P \sim Q$.

*Demostración.* Queremos ver que existe una relación binaria $\mathcal{R}$ de bisimulación entre $P$ y $Q$. Como $P = Q$ entonces existe una función $f$ tal que si $(q, \ell, q') \in \Delta_P$ entonces $(f(q), \ell, f(q')) \in \Delta_Q$. Luego, para cada $a \in Act$ vale que:

- si $(P \xrightarrow{a} P')$, entonces, $f(P) \xrightarrow{a} f(P') \wedge (P', f(P')) \in \mathcal{R}$
- si $(f(P) \xrightarrow{a} f(P'))$, entonces, $(P \xrightarrow{a} P') \wedge (f(P'), P') \in \mathcal{R}$

donde siendo $P$ un LTS, $f(P)$ es el LTS que resulta de aplicar el isomorfismo al LTS $P$. Luego $P$ y $Q$ son bisimilares. □

Los sistemas reactivos se construyen composicionalmente. Esta composición es modelada usualmente con LTS como el producto cartesiano de los estados de los LTS componentes donde la comunicación entre los componentes es modelada como una comunicación sincrónica en los eventos compartidos y proposiciones. Formalmente:

**Definición 2.1.7.** (Composición Paralela) *La composición paralela* $E\|C$ *de dos LTS* $E = (S_E, A_E, \Delta_E, e_0)$ *y* $C = (S_C, A_C, \Delta_C, c_0)$ *es un LTS* $(S_E \times S_C, A_E \cup A_C, \Delta_\|, (e_0, c_0))$ *tal que* $\Delta_\|$ *es la relación más chica que satisface las siguiente reglas:*

$$\frac{(e, \ell, e') \in \Delta_E}{((e, c), \ell, (e', c)) \in \Delta_\|} \ell \notin A_C$$

$$\frac{(c, \ell, c') \in \Delta_C}{((e, c), \ell, (e, c')) \in \Delta_\|} \ell \notin A_E$$

$$\frac{(e, \ell, e') \in \Delta_E,\ (c, \ell, c') \in \Delta_C}{((e, c), \ell, (e', c')) \in \Delta_\|} \ell \in A_E \cap A_C$$

La operación de reetiquetado define un nuevo LTS a través de reetiquetar o borrar transiciones de otro LTS. Sucede que una transición es borrada cuando su etiqueta no está definida en la función de reetiquetado, o la transición cambia de etiqueta cuando su etiqueta está definida en la función de reetiquetado. Nótese que la función de reetiquetado identidad ($ID$) nos devuelve el mismo LTS.

**Definición 2.1.8.** (Operador de Reetiquetado) *Sea* $E = (S_E, A_E, \Delta_E, e_0)$ *un LTS y* $f : A_E \to A_f \subseteq$ Act *una función injectiva parcial. La operación de reetiquetado* $[E]_f$ *es un LTS* $(S_E, A_f, \Delta_f, e_0)$*, donde* $\Delta_f$ *es la relación mas chica que satisface la siguiente regla:*

$$\frac{(e, \ell, e') \in \Delta_E}{(e, f(\ell), e') \in \Delta_f}$$

Notar que como $f$ es una función injectiva, entonces, no es posible generar un LTS no-deterministico a partir de la operación de reetiquetado.

**Propiedad 2.1.2.** *Sea $E, B$ LTS deterministicos, y $f : Act \to Act$ es una función parcial injectiva, entonces, la siguiente equivalencia vale:* $[E\|B]_f \sim [E]_f \| [B]_f$

*Demostración.* Voy a probar que $E$ y $B$ son isomorfos y por lo tanto son bisimilares (Propiedad 2.1.1). Para todo par de estados $(e, b) \in E\|B$ vale lo siguiente, separando en los casos A) cuando la etiqueta $\ell \in E$ pero no en $B$, B) cuando $\ell \in B$ pero no en $E$ y en el caso C) cuando $\ell \in E$ y $\ell \in B$:

A) $(e,b) \xrightarrow[[E\|B]_f]{f(\ell)} (e',b)$ sii $(e,b) \xrightarrow[E\|B]{\ell} (e',b)$ sii $e \xrightarrow[E]{\ell} e'$ y $\ell \notin B$ sii
$e \xrightarrow[[E]_f]{f(\ell)} e'$ y $f(\ell) \notin B$ sii $(e,b) \xrightarrow[[E]_f\|[B]_f]{f(\ell)} (e',b)$

B) $(e,b) \xrightarrow[[E\|B]_f]{f(\ell)} (e,b')$ sii $(e,b) \xrightarrow[E\|B]{\ell} (e,b')$ sii $\ell \notin E$ y $b \xrightarrow[B]{\ell} b'$ sii
$f(\ell) \notin E$ y $b \xrightarrow[[B]_f]{f(\ell)} b'$ sii $(e,b) \xrightarrow[[E]_f\|[B]_f]{f(\ell)} (e,b')$

C) $(e,b) \xrightarrow[[E\|B]_f]{f(\ell)} (e',b')$ sii $(e,b) \xrightarrow[E\|B]{\ell} (e',b')$ sii $e \xrightarrow[E]{\ell} e'$ y $b \xrightarrow[B]{\ell} b'$ sii
$e \xrightarrow[[E]_f]{f(\ell)} e'$ y $b \xrightarrow[[B]_f]{f(\ell)} b'$ sii $(e,b) \xrightarrow[[E]_f\|[B]_f]{f(\ell)} (e',b')$

Luego, la función $g(x) = x$ es una función isomorfica que preserva transiciones. □

El manejador de interrupciones define la ejecución secuencial de dos LTS donde se produce el cambio de un LTS al otro al momento en el cual se ejecutan un evento ($\alpha$).

**Definición 2.1.9.** (Manejador de interrupciones) *Sea $E = (S_E, A_E, \Delta_E, e_0)$ y $N = (S_N, A_N, \Delta_N, n_0)$ un LTS, $H$ una relación de interrupción tal que $H \subseteq (S_E \times S_N)$, y $\alpha$ un evento de interrupción tal que $\alpha \notin (A_E \cup A_N)$.*
*El manejador de interrupción $E \lhd\!\rhd^{\alpha}_{H} N$ es un LTS definido como $(S_E \cup S_N, A_E \cup A_N \cup \{\alpha\}, \Delta_{\lhd\!\rhd}, e_0)$, donde $\Delta_{\lhd\!\rhd}$ es la relación mas chica que satisface las siguientes reglas:*

$$\frac{(e,\ell,e') \in \Delta_E}{(e,\ell,e') \in \Delta_{\lhd\!\rhd}} \quad \frac{(n,\ell,n') \in \Delta_N}{(n,\ell,n') \in \Delta_{\lhd\!\rhd}} \quad \frac{}{(e,\alpha,n) \in \Delta_{\lhd\!\rhd}}\ (e,n) \in H$$

**Propiedad 2.1.3.** *Sean $A, B, C$ y $D$ LTS, $H$ y $K$ relaciones de interrupción y $\ell$ un evento de interrupción. La siguiente equivalencia vale:*

$$(A \lightning^{\alpha}_{H} B) \,||\, (C \lightning^{\alpha}_{K} D) \sim ((A||C) \lightning^{\alpha}_{H'} B) \,||\, ((A||C) \lightning^{\alpha}_{K'} D)$$

*donde $((a, c), b) \in H'$ sii $(a, b) \in H$, y, $((a, c), d) \in K'$ sii $(c, d) \in K$.*

*Demostración.* Para probar que estos LTS son bisimilares debemos encontrar una relación de bisimulación entre las dos máquinas. La relación binaria la podemos partir en dos partes. La parte previa y la parte posterior a $\alpha$.

Para probar que existe una relación de bisimulación antes de $\alpha$, veamos que el estado actual de $A$ y de $C$ son algunos tal que pueden transicionar por un evento que comparten ($\ell$) o que no ($e$). Para el caso donde es compartido tenemos que $A \xrightarrow{\ell} A'$ y $C \xrightarrow{\ell} C'$. Este tipo de ejecución puede simularse en $A||C$. Para el caso donde no es compartido, tenemos que $A \xrightarrow{e} A'$ ó $C \xrightarrow{e} C'$. Este tipo de ejecución también puede simularse en $A||C$. Si quisieramos probar que $A||C$ puede simular al termino de la izquierda antes de $\alpha$ alcanza con ver que para todo evento $a$ que pueda ejecutar $A||C$ vale que $a$ puede ejecutarse en $A$ o en $C$ por Definición 2.1.7. Por lo tanto existe una relación de bisimulación antes de $\alpha$.

Cuando sucede $\alpha$, el LTS $(A \lightning^{\alpha}_{H} B) \,||\, (C \lightning^{\alpha}_{K} D)$ sigue simulando al otro ya que $H$ y $H'$ están definidas para que se preserven los estados. Lo mismo sucede con $K$ y $K'$.

Luego de $\alpha$ los dos términos pasan a comportarse como $B||D$. Por lo tanto la relación de igualdad es una relación de bisimulación. □

## 2.2 Lógica Lineal Temporal basada en Fluents

La Lógica Lineal Temporal o LTL por su sigla en inglés (Lineal Temporal Logic) se utilizan normalmente para describir requerimientos de comportamiento [LL00; GM03].

La motivación de utilizar LTL con fluents es que este nos provee un framework para especificar propiedades temporales basadas en estados para modelos basados en eventos [GM03]

La Lógica Lineal Temporal basada en Fluents, o FLTL por su sigla en inglés (Fluent Linear Temporal Logic [GM03]) es una lógica lineal temporal para razonar sobre fluents. Un *fluent* está definido como un par de conjuntos y un valor Booleano: $f = \langle I, T, Init \rangle$, donde $f.I$ es un conjunto de acciones iniciadoras, $f.T$ es un conjunto de acciones terminadoras y $f.I \cap f.T = \emptyset$. Un fluent puede ser inicialmente *true* o *false* tal como lo indique $f.Init$. Cada acción $\ell \in Act$ induce un fluent que llamaremos $\dot{\ell} = \langle\, \ell, Act \setminus \{\ell\}, \bot \rangle$. El alfabeto de un fluent es la union de las acciones iniciadoras y terminadoras del fluent.

Sea $\mathcal{F}$ el conjunto de todos los posibles fluents y $d$ sea una fluent definition, $d : \mathcal{F} \to \langle I, T, Init \rangle$. Una fórmula FLTL está definida inductivamente usando los conectivos Booleanos estandar y los operadores temporales $\mathbf{X}$ (siguiente), $\mathbf{U}$ (until) de la siguiente manera:

$$\varphi ::= f \mid \neg\varphi \mid \varphi \vee \psi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \psi,$$

donde $f \in \mathcal{F}$. Definimos (como en [GM03]), $\varphi \wedge \psi$ como $\neg\varphi \vee \neg\psi$, $\Diamond\varphi$ (eventualmente) como $\top \mathbf{U} \varphi$, $\Box\varphi$ (siempre) como $\neg\Diamond\neg\varphi$, y $\varphi \mathbf{W} \psi$ (weak until) como $\varphi \mathbf{U} \psi \vee \Box\varphi$.

Sea $\Pi$ el conjunto de trazas infinitas sobre *Act*. La traza $\pi = \ell_0, \ell_1, \ldots$ satisface el fluent $f$ para la definición de fluents $d$ en la posición $i$, denotado $\pi, i \models_d f$, si y solo sí, una de las siguientes condiciones vale:

I) $d(f).Init \wedge (\forall j \in \mathbb{N} \cdot 0 \leq j \leq i \;\Rightarrow\; \ell_j \notin d(f).T)$

II) $\exists j \in \mathbb{N} \cdot (j \leq i \wedge \ell_j \in d(f).I) \quad \wedge \quad (\forall k \in \mathbb{N} \cdot j < k \leq i \Rightarrow \ell_k \notin d(f).T)$

$$
\begin{array}{lcl}
\pi, i \models_d \neg\varphi & \triangleq & \pi, i \not\models_d \varphi \\
\pi, i \models_d \varphi \vee \psi & \triangleq & (\pi, i \models_d \varphi) \vee (\pi, i \models_d \psi) \\
\pi, i \models_d \mathbf{X}\varphi & \triangleq & \pi, i+1 \models_d \varphi \\
\pi, i \models_d \varphi \mathbf{U} \psi & \triangleq & \exists j \geq i \cdot \pi, j \models_d \psi \quad \wedge \\
 & & \quad \forall\, i \leq k < j \cdot \pi, k \models_d \varphi
\end{array}
$$

**Fig. 2.1:** Semántica del operador de satisfacción.

En otras palabras, un fluent vale en la posición $i$ si y solo si I) vale inicialmente y nunca sucede una acción terminadora hasta la posición $i$ o II) alguna acción iniciadora ocurrió, y no ha sucedido la ocurrencia de una acción terminadora desde ese entonces hasta la posición $i$.

Dada una traza infinita $\pi$, la satisfacción de una formula $\varphi$ en la posición $i$ para una definición de fluents $d$, denotado $\pi, i \models_d \varphi$, esta definido como en la Figura 2.1. Decimos que $\varphi$ vale en $\pi$ para una definición de fluents $d$, denotado $\pi \models_d \varphi$, si $\pi, 0 \models_d \varphi$.

Para poder razonar sobre dos especificaciones (la vieja y la nueva) que pueden tener diferentes alcances (i.e. refiere a diferentes conjuntos de eventos) introduzco una extensión de definición de fluent.

**Definición 2.2.1.** (Extensión de Definición de Fluents) *Sean $d$ y $d'$ Definición de Fluents sobre $\Sigma$ y $\Sigma'$ respectivamente, $d_e$ es una extensión de Definición de Fluents de $d$ y $d'$, si y solo sí, valen las siguiente condiciones:*

- $\forall\, (f_e, \langle I_e, T_e, Init_e\rangle) \in d_e$, $\exists\, (f, \langle I, T, Init\rangle) \in d$ *tal que* $I_e = I \cup \dot{I}$ *y* $T_e = T \cup \dot{T}$ *donde* $\dot{I} \subseteq (\Sigma' \setminus \Sigma)$ *y* $\dot{T} \subseteq (\Sigma' \setminus \Sigma)$.
- $\forall\, (f, \langle I, T, Init\rangle) \in d$, $\exists\, (f_e, \langle I_e, T_e, Init_e\rangle) \in d_e$ *tal que* $I_e = I \cup \dot{I}$ *y* $T_e = T \cup \dot{T}$ *donde* $\dot{I} \subseteq (\Sigma' \setminus \Sigma)$ *y* $\dot{T} \subseteq (\Sigma' \setminus \Sigma)$.
- $\forall\, (f_e, \langle I_e, T_e, Init_e\rangle) \in d_e$, $\exists\, (f', \langle I', T', Init'\rangle) \in d'$ *tal que* $I_e = I' \cup \dot{I}$ *y* $T_e = T' \cup \dot{T}$ *donde* $\dot{I} \subseteq (\Sigma \setminus \Sigma')$ *y* $\dot{T} \subseteq (\Sigma \setminus \Sigma')$.
- $\forall\, (f', \langle I', T', Init'\rangle) \in d'$, $\exists\, (f_e, \langle I_e, T_e, Init_e\rangle) \in d_e$ *tal que* $I_e = I' \cup \dot{I}$ *y* $T_e = T' \cup \dot{T}$ *donde* $\dot{I} \subseteq (\Sigma \setminus \Sigma')$ *y* $\dot{T} \subseteq (\Sigma \setminus \Sigma')$.

Las propiedades de safety son generalmente caracterizadas como “nada malo puede ocurrir”. La propiedad de exclusión mutua - siempre a lo sumo un proceso está en la sección crítica - es una propiedad de safety típica. Lo que dice es que lo malo (tener a dos o más procesos en la sección crítica en simultaneo) nunca ocurre. Por otro lado, tenemos las propiedades de liveness que contrariamente a las de safety predican que “algo bueno” pasará en el futuro. La característica fundamental que tienen estas propiedades es que para encontrar una violación de una de estas propiedades alcanza con una traza finita para una propiedad de safety mientras que se necesita una traza infinita para una propiedad de liveness. Esto se debe a que si encuentro una traza finita donde “lo malo” sucede, encontré una violación a la propiedad de safety, mientras que, encontrar que “algo bueno” no sucederá, necesito una traza infinita. Utilizaré esta intuición para definir formalmente las propiedades de safety y de liveness.

**Definición 2.2.2** (Propiedad de Safety)**.** *Sea $\varphi$ una fórmula en FLTL en el alfabeto de $Act$, diremos que $\varphi$ es una propiedad de safety si existe una traza $\pi$ finita con el alfabeto de $Act$ tal que $\pi \not\models \varphi$.*

**Definición 2.2.3** (Propiedad de Liveness)**.** *Sea $\varphi$ una fórmula en FLTL. Si $\varphi$ no es una propiedad de safety, entonces es una Propiedad de Liveness.*

## 2.3 Problema de Control de Sistemas de Transiciones Etiquetados

La noción de legalidad (basada en Interface Automata [AH01]) nos permite modelar controlabilidad y monitoreabilidad de eventos. Un LTS legal no puede bloquear la ocurrencia de acciones que no controla (i.e. solamente monitorea), y además, no puede ejecutar acciones que controla pero que el ambiente no acepta.

**Definición 2.3.1.** (LTS Legal) *Sean $P = (S_P, A_P, \Delta_P, p_0)$ y $Q = (S_Q, A_Q, \Delta_Q, q_0)$ LTS, $C \subseteq (A_P \cup A_Q)$ un conjunto de eventos que $P$ controla y $U \subseteq (A_P \cup A_Q)$ un conjunto de eventos que $P$ no controla.*

*Decimos que $P$ es un LTS legal para $Q$ con respecto a $(C, U)$ si $\forall\ (p, q) \in S_{P\|Q}$, $p$ y $q$ son legales de la siguiente manera:*

- $(\Delta_P(p) \cap U) = (\Delta_Q(q) \cap U)$, y
- $(\Delta_P(p) \cap C) \subseteq (\Delta_Q(q) \cap C)$.

Nótese que adoptamos una noción levemente mas fuerte que la presentada en [AH01]. En esta presentación, requerimos que $P$ sea menos o igual de robusto que lo que $Q$ puede exhibir, es decir, $P$ no puede aceptar más acciones no controlables que las que $Q$ posee.

Un problema de control LTS puede ser descrito de la siguiente manera: Dado un LTS que describe el comportamiento de un ambiente, un conjunto de acciones controlables, una formula FLTL que describe el objetivo para el controlador y una definición de fluents, el problema de control LTS es encontrar un LTS que solamente restringe la ocurrencia de acciones controlables y garantiza que la composición en paralelo entre el ambiente y el LTS controlador es libre de deadlocks y satisface el objetivo.

**Definición 2.3.2** (Control de LTS [D'I+13])**.** *Sean $E = (S_E,\ A_E, \Delta_E, e_0)$ un modelo del ambiente en la forma de un LTS, $A \subseteq A_E$ un conjunto de eventos controlables, $\widehat{A} \subseteq A_E$ un conjunto de acciones no controlables, $G$ un objetivo del controlador expresado como una fórmula FLTL, y $d$ una definición de fluents.*
*Una solución al problema de control de LTS con especificación $\mathcal{E} = (E, G, d, A, \widehat{A})$ es un LTS $C = (S_C, A_E, \Delta_C, c_0)$ tal que $C$ es un LTS legal a $E$ con respecto a $(A, \widehat{A})$, $E\|C$ es libre de deadlocks, y $E\|C \models_d G$.*

**Propiedad 2.3.1.** *Sea $C$ una solución a un problema de control LTS con especificación $\mathcal{E} = (E, G, d, A, \overline{A})$, entonces, $E\|C \sim C$.*

*Demostración.* La relación binaria $\mathcal{R}$ tal que para todo $(e, c) \in E\|C$, $(e, c)\ \mathcal{R}\ c$ es una bisimulación

Nótese que $C$ y $E$ comparten el mismo alfabeto, y por lo tanto, cada transición de $E\|C$ es una transición en $C$ por la tercer regla de la composición paralela (Definición 2.1.7). Además, como $C$ es un LTS legal para $E$, entonces, $C$ es un subconjunto de $E$. (i.e. $C$ tiene menos comportamiento que $E$). Por lo tanto, $(e,c) \xrightarrow[E\|C]{\ell} (e',c')$ sii $c \xrightarrow[C]{\ell} c'$ y $(e',c') \ \mathcal{R} \ c'$. □

Existen diversos tipos de controladores, diferenciándose entre ellos por las propiedades que cumplen o los requerimientos que satisfacen. En esta tesis solo vamos a utilizar los tipos de controladores que definimos a continuación.

**Definición 2.3.3** (Controladores Safe)**.** *Sea $\mathcal{E} = (E, G, L_C)$ un problema de control. Decimos que $C$ es un controlador safe si y solo si $G$ es una propiedad en FLTL que tiene contraejemplo finito.*

Notar que la definición anterior se cumple por ejemplo para toda fórmula que pueda escribirse como $\Box\varphi$ con $\varphi$ proposicional. Un contraejemplo (finito) para esa formula es encontrar un estado del LTS donde no valga $\varphi$.

**Definición 2.3.4** (Controladores Live)**.** *Sea $\mathcal{E} = (E, G, L_C)$ un problema de control. Decimos que $C$ es un controlador live si no es safe.*

**Definición 2.3.5** (Controladores Maximales)**.** *Sea $\mathcal{E} = (E, G, L_C)$ un problema de control. Decimos que $C$ es un controlador maximal para $\mathcal{E}$ si $C$ es una solución a $\mathcal{E}$ y para toda otra solución $C'$ a $\mathcal{E}$ las trazas de $C$ incluyen a aquellas de $C'$.*

**Propiedad 2.3.2** (Existencia de Controladores Maximales)**.** *Un controlador maximal existe para cualquier problema de control realizable $\mathcal{E} = (E, G, L_C)$ donde $G$ es una propiedad de safety.*

## 2.4 Grafos Dinámicos de Condición y Respuesta

Como lenguaje de especificación de procesos de negocio, vamos a utilizar el lenguaje de Grafos Dínamico de Condición y Respuesta o DCR Graphs por su sigla en inglés

(Dynamic Condition Response Graph), siendo este un lenguaje declarativo para expresar requerimientos de negocio, desarrollado por DCR Solutions [Dcr] y utilizado en muchos organismos públicos del gobierno de Dinamarca. Los requerimientos son definidos dando restricciones entre actividades. Las actividades son denotadas con cajas y las restricciones con flechas entre cajas. Formalmente vamos a definir este lenguaje siguiendo la definición más simple y utilizada en [Muk12], que no incluye subprocesos, tiempo ni dataflow:

**Definición 2.4.1.** (Diagrama Dinámico de Condición y Respuesta) *Un* Diagrama Dinámico de Condición y Respuesta *(DCR Graph) es una tupla* $DG = (A, R, M)$ *donde* $A$ *es un conjunto finito de actividades, los nodos del grafo.* $R$ *es un conjunto finito de arcos del grafo. Los arcos están particionados en cinco tipos de arcos, llamados y dibujados de la siguiente manera: condiciones (*$\rightarrow\bullet$*), respuestas (*$\bullet\rightarrow$*), inclusiones (*$\rightarrow_+$*), exclusiones (*$\rightarrow\%$*) y milestones (*$\rightarrow_\Diamond$*).* $M$ *es el marking del grafo que representa el estado de ejecución. Es una tripla de conjuntos de actividades* $(Ex, Re, In)$*, donde* $Ex$ *son las actividades ejecutadas previamente,* $Re$ *son las actividades que estan actualmente pendientes de ejecución y,* $In$ *son las actividades que estan actualemente incluidas. Para todo* $(e, e') \in E \times E$*,* $e \rightarrow_+ e'$ *o* $e \rightarrow \% e'$ *pero no ambos. Denotamos* $(\bullet\rightarrow e) = \{e' \in A \mid e' \bullet\rightarrow e\}$*,* $(e\bullet\rightarrow) = \{e' \in A \mid e\bullet\rightarrow e'\}$*, y similarmente para* $\rightarrow\bullet$*,* $\rightarrow_+$*,* $\rightarrow\%$ *y* $\rightarrow_\Diamond$*. Para la mayoría de los DCR graphs, vamos a asumir que inicialmente* $In = A$ *y* $Re = Ex = \emptyset$*, que es el escenario más común (i.e., todas las actividades están incluidas, y ninguna está ejecutada ni requerida).*

Para entender la semántica de ejecución de un DCR Graph tenemos que dar la definición de cuando un evento está habilitado para ser ejecutado y cual es el resultado de la ejecución.

Para que una actividad $a$ esté habilitada, primero que todo, debe estar incluido en el grafo (a). Luego, todos los eventos incluidos que son condiciones de la actividad $a$ deben haber sido ejecutadas previamente (b). Por último, ninguna de las actividades incluidas que son milestones de $a$, están en el conjunto de actividades pendientes (c).

**Definición 2.4.2.** (Actividades habilitadas en un DCR Graph) *Sea $DG = (A, R, M)$ un DCR graph, con $M = (Ex, Re, In)$. Una actividad $a \in A$ está habilitada si y sólo si*

*(a)* $a \in In$

*(b)* $(In \cap (\rightarrow\bullet a)) \subseteq Ex$, *y*

*(c)* $Re \cap In \cap (\rightarrow \Diamond a) = \emptyset$

*Notaremos con $M \xrightarrow{a}$ que un marking $M$ tiene habilitada la actividad $a$. Análogamente, notaremos con $DG \xrightarrow{a}$ que la actividad $a$ está habilitada en el DCR graph $DG$.*

Ahora pasaremos a formalizar el cambio de un marking cuando una actividad habilitada es ejecutada. Primero, la actividad ejecutada pasa a estar incluida en el conjunto de ejecutadas ($Ex$) y quitada del conjunto de actividades pendientes ($Re$). Luego, todas las actividades que son respuestas de la actividad ejecutada, son incluidas al conjunto de respuestas pendientes ($Re$). Note que si una actividad se tiene a si misma como respuesta, la actividad quedá pendiente luego de ser ejecutada. Similarmente, el conjunto de actividades incluidas ($In$) es actualizado agregando actividades que son incluidas por la acción ejecutada y quitando las actividades que fueron quitadas por la actividad ejecutada. Formalmente:

**Definición 2.4.3.** (Resultado de ejecución en un DCR Graph) *Sea $DG = (A, R, M)$ un DCR Graph, con $M = (Ex, Re, In)$ y $M \xrightarrow{a}$. El resultado de ejecutar $a$ es un DCR Graph $DG' = (A, R, M')$ con $M' = (Ex', Re', In')$ (notado $DG \xrightarrow{a} DG'$) tal que*

*(A)* $Ex' = Ex \cup \{a\}$,

*(B)* $Re' = (Re \setminus \{a\}) \cup (a\bullet\rightarrow)$, *y*

*(C)* $In' = (In \cup (a \rightarrow_+)) \setminus (a\rightarrow\%)$.

*Cuando no este especificado, vamos a suponer que para un DCR Graph inicialmente vale que $In = A$ y $Re = Ex = \emptyset$.*

Habiendo definido cuando las actividades son habilitadas para su ejecución y el efecto de ejecutar una actividad, definimos ejecuciones finitas e infinitas en DCR Graphs de la siguiente manera:

**Definición 2.4.4.** (Ejecución de un DCR Graph) *Sea $DG = (A, R, M)$ un DCR Graph con $M = (Ex, Re, In)$ tal que $M \xrightarrow{a}$. Una ejecución de $DG$ es una secuencia (finita o infinita) de DCR Graphs $DG_i$ y actividades $a_i$ tales que: $DG = DG_0 \xrightarrow{a_0} DG_1 \xrightarrow{a_1} \ldots$. Una traza de $DG$ es una secuencia de actividades $a_i$ asociadas a una ejecución de $DG$. Escribiremos $runs(DG)$ y $traces(DG)$ para el conjunto de ejecuciones y trazas de $DG$, respectivamente.*

**Definición 2.4.5.** (Marking Alcanzable) *Sea $D$ un DCR graph. Un marking $M$ es alcanzable desde $D$ si existe una corrida finita $D_0 \xrightarrow{e_0} D_1 \xrightarrow{e1} \ldots D_n$ donde $M$ es el marking de $D_n$.*

Introducimos la noción de marking consistente con respecto a una secuencia de actividades para capturar una actualización correcta de un marking con respecto a las restricciones de un DCR graph independientemente de si la secuencia corresponde a una corrida del DCR graph.

**Definición 2.4.6.** (Marking Consistente) *Sea $D = (A, R, M)$ un DCR Graph con marking $M = (Ex, Re, In)$. Decimos que un marking $M' = (Ex', Re', In')$ es consistente con $M$ y una secuencia finita de actividades $\pi$ si $Ex' = \delta_{Ex}$, $Re' = \delta_{Re}$ y $In' = \delta_{In}$, donde $\delta_{Ex}, \delta_{Re}$ y $\delta_{In}$ están definidas de la siguiente manera ($\lambda$ corresponde a la secuencia vacía):*

$$\delta_{Ex}(Ex, \lambda) = Ex \quad y \quad \delta_{Ex}(Ex, e.\pi) = \delta_{Ex}(Ex, \pi) \cup \{e\}$$
$$\delta_{Re}(Re, \lambda) = Re \quad y \quad \delta_{Re}(Re, e.\pi) = (\delta_{Re}(Re, \pi) \setminus \{e\}) \cup (e\bullet\!\!\rightarrow)$$
$$\delta_{In}(In, \lambda) = In \quad y \quad \delta_{In}(In, e.\pi) = \delta_{In}(In, \pi) \cup (e \rightarrow_+) \setminus (e\!\rightarrow\!\%)$$

# 3 Actualización Dinámica de Controladores de Eventos Discretos

> *We were doing well but things happened.*
>
> — **Mauricio Macri**
> (Empresario y político)

En este capítulo introducimos la definición formal al problema de actualizar dinámicamente un proceso (que puede ser software o no), representado por un componente al que llamaremos controlador. Este controlador es fácilmente representado po un sistema de eventos discretos. Para explicar la técnica desarrollada, este capítulo cuenta con tres secciones. Primero, una sección donde introducimos un ejemplo motivador detallando cual es el controlador a ser actualizado. Luego, una sección donde definimos formalmente el problema a resolver y finalmente una sección que presenta una posible solución a este problema mediante un algoritmo de síntesis de controladores de eventos discretos. La formalización del problema fue introducida originalmente en [Nah+16] bajo controladores de eventos discretos levemente distintos a los presentados en esta tesis. Luego en [Nah+18] introducimos los LTS con fluents que nos permitió producir una optimización de la solución. En esta presentación vamos a utilizar el enfoque tal y como fue presentado en [Nah+18].

## 3.1 Ejemplo Motivador: Cadena de Montaje

Imaginemos un escenario de automatización industrial [LL95] en el cual un brazo robótico aplica varias herramientas a productos crudos tomados de un bandeja de entrada (*In*), y luego, suelta los productos terminados en la bandeja de salida (*Out*). El funcionamiento de la fábrica está siendo dirigida por un software controlador

que secuencia comandos que están declarados en la especificación $E$, $G$ y $A$. $A$ es el conjunto de eventos que el controlador puede ejecutar, por ejemplo *drill*, *polish*, *clean*, y *stamp*. $E$ modela las suposiciones en los que el controlador debe confiar para alcanzar sus objetivos. $E$ puede incluir, por ejemplo, que luego de que la herramienta de pulido ejecute el comando de pulir el producto $x$ (e.g., $polish(x)$) el sistema recibirá una respuesta de $polishOK(x)$ o $polishNOK(x)$ representando éxito o fracaso de la acción de pulir $x$. Esta suposición puede fácilmente ser modelada con un autómata. Finalmente, $G$ modela los objetivos para el controlador. Por ejemplo, $G$ puede requerir que un producto sea puesto en la bandeja de salida (*Out*) solo si esta pieza ha sido limpiada, pulida y taladrada (en ese orden) y no han sucedido errores, o de manera alternativa, si un error ha ocurrido debe haberse marcado este producto como defectuoso. Una formalización usando fluent linear temporal logic (ver FLTL en la Sección 2.2) de alguno de estos objetivos, puede ser:

$$\textit{ToolOrder} \equiv \Box\, \forall x \cdot (\textit{Cleaned(x)} \Rightarrow \textit{Polished(x)}) \ \wedge (\textit{Polished(x)} \Rightarrow \textit{Drilled(x)})$$

$$\textit{ToolsRequired} \equiv \Box\, \forall x \cdot out(x) \Rightarrow (\textit{Faulty(x)} \vee (\textit{Drilled(x)} \ \wedge \textit{Polished(x)} \wedge \textit{Cleaned(x)} \ \wedge \neg \textit{Stamped(x)}))$$

$$\textit{NoProcessingIfFaulty} \equiv \Box\, \forall x \cdot (\textit{Faulty(x)} \Rightarrow \neg(\textit{drill(x)} \ \vee \ \textit{polish(x)} \ \vee \ \textit{clean(x)}))$$

$$Cleaned(x) \equiv \langle cleanOk(x), out(x), \bot\rangle,\ Polished(x) \equiv \langle polishOk(x), out(x), \bot\rangle,$$

$$Drilled(x) \equiv \langle drillOk(x), out(x), \bot\rangle,\ Stamped(x) \equiv \langle stampOk(x), out(x), \bot\rangle,$$

$$Faulty(x) \equiv \langle faultOk(x), out(x), \bot\rangle$$

Veamos ahora un escenario en el cual mientras la fábrica está procesando productos, se decide que el proceso de producción debe cambiar. Esta decisión puede deberse a diversos factores: el conjunto de herramientas disponibles cambió (e.g. una herramienta se rompió, o una nueva herramienta es introducida), la especificación de como se procesa un tipo de producto cambia (e.g. nuevas reglas de negocio), u otras restricciones cambian (e.g. un nuevo requerimiento de consumo de energía fuerza el uso de ciertas máquinas de manera concurrente).

Una simple solución a este problema es esperar a la cadena de montaje se vacíe (i.e. esperar a que todos los productos sean procesados y movidos a la bandeja de salida),

parar la planta, cambiar el controlador, y luego, reiniciar la planta. Una actualización off-line como esta podría no ser aceptable en el caso de que frenar por completo durante un período a la fábrica genere serios problemas económicos.

Supongamos que por razones de negocios, la herramienta de pulido debe ser intercambiada por una herramienta de pintado, y por lo tanto, el proceso de producción debe re-ordenarse. El nuevo proceso de producción está definido por la especificación $E'$, $G'$ y $A'$ donde $A'$ no tiene *polish* pero tiene *paint*. Por ejemplo, $E'$ incluirá supuestos de como la herramienta de pintado funcionará y $G'$ podría tener las siguientes formulas corregidas:

$$\textit{ToolOrder'} \equiv \Box\, \forall x \cdot (\textit{Drilled(x)} \Rightarrow \textbf{\textit{Painted(x)}}) \wedge (\textbf{\textit{Painted(x)}} \Rightarrow \textit{Cleaned(x)})$$

$$\textit{ToolsRequired'} \equiv \Box\, \forall x \cdot out(x) \Rightarrow (\textit{Faulty(x)} \vee (\textit{Drilled(x)} \wedge \textbf{\textit{Painted(x)}} \wedge \textit{Cleaned(x)} \wedge \neg \textit{Stamped(x)}))$$

¿Cómo debería el controlador actual ser actualizado para satisfacer la nueva especificación? ¿Cuándo es "seguro" intercambiar los controladores? ¿Qué estrategia debería el nuevo controlador usar una vez instalado? ¿Cuándo puede establecerse el driver de la herramienta de pintado y añadirse a la arquitectura de software actual? ¿Cuándo puede ser removida la herramienta de pulido de la arquitectura actual? Las respuestas a estas preguntas dependen del dominio. Como se explica en [VAS00], las respuestas a estas preguntas son requerimientos de transición y estos deben ser propuestos por los expertos del dominio.

¿Por ejemplo, que debería hacerse con los productos que han sido parcialmente procesados siguiendo las reglas $G$? ¿Deberían estos productos terminar de elaborarse siguiendo los nuevos requerimientos expresados en $G'$? ¿Debería un producto pulido pero no limpio ser limpiado para luego colocarse en la bandeja de salida? ¿Debería ser descartado inmediatamente sin más procesamiento?. O ¿debería demorarse la actualización hasta que no haya mas productos pulidos en la cadena de montaje? Claramente, la respuesta a esta pregunta de como tratar a las instancias que están a medio camino en el workflow es especifica del dominio.

Para especificar requerimientos de transición, primero debo definir qué es una transición y como nos referimos a esta. Para esto, supongamos que dentro del proceso de actualización el controlador va a tener un comando para indicar cuando la vieja especificación es dada de baja (*stopOldSpec*) y un evento para indicar desde que momento la nueva especificación empieza a garantizarse (*startNewSpec*). Para señalar si estos eventos han ocurrido, utilizo los fluents que definio a continuación:

*OldSpecStopped* $= \langle$*stopOldSpec*$, \emptyset, \bot\rangle$ y *NewSpecStarted* $= \langle$*startNewSpec*$, \emptyset, \bot\rangle$.

Un requerimiento de transición posible es que no haya productos pulidos en la linea de montaje:

$$T_1 = \Box((\textit{startNewSpec} \Rightarrow \neg \textit{OldSpecStopped}) \wedge (\forall x \cdot \textit{OnProductionLine(x)} \Rightarrow \neg \textit{Polished(x)}))$$

Otro requerimiento de transición podría ser que los productos puedan ser puestos en la bandeja de salida de dos formas posibles: siguiendo la especificación $G'$ o habiendo sidos marcados como fallidos. Esto permite, por ejemplo, que se realice la actualización incluso habiendo elementos que no pueden cumplir con $G'$, pero si pueden ser marcados como defectuosos.

$$T_2 \equiv \Box((\textit{OldSpecStopped} \wedge \neg \textit{NewSpecStarted}) \Rightarrow (\textit{ToolOrder'} \wedge \textit{NoProcessingIfFaulty} \wedge \textit{ToolsRequired''} \wedge \ldots))$$

$$\textit{ToolsRequired''} \equiv \Box \forall x \cdot out(x) \Rightarrow (\textit{Faulty(x)} \vee \textit{Stamped(x)} \vee (\textit{Drilled(x)} \wedge \textit{Painted(x)} \wedge \textit{Cleaned(x)} \wedge \neg \textit{Polished(x)}))$$

Volviendo al problema de cuando reconfigurar, $T_1$ puede además incluir un requerimiento para no permitir reconfiguración cuando la herramienta de pulido está trabajando en un producto (i.e., *reconfigure* $\implies \forall x \cdot \neg BeingPolished(x)$). Este requerimiento puede forzar que el comando *reconfigure* (que instalará el driver de la herramienta de pintado y desinstalará el driver de la herramienta de pulido) es ejecutado de manera segura.

Nótese que si $T_1$ fuese seleccionada, podría surgir un interesante problema de liveness. Podría ser el caso en el que siempre haya un producto pulido en la cadena de montaje: si nuevos productos llegan regularmente y el controlador actual los manda a pulir, el comando de reconfigurar puede nunca ser ejecutado, violando así a $T_1$. El controlador actual necesita ser guiado a un estado en el cual la actualización pueda ocurrir. De hecho, debe dejar de pulir y finalizar todos los productos que estén ya pulidos.

El hecho de que el controlador actual necesite ser guiado a un estado actualizable muestra que una estrategia de actualización de controladores requiere reemplazar el controlador actual por otro que puede continuar satisfaciendo $G$ (e.g. terminando productos ya pulidos respetando $G$) mientras asegura que eventualmente un estado actualizable es alcanzado (e.g. no hay productos pulidos en la cadena de montaje). Por lo tanto, la solución de como el sistema es actualizado de $E$, $G$ y $A$; a $E'$, $G'$ y $A'$ satisfaciendo también $T_1$ es teniendo un controlador actualizador que reemplace al controlador actual, guie al sistema a estados en los cuales pueda hacerse la reconfiguración, pueda indicar que la vieja especificación deja de valer y que la nueva empieza a valer sin violar $T_1$.

De hecho, presentamos una técnica totalmente automática que garantiza una actualización de controladores correcta para la cadena de montaje. Informalmente, el input de la técnica que presentamos es la especificación actual $E$, $G$ y $A$, el controlador que esta actualmente dirigiendo la cadena de montaje ($C$), la nueva especificación $E'$, $G'$ y $A'$, y el requerimiento de transición ($T$). El output (que debería ser una solución al problema) es un controlador actualizador ($C'$) que asegura que el sistema resultante satisface los siguientes requerimientos:

(I) $C$ puede ser reemplazado por $C'$ en cualquier momento.
(II) $G$ va a valer hasta que $C'$ haga *stopOldSpec*.
(III) $T$ que predica sobre *stopOldSpec*, *startNewSpec* y *reconfigure* vale.
(IV) $G'$ va a valer una vez que $C'$ haga *startNewSpec*.

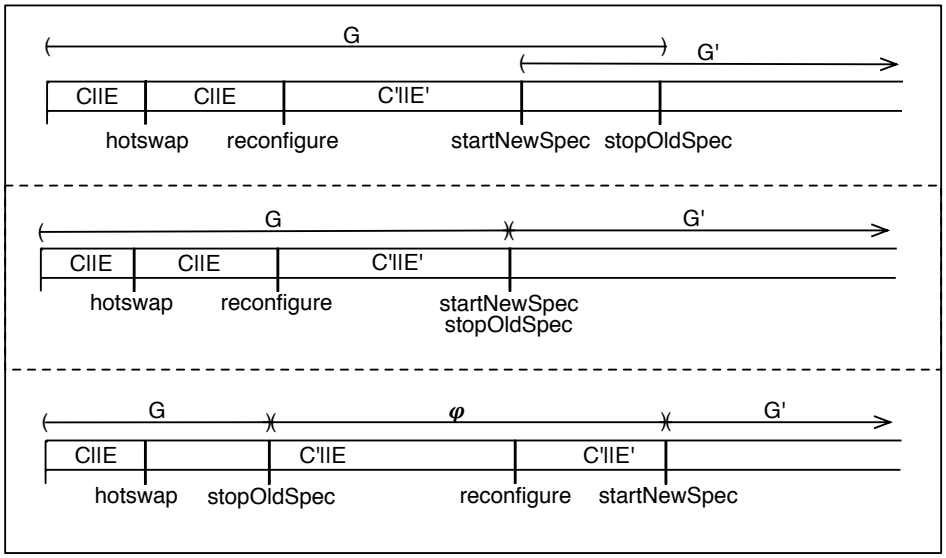


**Fig. 3.1:** Actualización dinámica de controladores con diferentes requerimientos de transición (de arriba a abajo): *stopOldSpec* $\Rightarrow$ *NewSpecStarted*, *startNewSpec* $\Rightarrow$ *OldSpecStopped* y (*OldSpecStopped* $\wedge \neg$*NewSpecStarted*) $\Rightarrow \varphi$, donde $\varphi$ es una propiedad de safety.

(V) Una vez que $C$ es reemplazado por $C'$, los siguientes eventos van a suceder eventualmente: *reconfigure*, *startNewSpec* and *stopOldSpec*.

Un diagrama esquemático que muestra ejemplos de un controlador actualizador dinámico para tres requerimientos de transición distintos puede verse en Figura 3.1. Destaco los eventos importantes (*hotSwap*, *reconfigure*, *stopOldSpec* y *startNewSpec*), como *hotSwap* y *reconfigure* cambian el sistema corriendo (desde $C\|E$ a $C'\|E$ y desde $C'\|E$ a $C'\|E'$ respectivamente), y cuando los objetivos $G$ y $G'$ valen.

Nótese que el computo de un controlador actualizador se realiza mientras que el sistema está ejecutando. Un escenario de actualización procedería de la siguiente manera: La planta está siendo controlada por $C$ para satisfacer $G$ cuando se decide cambiar el proceso de producción. Esto puede ocurrir, por ejemplo, porque algún chequeo de calidad en los productos terminados falló y el problema fue detectado en la pulidora, o también, por alguna otra cuestión relacionada con el negocio en general. Esta decisión puede ser el resultado de una intervención humana o puede ser parte de, por ejemplo, las fases de Monitoreo y Análisis de un MAPE loop [KC03] en un sistema adaptativo. Luego, se debe tomar una decisión a cerca de

qué hacer para resolver los problemas inesperados. En nuestro escenario una decisión podría resultar en $G'$ y $T$. De nuevo, esta decisión puede hacerse manualmente (y en la cadena de montaje es probable que así sea), pero podría incluso ser el resultado de una fase automática o semi-automática de un sistema adaptativo MAPE loop. Nuestra técnica computa un controlador actualizador $C'$ que podría ser intercambiado por $C$ designando un estado inicial de $C'$ acorde al estado actual de $C$. El controlador $C'$ ejecuta una estrategia que satisface el requerimiento de transición $T$ para cualquier posible estado de la planta. Por ejemplo, el proceso tira a la basura aquellos productos parcialmente procesados que no pueden ser continuados para satisfacer los requerimientos $G'$; reconfigura (*reconfigure*) el sistema instalando la maquina de pintado a la cadena de montaje; continua procesando los elementos parcialmente procesados que pueden adaptarse para satisfacer $G'$; y procesa todos los nuevos productos que vengan por la bandeja de entrada siguiendo la especificación $G'$.

## 3.2 Formalización del problema

En esta sección presentamos el problema de actualizar dinámicamente controladores de eventos discretos. En otras palabras, formalizamos qué es lo que significa que un controlador sea cambiado en tiempo de ejecución, garantizando que el sistema va a transicionar correctamente hacia una nueva especificación. Primero formalizamos el hotswap de un controlador, luego la reconfiguración del ambiente y finalmente discutiremos cuando podremos afirmar que todo esto se produce de manera correcta.

### 3.2.1 Hotswap de Controladores

Ahora definimos formalmente el comportamiento del sistema en el cual el controlador actual es cambiado en caliente por otro. Supongamos que un controlador $C$ es interpretado en un ambiente $E$, i.e. un sistema $C||E$. Ahora bien, este controlador $C$ tiene que ser cambiado en caliente por un controlador $C'$. En algunos casos, el

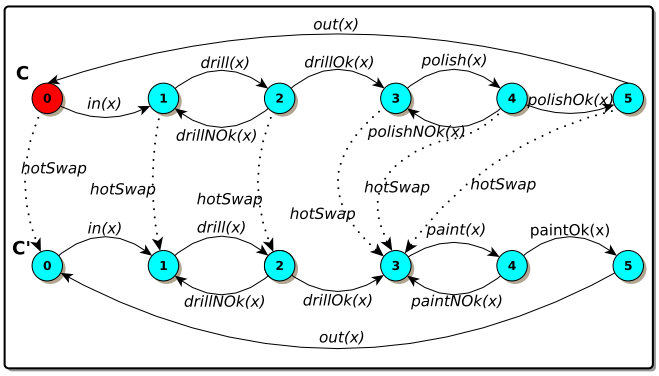


**Fig. 3.2:** Hotswap de controlador por otro para una cadena de montaje (una reducción – dos herramientas. Ningún controlador tiene herramienta limpiadora). $(C \not\downarrow_H^{hotSwap} C')$. Las lineas punteadas representan la transición de hotswap inducida por $H$.

estado inicial de $C'$ al momento en el cual el controlador es seteado puede definirse en tiempo de diseño. Sin embargo, en general el estado inicial de $C'$ depende del estado del sistema en el instante en el cual $C'$ empieza a tomar control. Por lo tanto, daremos soporte para modelar un hotswap de un controlador con un mapeo ($H$) de estados de $C$ a $C'$ que es provisto por el usuario y fija el estado inicial del controlador $C'$. Modelamos el hotswap de $C$ a $C'$ como $(C \not\downarrow_H^{hotSwap} C')\|E$ usando un manejador de interrupciones de la Definición 2.1.9.

Suponemos que *hotSwap* no está en los alfabeto de $C$, $C'$ ni $E$, y que $H$ cubre todo estado de $C$. Consecuentemente, en $(C \not\downarrow_H^{hotSwap} C')\|E$, el evento *hotSwap* no está restringido y puede ocurrir solo una vez pero en cualquier momento. De hecho, este modelo que hace *hotSwap* debe dispararse por la infraestructura subyacente, la cual usará $H$ para definir el estado actual de $C'$ basándose en el estado actual de $C$.

En la parte superior de la Figura 3.2 se muestra un controlador $C$ para una cadena de montaje en versión reducida (i.e, dos herramientas) con unos objetivos levemente modificados (reducido por razones de claridad): luego de recibir un elemento crudo $x$ en la bandeja de entrada (*in(x)*), el controlador taladrará (*drill(x)*) y pulirá (*polish(x)*) el elemento. El comando de poner el elemento en la bandeja de salida (*out(x)*) será ejecutado por el controlador cuando ambas herramientas fueron aplicadas al elemento de manera correcta. En la parte de abajo de la misma figura, mostramos un controlador $C'$ que en vez de pulir elementos, los pinta (*paint(x)*).

La combinación de ambos controladores más las transiciones de linea punteada con etiqueta *hotSwap* es el resultado de $(C \not\downarrow_H^{hotSwap} C')$, donde las transiciones de *hotSwap* coinciden con la relación $H$ que va de estados de $C$ a estados de $C'$ (i.e $H = \{(0,0),(1,1),(2,2),(3,3),(4,3),(5,3)\}$).

Es importante notar que el alfabeto de comunicación de $C \not\downarrow_H^{hotSwap} C'$ es un superconjuto de los alfabetos de $C$ y $C'$. Por lo tanto, un evento $\ell$ que está en el alfabeto de $C$ y no está en el alfabeto de $C'$ va a tener su ocurrencia restringida luego de *hotSwap* en el término $C \not\downarrow_H^{hotSwap} C'$ porque no hay transición en $C'$ etiquetada con $\ell$. Lo mismo vale para eventos en el alfabeto de $C'$, donde estos eventos van a estar restringidos en $C \not\downarrow_H^{hotSwap} C'$ antes de *hotSwap*. Volviendo a nuestro ejemplo, $C$ tiene en su alfabeto *polish(x)* pero no *paint(x)* que si está en el alfabeto de $C'$. El término $C \not\downarrow_H^{hotSwap} C'$ prohíbe la ocurrencia de *paint(x)* antes de *hotSwap* y de la ocurrencia de *polish(x)* luego de *hotSwap*.

### 3.2.2 Controlando la Reconfiguración del Ambiente

Los controladores son actualizados en sistemas porque los objetivos del sistema o los supuestos del ambiente han cambiado. Esto puede deberse a la adopción de supuestos no realistas o inválidos, o porque el ambiente de ejecución (del software, del hardware o factores sociales que afectan al proceso) en donde ejecuta el controlador ha cambiado: una API que provee servicios al controlador necesita ser cambiada, un sensor o actuador no está funcionando con la precisión deseada, etc. En el caso de nuestro ejemplo, el cambio requerido en la configuración del ambiente es que la herramienta de pulido no se necesita más pero sí se necesita una herramienta de pintado. Este cambio involucra desinstalar el driver de la pulidora e instalar el driver de la herramienta de pintado.

Un cambio en la ejecución de la configuración del ambiente, típicamente, necesita ser manejado y coordinado con el comportamiento general del sistema que está satisfaciendo los objetivos del sistema. Por lo tanto, el controlador que maneja la actualización dinámica necesita ser capaz de controlar cuando la reconfiguración

del ambiente puede ocurrir, y debe indicar el cambio en un punto particular donde toda la actualización pueda garantizar ser correcta. Notar que esto último garantiza cambios de ambiente planeados, donde el controlador actualizador elige cuando reconfigurar el ambiente.

Modelamos, a continuación, un controlador que es capaz de cambiar su configuración dinámicamente con el término $C' \| (E \lightning_R^{reconfigure} E')$ suponiendo que *reconfigure* está incluido en el alfabeto de comunicación de $C'$. Esto significa que, al contrario que *hotSwap*, *reconfigure* es controlado por el controlador $C'$. Notar que de esta forma estamos encapsulando un posible procedimiento complejo de reconfiguración con muchos pasos, en un evento atómico de reconfiguración. ¿Podría una reconfiguración atómica no ser una suposición razonable?, Si, y sería posible modelarlo como se menciona en [Taj+10; Alv+17], dentro de $E'$, incluyendo los pasos de la reconfiguración, y además, incluyéndolo como parte del problema de control.

La relación $R$, configura el estado inicial de $E'$ en relación al estado actual de $E$ al momento de reconfigurar (*reconfigure*). Una posible opción para $R$ es mapear todos los estados de $E$ a un estado inicial de $E'$ modelando una inicialización fija del ambiente. Notar que no exigimos que $R$ esté definida para todo estado de $E$; esto nos da un método de restringir cuando $C'$ está habilitado para reconfigurar.

Es importante notar que $R$ puede introducir no-determinismo al mapear un estado de $E$ con muchos estados de $E'$. Esto es una característica importante que es utilizada para modelar el caso de estudio del Railcab de [Ghe+12]. El RailCab, explicado en detalle en el capítulo 5, propone un escenario en el cual la infraestructura de rieles han sido mejorada con sensores adicionales. El sistema actual del RailCab recibe mensajes en varios hitos a medida que se acerca a cruzar el fin de la via: *endOfTrunkSection*, *lastBrake* y luego *noReturn*. Su objetivo está expresado en términos de acciones que debe realizar entre diferentes hitos. El nuevo sistema empieza a recibir un mensaje adicional (*approachingCrossing*) por un nuevo sensor introducido que se ubica entre los hitos de *endOfTrunkSection* y *lastBrake*. La figura 3.3 (ignorando las

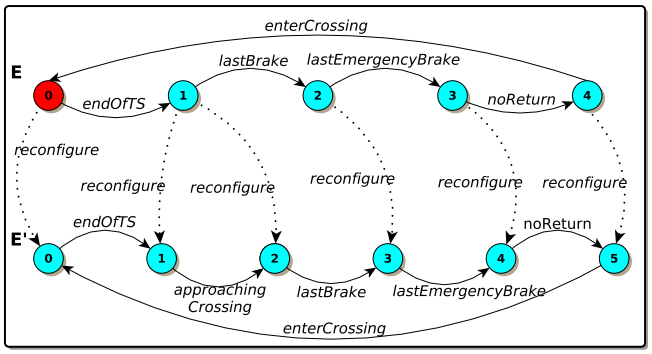


**Fig. 3.3:** Supuestos del ambiente $(E \not\downarrow_R^{reconfigure} E')$ para la actualización dinámica del caso de estudio del RailCab [Ghe+12]. Las transiciones de reconfiguración con linea punteada indican como los viejos supuestos son mapeados hacia los nuevos como es definido en la realación $R$.

transiciones de linea punteada) muestra una porción del modelo del ambiente para la especificación del RailCab actual (arriba) y nueva (abajo).

El no-determinismo es introducido en el mapping de $E$ a $E'$ en la configuración del RailCab en el estado en el ambiente actual que modela que el RailCab está entre *endOfTrunkSection* y *lastBrake* (estado $1$ del modelo de arriba en Figura 3.3) y los dos estados en el nuevo modelo del ambiente que describe el hecho de que el RailCab está entre *endOfTrunkSection* y *lastBrake* (estados $1$ y $2$ en el modelo de abajo en Figura 3.3). Debido a que el evento *approachingCrossing* no está siendo monitoreado en $E$, la manera correcta de modelar la transición a $E'$ es sin suponer si *approachingCrossing* sucedió o no. De hecho, en la Figura 3.3 se muestra la combinación de los dos ambientes a través de una acción de interrupción *reconfigure* y una relación ($R = \{(0,0), (1,1), (1,2), (2,3), (3,4), (4,5)\}$).

Como nota metodológica, como con *hotSwap*, el alfabeto de $E \not\downarrow_R^{reconfigure} E'$ es la unión de los alfabetos de $E$ y $E'$. Esto incluye los eventos en el alfabeto de comunicación de $E$ y no en los de $E'$ (respectivamente en $E'$ y no en $E$) están siendo restringidos para su ocurrencia luego (respectivamente antes) de *reconfigure*.

### 3.2.3 Criterio de Correctitud de la Actualización de Controladores

Ahora formalizaremos qué significa que una actualización dinámica de controladores sea correcta. En particular, ¿cual debería ser el comportamiento esperado por el sistema cuando un controlador $C$ que está garantizando una especificación (en la forma de supuestos del ambiente actuales $E$, requerimientos actuales $\Box G$ con $G$ una formula proposicional LTL, $d$ una fluent definition, y un conjunto de eventos controlables $A$) es reemplazado por un controlador nuevo que debería satisfacer una nueva especificación (similarmente, de la forma de $E'$, $\Box G'$, $d'$ y $A'$) bajo un requerimiento de transición $T$?

Como input del criterio de correctitud, pedimos una especificación de como el estado de $C'$ va a configurarse en base a los estados de $C$, (i.e. la relación $H$) y tambien como el estado del ambiente reconfigurado $E'$ está influenciado por el estado de $E$ (i.e. la relación $R$).

Usamos el siguiente término para modelar el comportamiento de actualizar $C$ con $C'$ de forma tal que $C'$ pueda cambiar el ambiente $E$ por $E'$ vía *reconfigure*: $(C \lightning_H^{hotSwap} C') \parallel (E \lightning_R^{reconfigure} E')$. *stopOldSpec*, *startNewSpec* y *reconfigure* deberían estar solamente en el alfabeto de $C'$, y así, definir que sólo el nuevo controlador indicará cuando la vieja especificación deja de valer, desde cuando la nueva empieza a garantizarse y cuando el ambiente puede ser reconfigurado. Como explicamos anteriormente, en esta sección, *hotSwap* no debería estar en el alfabeto de $C$, $C'$, $E$ ni $E'$. $H$ debería estar definida para cada estado de $C$. Además, $C \lightning_H^{hotSwap} C'$ debería ser legal para $E \lightning_R^{reconfigure} E'$ con respecto a $(A_u, \widehat{A_u})$ donde $A_u = A \cup A' \cup \{stopOldSpec, startNewSpec, reconfigure\}$ es el conjunto de eventos controlables de la actualización, y, $\widehat{A_u} = \widehat{A} \cup \widehat{A'}$ es el conjunto de eventos no controlables de la actualización. En otras palabras, $C$ y $C'$ no deben bloquear eventos monitoreables ($\ell \in \widehat{A_u}$) ni tampoco podrían ejecutar eventos controlables que $E$ y $E'$ prohíbe. Por último, nótese que *hotSwap* es un evento interno de $C \lightning_H^{hotSwap} C'$ y no restringiremos la ejecución de este evento (i.e. *hotSwap* $\notin A_u \cup \widehat{A_u}$).

A continuación, definimos la formula FLTL que modela el comportamiento esperado del sistema $(C \lightning_H^{hotSwap} C') \| (E \lightning_R^{reconfigure} E')$:

**Definición 3.2.1.** (Objetivo de una Actualización Dinámica de Controladores) *Sean $\Box G$ y $\Box G'$ los objetivos actuales y nuevos de un sistema que realizará una actualización dinámica de controladores, y $G$ y $G'$ son fórmulas FLTL proposicionales que no incluyen stopOldSpec, startNewSpec, reconfigure, ni tampoco hotSwap. Sea $T$ una fórmula FLTL modelando el requerimiento de transición que puede predicar sobre los eventos stopOldSpec, startNewSpec, y reconfigure, pero no sobre hotSwap.*

*Definimos $G_u$, el objetivo de una actualización dinámica de controladores como la conjunción de las siguientes fórmulas FLTL:*

1. $G$ **W** *stopOldSpec*
2. $T$
3. $\Box(startNewSpec \implies \Box G')$
4. $\Box(hotSwap \implies (\Diamond stopOldSpec \ \wedge \Diamond reconfigure \wedge \Diamond startNewSpec))$

La primera fórmula exige que los viejos objetivos $G$ valgan hasta que el controlador indique *stopOldSpec*. Recordar que si el momento en el cual los viejos objetivos dejan de valer necesitan ser restringidos, se debe utilizar los requerimientos de transición $T$ para tal fin. La segunda fórmula afirma que los requerimientos de transición $T$ deben valer. Es esperado que estos prediquen sobre *stopOldSpec* y *startNewSpec* ya que estos eventos deben restringirse basándose en qué es considerado, por el usuario, un estado seguro para actualizar. La tercer fórmula simplemente exige que los nuevos objetivos valgan desde el punto en el cual el controlador indica *startNewSpec*. Esto forzará al controlador a solamente ejecutar este evento cuando puede garantizarse $G'$. La última fórmula exige al controlador que una vez que suceda *hotSwap*, debe progresar en búsqueda de la ocurrencia de los eventos *stopOldSpec*, *startNewSpec* y *reconfigure*.

El objetivo anterior para el problema de ADC debe ser evaluado con la extensión de fluent definition entre $d$ y $d'$ (Definición 2.2.1). La llamaremos $d_u$. Notar que la

unión de $d$ y $d'$ garantiza que la interpretación de $G$ y $G'$ no cambie sobre $E$ y $E'$ respectivamente. La siguiente definición pone todas las partes juntas:

**Definición 3.2.2.** (Criterio de Correctitud para la Actualización Dinámica de Controladores) *Sea $\mathcal{P}$ una tupla $(\mathcal{E}, C, \mathcal{E}', C', T, H, R, d_u)$ donde, $C$ y $C'$ son LTS que modelan respectivamente el controlador actual y el nuevo controlador tal que el alfabeto de comunicación de $C$ no contiene stopOldSpec, startNewSpec, reconfigure, ni hotSwap, y el alfabeto de comunicación de $C'$ contiene stopOldSpec, startNewSpec y reconfigure pero no contiene hotSwap; $\mathcal{E} = (E, \Box G, d, A, \widehat{A})$ y $\mathcal{E}' = (E', \Box G', d', A', \widehat{A'})$ son, respectively, la vieja y la nueva especificación de sistemas para una actualización dinámica de controladores, donde $G$ y $G'$ son fórmulas proposicionales FLTL que no predican sobre stopOldSpec, startNewSpec, reconfigure, ni hotSwap; $d_u$ es una extension de fluent definitions de $d$ y $d'$ como está definido en Definición 2.2.1; $T$ es una fórmula FLTL modelando el requerimiento de transición, que puede predicar sobre stopOldSpec, startNewSpec y reconfigure, pero no sobre hotSwap; $H$ y $R$ son relaciones que definen como $C$ y $E$ son interrumpidas para ser remplazadas por $C'$ y $E'$ respectivamente; Sea $G_u$ la fórmula definida como en Definición 3.2.1;*

*Decimos que $\mathcal{P}$ es una ADC correcto si todas de las siguientes valen:*

1. $(C \not\downarrow_H^{hotSwap} C') \| (E \not\downarrow_R^{reconfigure} E') \models_{d_u} G_u$.
2. $(C \not\downarrow_H^{hotSwap} C') \| (E \not\downarrow_R^{reconfigure} E')$ *es libre de deadlock.*
3. *Todo estado de $C$ está vinculado por $H$.*
4. $(C \not\downarrow_H^{hotSwap} C')$ *es un LTS legal para* $(E \not\downarrow_R^{reconfigure} E')$ *con respecto a* $(A_u, \widehat{A_u})$.

Para los requerimientos informales de actualización dinámica de controladores introducidos en la Sección 3.1, las reglas II) a V) están capturadas en las reglas 1 a 4 de la definición 3.2.1. La regla I) está capturada por el hecho de que *hotSwap* no es parte del alfabeto de $C$, $C'$, $E$ ni $E'$, y $H$ está definida para todo estado alcanzable en $C$. Por lo tanto, en el término $(C \not\downarrow_H^{hotSwap} C') \| (E \not\downarrow_R^{reconfigure} E')$, *hotSwap* no está restringido y puede pasar una sola vez.

Durante toda la tesis, siempre pensamos enfoques automáticos que producen la solución correcta en vez de producir enfoques de construir y después verificar. Esto puede formalizarse de la siguiente manera:

**Definición 3.2.3.** (Problema de Síntesis de ACD) *Sea $C$ un controlador para la vieja especificación $\mathcal{E}$, y $\mathcal{E}'$ la nueva especificación, $T$ los requerimientos de transición, $R \subseteq (S_E \times S_{E'})$ una relación, y $d_u$ una extensión de fluent definitions de $d$ y $d'$.*

*Una solución para el problema de ADC es un par $(C', H)$ tal que $(\mathcal{E}, C, \mathcal{E}', C', T, H, R, d_u)$ es una actualización dinámica de controladores correcta.*

Nótese que el problema de ADC puede o no tener solución. La existencia de una solución indica que el controlador resultante de resolver el problema va a garantizar $G_u$ La validez de los supuestos $E$ y $E'$ es responsabilidad del ingeniero. En general, la no existencia de una solución proviene de combinaciones restrictivas de especificaciones ($E$, $G$, $E'$, $G'$ y $T$) y falta de controlabilidad del controlador de los eventos del ambiente.

La no existencia de una solución al problema de control puede surgir en tres diferentes situaciones. La primera es que el nuevo objetivo bajo los nuevos supuestos del ambiente no es lograble por el controlador. Por ejemplo, no hay ninguna forma de manejar los productos para satisfacer la especificación del nuevo proceso de producción. En este caso, la nueva especificación debe ser corregida.

La segunda razón es que no sea posible garantizar que la transición vaya a satisfacer $T$. Por ejemplo, pidiendo en $T$ que la nueva especificación empiece a valer inmediatamente es imposible ya que existe un caso (o más) donde un producto parcialmente procesado no puede ser consistente con la nueva especificación (e.g. un producto ya pulido). En este caso, el requerimiento de transición debe ser mas débil.

Finalmente, la no existencia de solución puede deberse a la imposibilidad de garantizar que la actualización va a *eventualmente* ocurrir. Por ejemplo, exigir que la cadena de montaje esté vacía antes de cambiar no puede ser garantizado por el controlador

ya que éste no controla el evento *in(x)*. En este caso, el requerimiento de transición necesita ser cambiado para permitir actualizaciones con una cadena de montaje no vacía.

## 3.3 Propuesta de Solución

En esta sección proponemos una solución al problema de síntesis de actualización dinámica de controladores como fue formulado en la Definición 3.2.3. La solución está basada en plantear el problema de síntesis de ADC como un problema de control LTS (Definición 2.3.2). Dado un problema de síntesis ADC (definido por $\mathcal{E}$, $C$, $\mathcal{E}'$, $T$, $R$, y $\overline{d_u}$), mostramos como construir un problema de control LTS $(E_u, G_u, \overline{d_u}, A_u)$ tal que su solución $C_u$ puede ser usada para construir una solución de la forma $(C', H)$ de un problema de síntesis de ADC.

Primero mostraremos como construir $E_u$ ($\overline{d_u}$ y $A_u$ se construyen de manera trivial) y luego explicaremos como extraer $C'$ y $H$ de $C_u$.

### 3.3.1 Modelo del Ambiente

El ambiente $E_u$ debe modelar la reconfiguración del ambiente de $E$ a $E'$ y tambien cuando el controlador $C_u$ debe reaccionar al evento *hotSwap*, o cuando puede utilizar los eventos *stopOldSpec*, *startNewSpec* y *reconfigure*. Además, un requerimiento clave para $E_u$ es que debe garantizar que el controlador resultado pueda ser intercambiado en el nuevo sistema en cualquier momento, independientemente del estado actual del controlador que será reemplazado.

A continuación, describimos como construir $E_u$ como un LTS con tres fases. La primer fase está diseñada para permitir el intercambio, y finaliza cuando *hotSwap* sucede. La segunda fase modela el comportamiento del ambiente $E$, terminando cuando el evento *reconfigure* sucede. La última fase modela el comportamiento del

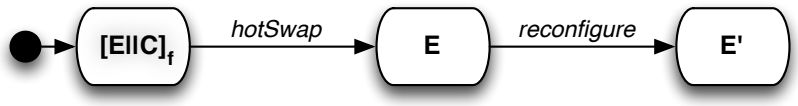


**Fig. 3.4:** Esquema informal de las tres fases del ambiente $E_u$. Primero como un ambiente controlado por $C$ pero no controlable por $C'$ ($[C\|E]_f$), luego como ambiente no controlado que se comporta como $E$ y finalmente como un ambiente no controlado que se comporta como $E'$.

nuevo ambiente $E'$. Una representación informal de estas tres fases de $E_u$ se muestra en la Figura 3.4.

Cuando $C_u$ es construido necesitamos que exhiba un comportamiento equivalente al de $C$ hasta el punto en que *hotSwap* sucede. Esto se logra haciendo que primero $E_u$ emule al sistema actual ($E_u\|C$) y garantizando que $C_u$, cuando sea computado, no restrinja su comportamiento tras debilitar el control que $C$ ejerce sobre $E$: hicimos a todos los eventos en $E_u\|C$ no controlable. Por lo tanto, en esta primera fase, $C_u$ no tiene otra opción mas que simplemente monitorear $E_u\|C$. (Recuerde las definiciones de LTS control, Definición 2.3.2 en donde en controlador no puede restringir eventos no controlables). Consecuentemente, la primer fase de $E_u$ puede definirse como $[E\|C]_f$ donde $f$ es una función de reetiquetado que mapea todos los eventos $\ell \in A$ a eventos frescos que no están en $A_u = A \cup A' \cup \{\mathit{stopOldSpec}, \mathit{startNewSpec}, \mathit{reconfigure}\}$.

La segunda fase empieza cuando *hotSwap* sucede. A este punto la actualización de controlador debe empezar a restringir el comportamiento de $E$ para garantizar una correcta transición donde se satisface la nueva especificación y $C$ ya no está activo. Por lo tanto, la segunda fase puede definirse con $E$, mientras que la transición entre la primera y segunda fase está modelado con el manejador de interrupciones $\lightning$ con etiqueta *hotSwap*. Luego, las primeras dos fases pueden capturarse con el siguiente término: $[E\|C]_f \lightning_I^{hotswap} E$, donde $I$ mapea un estado $(e, c)$ de $[E\|C]_f$ a un estado $e$ de $E$.

El evento *reconfigure* mueve $E_u$ desde la segunda fase a la tercera, donde el estado del nuevo ambiente es descrito utilizando la función propuesta por el usuario $R$

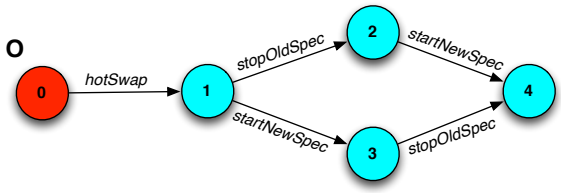


**Fig. 3.5:** LTS que define la ocurrencia de los eventos *stopOldSpec* y *startNewSpec*. Antes del evento *hotSwap* ninguno de los eventos *stopOldSpec* o *startNewSpec* puede suceder. Luego de este, pueden suceder en cualquier orden.

que mapea estados en $E$ a estados de $E'$. Las tres fases de $E_u$ puede entonces describirse como $([E\|C]_f \lightning_I^{hotSwap} E \lightning_R^{reconfigure} E')$, como se muestra informalmente en la Figura 3.4.

Finalmente, para simplificar la especificación de $G$, $G'$ y $T$, y para permitir una escritura precisa de $G_u$, introduzco en $E_u$ los eventos *stopOldSpec* y *startNewSpec*. Estos pueden dispararse una sola vez en $E_u$ luego de *hotSwap* y en cualquier momento antes o después de *reconfigure*. Esto se logra fácilmente al componer al término anteriormente introducido el LTS $O$ dibujado en la Figura 3.5 con alfabeto $\{hotSwap, stopOldSpec, startNewSpec\}$. Como *reconfigure* no está en el alfabeto de $O$ no es necesario restringir su ocurrencia. En conclusión, el modelo del ambiente $E_u$ se define de la siguiente manera:

**Definición 3.3.1.** (Ambiente para el problema de control) *Sea $C$ el controlador actual, $A$ el conjunto de eventos controlados por $C$, $E$ y $E'$ el viejo y el nuevo ambiente, $R \subseteq (S_E \times S_{E'})$ una relación propuesta por el usuario.*

*El* Ambiente para el Problema de Síntesis de ADC *es un LTS definido como $E_u \triangleq ([E\|C]_f \lightning_I^{hotSwap} E \lightning_R^{reconfigure} E')\|O$ donde $I$ es una función parcial tal que $((e,c),e) \in I$ si y sólo si $(e,c) \in S_{E\|C}$, $O$ es el LTS dibujado en la Figura 3.5 con alfabeto $\{hotSwap, stopOldSpec, startNewSpec\}$, y $f$ es la función que reetiqueta todos los eventos $\ell \in A$ a eventos frescos $\overline{\ell} \notin A$.*

### 3.3.2 Resolviendo la Síntesis de ADC con control de LTS

Para lograr una solución correcta al problema de ADC, primero tenemos que definir y resolver un problema de control de LTS (Definición 2.3.2), y luego, extraer un controlador $C'$ y una relación $H$.

Una solución al problema de control LTS puede ser construido usando el ambiente $E_u$ como está definido en Definición 3.3.1; los objetivos $G_u$ como están definidos en Definición 3.2.1; el conjunto de eventos controlables $A_u$ definidos como la unión de los eventos controlables $A$, $A'$ y {*stopOldSpec*,*startNewSpec*,*reconfigure* }; y el conjunto de eventos no controlables $\widehat{A_u}$ definido como la unión de los eventos no controlables $\widehat{A}$, $\widehat{A'}$ más el evento *hotSwap*. Incorporamos *hotSwap* al conjunto de eventos no controlables para poder producir un controlador que no restrinja su ejecución. Las lineas 2 a 6 del Algoritmo 1 muestra como generar estos elementos.

Notar que las trazas en $C$ que garantizan $G$ no van a satisfacer $G_u$ cuando sean reetiquetadas por $f$ en $E_u$ (recordar $f$ de Definición 3.3.1). Para hacer que las trazas en $[E\|C]_f$ sean aceptadas por $G_u$, es necesario modificar levemente la extensión de fluent definitions $d_u$. Usamos la extensión de fluent definitions $\overline{d_u}$ donde cada fluent definition en $d_u$ es cambiado de la siguiente manera: si un fluent está definido como $\langle I, T, Init\rangle$ en $d_u$, entonces, estará definido como $\langle\ I \cup \{f(\ell)\ |\ \ell \in I\}$, $F \cup \{f(\ell)\ |\ \ell \in F\}$, $Init\ \rangle$ en $\overline{d_u}$. La construcción de $\overline{d_u}$ a partir de $d_u$ genera una extensión de fluent definitions de $d$ y $d'$ porque para cada fluent definition en $d_u$ agrego eventos a sus conjuntos de eventos inicializadores y terminadores que no estan en el alfabeto de $d$ ni de $d'$. Por lo tanto, las condiciones de una extensión de fluent definitions de la Definición 2.2.1 se mantiene. El primer *foreach* del Algoritmo 1 (lineas 7 a 12) muestra como se construye $\overline{d_u}$ a partir de $d_u$.

Si existe una solución $C_u$ al problema de control LTS con especificación $(E_u, G_u, \overline{d_u}, A_u, \widehat{A_u} \cup \{\textit{hotSwap}\})$, se garantiza que $C_u$ tiene un conjunto de estados, todos alcanzables desde el estado inicial, que son bisimilares a $[C]_f$ hasta la ocurrencia de *hotSwap* (ver Sección 3.3.4). De hecho, $C_u$ tendrá tres fases: la primera que es

**ALGORITHM 1:** Pseudocódigo que extrae la solución al problema de síntesis de ADC ($\mathcal{S}$) a partir de la solución de problema de control LTS con especificación $\mathcal{E}_u$.

**Input** : $\mathcal{E}$, $C$, $\mathcal{E}'$, $T$, $R$, $d_u$, $f$
**Output** : $(C'_u, H)$ como solución a $\mathcal{S}$

**DynamicUpdateOfControllers** $(\mathcal{E}, C, \mathcal{E}', T, R, d_u, f)$
  $E_u \leftarrow$ buildEnv($\mathcal{E}$, $C$, $\mathcal{E}'$, $R$, $f$); // Definición 3.3.1
  $G_u \leftarrow$ buildGoal($\mathcal{E}$, $T$, $\mathcal{E}'$); // Definición 3.2.1
  $A_u \leftarrow$ {*stopOldSpec*, *startNewSpec*, *reconfigure* };
  $A_u \leftarrow A_u \cup \mathcal{E}.A \cup \mathcal{E}'.A$;
  $\widehat{A_u} \leftarrow \mathcal{E}.\widehat{A} \cup \mathcal{E}'.\widehat{A}$ ;
  $\overline{d_u} \leftarrow \emptyset$;
  **foreach** $\langle I, F, Init \rangle \in d_u$ **do**
    $\overline{I} \leftarrow I \cup \{f(\ell) \mid \ell \in I\}$;
    $\overline{F} \leftarrow F \cup \{f(\ell) \mid \ell \in F\}$;
    $\overline{d_u} \leftarrow \overline{d_u} \cup \{\langle \overline{I}, \overline{F}, Init \rangle\}$
  **end**
  $\mathcal{E}_u \leftarrow (E_u, G_u, \overline{d_u}, A_u, \widehat{A_u} \cup \{\textit{hotSwap}\})$;
  $C_u \leftarrow$ solve($\mathcal{E}_u$); // Definición 2.3.2
  **if** $C_u$ *is null* **then**
    **return** *DCU has no solution*
  $H \leftarrow \emptyset$;
  **foreach** $c \in states(C)$ **do**
    $q \leftarrow$ getBisimilarStateUpToHotSwap($c$, $C_u$);
    $c'_u \leftarrow$ getSuccessor($q$, *hotSwap*, $C_u$);
    $H \leftarrow H \cup \{(c, c'_u)\}$;
  **end**
  $C'_u \leftarrow$ removeStatesBeforeHotSwap($C_u$);
  **return** $(C'_u, H)$

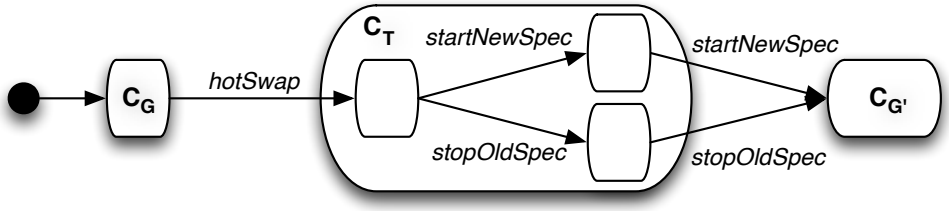


**Fig. 3.6:** Dibujo informal de las tres fases del controlador $C_u$. Primero comportándose exactamente como $C$, y por lo tanto, garantizando $\Box G$ ($C_G$), luego controlando el período de transición ($C_T$), y luego cuando ambos eventos *stopOldSpec* y *startNewSpec* han sucedido, garantiza $\Box G'$ ($C_{G'}$). Notar que, por simplicidad, *reconfigure* no está dibujado.

bisimilar a $[C]_f$, la segunda en la cual se maneja la transición de una especificación a la otra (teniendo en cuenta $\Box G$, $\Box G'$, $T$ y la reconfiguración), y la tercer parte en la cual ambos eventos *stopOldSpec* y *startNewSpec* han sucedido y $C_u$ está simplemente dedicado a satisfacer $\Box G'$ (ver Figura 3.6).

La primera fase permite relacionar los estados de $C$ con estados de $C_u$ siguiendo la definición de $H$: como $[C]_f$ es bisimilar a la parte de $C_u$ previa a *hotSwap*, podemos afirmar que, por cada estado $c \in S_C$ hay al menos un estado $q \in S_{C_u}$ tal que $c$ y $q$ son bisimilares. Además, la transición $(q, \mathit{hotSwap}, c'_u)$ debe existir en $C_u$ y es única porque todo estado antes de *hotSwap* tiene este evento no controlable habilitado en $E_u$. $H$ debe mapear $c$ con $c'_u$. Las lineas desde 17 a 22 en Algoritmo 1 muestra el pseudocódigo para construir $H$ donde el método **getBisimilarStateUpToHotSwap** devuelve solo un estado $q$ y **getSuccessor** devuelve el estado alcanzable luego de transicionar por *hotSwap* desde $q$.

Habiendo definido $H$, construimos $C'_u$ a partir de $C_u$. Notar que $H$ vincula todos los estados en $C$ a estados en la segunda fase de $C_u$ (ver $C_T$ en Figura 3.6). Esto significa que los estados en la primera fase de $C_u$ nunca serán visitados en $C \not\downarrow_H^{\mathit{hotSwap}} C_u$. Por lo tanto, construimos $C'_u$ para que sea la porción de $C_u$ que no incluye los estados de la primera fase. Esto nos asegura que $C'_u$ no tiene una replica completa de $C$ dentro de él, y así evitar que los controladores crezcan mucho a través de actualizaciones sucesivas. El método llamado **removeStatesUpToHotSwap** de la linea 23 en Algoritmo 1 realiza el procedimiento descrito en este párrafo.

### 3.3.3 Complejidad

Resolviendo un problema de control LTS (definido como en Definición 2.3.2 en Capítulo 2) para una propiedad arbitraria FLTL es 2EXPTIME complete [Pnu77]. Es fácil ver que si $T$ es una propiedad de safety, entonces, $G_u$ puede ser codificado como una propiedad de obligación (i.e. disjunción entre aserciones de safety y reachability, $\bigwedge_{i=1}^{n}(\Box S_i \vee \Diamond R_i)$). Problemas de control LTS con objetivos en la forma de obligaciones puede ser resuelto en tiempo linear con respecto al tamaño de $E_u$ para modelos de ambientes determinísitico. Para ambientes no determinísiticos, una construcción de subconjuntos especializados pueden ser usados para producir una versión determinística, sin embargo, una explosión exponencial puede ocurrir dependiendo del grado de no-determinisimo [Cio+17]. El mismo precio se paga para permitir observabilidad parcial (en esta tesis asumo que todos los eventos no controlables son observables por el controlador) puede similarmente ser reducido a un problema determinístico con el mismo costo.

Extendimos la herramienta de síntesis [D'I+08] para permitir especificaciones de un problema de ADC con propiedad de transiciones de safety ($T$), la construcción automática de un problema de control LTS como el de la Definición 2.3.2, la resolución de este problema y el extracción de una solución $(C'_u, H)$ para el problema de ADC previamente introducido. Los casos de estudios descritos en el Capítulo 5 fueron resueltos usando esta herramienta. Tanto los casos de estudio como la herramienta está disponible en [Myt] y una breve explicación de cómo utilizar la herramienta puede leerse en el Capítulo 4.

### 3.3.4 Correctitud y Completitud

A continuación presentamos y probamos la correctitud y completitud de la técnica. Por correcto nos referimos a que el Algoritmo descrito en la Sección 3.3.2 produce una solución correcta al problema de síntesis de ADC. Por completitud nos referimos

a que si el algoritmo no otorga una solución, es porque no hay una solución al problema de ADC.

**Teorema 3.3.1.** *Sea $\mathcal{S}$ un problema de síntesis de ADC con especificación $(\mathcal{E}, C, \mathcal{E}', T, R, d_u)$, $E_u$ definido como en la Definición 3.3.1 usando la función $f$ para reetiquetar, $G_u$ definido como en la Definición 3.2.1, $A_u$ la unión de los viejos y nuevos eventos controlables más los eventos especiales stopOldSpec, startNewSpec, y reconfigure, y $\widehat{A_u}$ la unión de los viejos y nuevos eventos no controlables.*

[*Correctitud*] *Si $C_u$ es la solución al problema de LTS control con especificación $\mathcal{E}_u = (E_u, G_u, \overline{d_u}, A_u, \widehat{A_u} \cup \{hotSwap\})$, entonces, construir $(C'_u, H)$ como en el Algoritmo 1 es una solución de $\mathcal{S}$.*

[*Completitud*] *Si $\mathcal{S}$ tiene una solución, entonces, el Algoritmo 1 devuelve una solución.*

Para la siguiente prueba usaremos $\lightning^{hs}$ y $\lightning^{rec}$ como macro para $\lightning^{hotSwap}$ y $\lightning^{reconfigure}$ respectivamente.

*Prueba de Correctitud.* Debemos probar que el Algoritmo 1 devuelve $(C'_u, H)$ que es solución a $\mathcal{S}$. De las lineas 17 a 23 en Algoritmo 1, $C'_u$ y $H$ están definidas de la siguiente manera: $C_u = ([C_u]_{cut} \lightning^{hs}_Q C'_u)$ y $(c, c'_u) \in H$ sii $\exists q \cdot (q, c'_u) \in Q$ y $[C_u]_{cut}(q) \sim [C]_f(c)$ con $cut(\ell) = \ell$ sii $\ell \neq$ *hotSwap* (ver Figura 3.7 para un mejor entendimiento). Luego, por Definición 3.2.2 debemos probar que:

(1) $(C \lightning^{hs}_H C'_u) \parallel (E \lightning^{rec}_R E') \models_{d_u} G_u$.
(2) $(C \lightning^{hs}_H C'_u) \parallel (E \lightning^{rec}_R E')$ es libre de deadlocks
(3) Todo estado de $C$ está vinculado por $H$.
(4) $(C \lightning^{hs}_H C'_u)$ es un LTS legal para $(E \lightning^{rec}_R E')$ con respecto a $(A_u, \widehat{A_u})$.

Para los puntos (1) y (2) primero vamos a mostrar que:

$$C_u \| E_u \sim ([C]_f \lightning^{hs}_H C'_u) \ \| ([E]_f \lightning^{hs}_{Id} E \lightning^{rec}_R E').$$

Luego, usando que $C_u \| E_u \models_{\overline{d_u}} G_u$ y $C_u \| E_u$ es libre de deadlocks, vamos a probar los puntos (1) y (2) por contradicción.

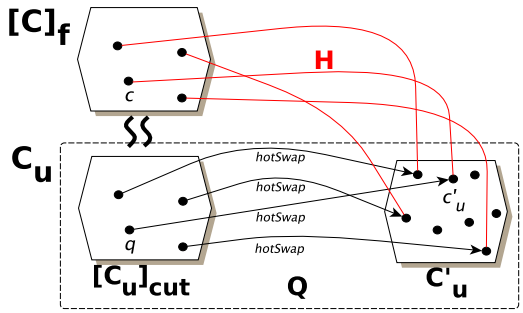


**Fig. 3.7:** Soporte para prueba. Dibujo informal de $C_u = ([C_u]_{cut} \not\downarrow^{hs}_{Q} C'_u)$. Muestro que para los estados $c$ y $q$ tal que $[C_u]_{cut}(q) \sim [C]_f(c)$, entonces, $H$ vincula a $c$ con un estado $c'_u$ en $C'_u$ de acuerdo con la relación $Q$.

Primero necesitamos mostrar que $C_u$ puede siempre ser descompuesto en $([C_u]_{cut} \not\downarrow^{hs}_{Q} C'_u)$: la descomposición es posible porque *hotSwap* está habilitado en todos los estados de $C_u$ previos a la ocurrencia de *hotSwap* (ver Lema 3.3.1). Además, es también fácil mostrar que $[C_u]_{cut} \sim [C]_f$ (ver Lema 3.3.2). Por lo tanto:

$$C_u \sim ([C]_f \not\downarrow^{hs}_{H} C'_u)$$

donde al reemplazar $[C_u]_{cut}$ por $[C]_f$ necesitamos cambiar $Q$ por $H$ (ver Figura 3.7).

Ahora, vamos a trabajar con el ambiente $E_u$. Por Definición 3.3.1, $E_u \triangleq ([E\|C]_f \not\downarrow^{hs}_{I} E \not\downarrow^{rec}_{R} E') \|O$, y como $E$ y $C$ son LTS deterministicos, por Propiedad 2.1.2:

$$E_u \sim (([E]_f \| [C]_f) \not\downarrow^{hs}_{I} E \not\downarrow^{rec}_{R} E') \ \| \ O$$

Notar que $O$ es un observador que restringe la ocurrencia de *stopOldSpec* y *startNewSpec* previo a *hotSwap* y solo permite que estos eventos sucedan una vez. $C_u$ también posee ese comportamiento ya que es una solución para el problema de control $\mathcal{E}_u$. Siendo ambos $C_u$ y $O$ determinisiticos, $C_u \| O$ es bisimilar a $C_u$, y por lo tanto:

$$C_u \| E_u \sim ([C]_f \not\downarrow^{hs}_{H} C'_u) \ \| \ (([E]_f \| [C]_f) \not\downarrow^{hs}_{I} E \not\downarrow^{rec}_{R} E')$$

Por Propiedad 2.1.3 tenemos que:

$$C_u \| E_u \sim (([C]_f \| [E]_f \| [C]_f) \; \lightning^{hs}_{H'} \; C'_u) \;\; \| \;\; (([C]_f \| [E]_f \| [C]_f) \; \lightning^{hs}_{I'} \; E \; \lightning^{rec}_{R} \; E')$$

Como $[C]_f$ es deterministico tenemos que $[C]_f \| [C]_f \sim [C]_f$, por lo tanto:

$$C_u \| E_u \sim (([C]_f \| [E]_f) \; \lightning^{hs}_{H'} \; C'_u) \;\; \| \;\; (([C]_f \| [E]_f) \; \lightning^{hs}_{I'} \; E \; \lightning^{rec}_{R} \; E')$$

Aplicando la Propiedad 2.1.3 nuevamente, y por la definición de $I$ en Definición 3.3.1, tenemos que:

$$C_u \| E_u \sim ([C]_f \; \lightning^{hs}_{H} \; C'_u) \; \| ([E]_f \; \lightning^{hs}_{Id} \; E \; \lightning^{rec}_{R} \; E')$$

Ahora empezaremos a probar el punto (1) por contradicción.

Supongamos que $(C \;\; \lightning^{hs}_{H} \;\; C'_u) \; \| \; (E \;\; \lightning^{rec}_{R} \;\; E') \not\models_{d_u} G_u$.

Por Lema 3.3.3 sabemos que $(C \;\; \lightning^{hs}_{H} \;\; C'_u) \; \| \; (E \;\; \lightning^{rec}_{R} \;\; E')$ es bisimilar a $(C \;\; \lightning^{hs}_{H} \;\; C'_u)$ $\| \; (E \;\; \lightning^{hs}_{Id} \;\; E \;\; \lightning^{rec}_{R} \;\; E')$, y por lo tanto, $(C \; \lightning^{hs}_{H} \; C'_u) \; \| \; (E \;\; \lightning^{hs}_{Id} \;\; E \;\; \lightning^{rec}_{R} \; E') \not\models_{d_u} G_u$.

Ahora, consideremos una traza $\pi \in (C \; \lightning^{hs}_{H} \; C'_u) \; \| \; (E \;\; \lightning^{hs}_{Id} \;\; E \;\; \lightning^{rec}_{R} \; E')$ tal que $\pi \not\models_{d_u}$ $G_u$. Notar que, $\pi$ es ó bien una traza tal que $\pi = \ell_0, \ldots, \ell_n, \ldots$ y *hotSwap* $\notin \pi$, ó $\pi$ $= \ell_0, \ldots, \ell_i, \mathit{hotSwap}, \ell_{i+1}, \ldots$.

Puedo construir una traza $\overline{\pi} \in ([C]_f \;\; \lightning^{hs}_{H} \;\; C'_u) \; \| \; ([E]_f \;\; \lightning^{hs}_{Id} \;\; E \;\; \lightning^{rec}_{R} \;\; E')$ tal que, ó bien $\overline{\pi} = f(\ell_0), \ldots, f(\ell_n), \ldots$ y *hotSwap* $\notin \overline{\pi}$, ó $\overline{\pi} = f(\ell_0), \ldots, f(\ell_i), \mathit{hotSwap}, \ell_{i+1}, \ldots$. Luego, $\overline{\pi} \not\models_{\overline{d_u}} G_u$ porque $\overline{d_u}$ cambia el valor de un fluent cuando sucede $f(\ell)$ si y sólo si $d_u$ cambia el valor cuando sucede $\ell$.

La traza $\overline{\pi}$ introduce una contradicción al suponer que $(C \; \lightning^{hs}_{H} \; C'_u) \| (E \; \lightning^{rec}_{R} \; E') \not\models_{d_u}$ $G_u$.

Para probar el punto (2), i.e., que $(C \ \lightning_H^{hs} \ C'_u) \parallel (E \ \lightning_R^{rec} \ E')$ es libre de deadlock, podemos seguir un razonamiento similar. Supongamos que no es libre de deadlock. Entonces, sabemos que este término tiene un estado que no tiene transiciones de salida. Luego podemos definir una traza $\pi \in (C \ \lightning_H^{hs} \ C'_u) \parallel (E \ \lightning_R^{rec} \ E')$ que alcanza a este estado. Esta traza tiene un equivalente $\overline{\pi}$ en $\in ([C]_f \ \lightning_H^{hs} \ C'_u) \parallel ([E]_f \ \lightning_{Id}^{hs} \ E \ \lightning_R^{rec} \ E')$ que es un absurdo.

Para probar el punto (3) necesitamos encontrar para cada $c \in S_C$, un estado $c'_u$ en $C'_u$ vinculado por $H$. Por lo tanto, necesitamos probar que existe un estado $q \in S_{[C_u]_{cut}}$ tal que $(q, c'_u) \in Q$ y $[C_u]_{cut}(q) \sim [C]_f(c)$ (ver Figura 3.7).

Sea $\pi$ una traza finita en $[C]_f$ que alcanza el estado $c$. Luego, $\pi$ también puede ejecutarse en $[C_u]_{cut}$ alcanzando un estado $q$ porque $[C_u]_{cut} \sim [C]_f$ (ver Lema 3.3.2), y por lo tanto, $[C_u]_{cut}(q) \sim [C]_f(c)$. Lo que queda probar es que hay un estado $c'_u$ tal que $(q, c'_u) \in Q$. En otras palabras que en $C_u$ hay una transición con etiqueta *hotSwap* de $q$ a $c'_u$. Esto puede demostrarse fácilmente: como $q \in [C_u]_{cut}$ y $C_u \sim ([C_u]_{cut} \ \lightning_Q^{hs} \ C'_u)$ con $Q$ una función total (Lema 3.3.1), por definición de $\lightning$, la transición debe existir.

Para probar el punto (4), sabemos que para cualquier traza $\pi$ en $(C \ \lightning_H^{hs} \ C'_u) \parallel (E \ \lightning_R^{rec} \ E')$, que alcanza un estado $(c, e)$, vale lo siguiente.

Si *hotSwap* $\notin \pi$ entonces $c \in S_C$ y $e \in S_E$. Luego, $(c, e)$ es alcanzable en $C \| E$ via $\pi$. Como $C$ es una solución a $\mathcal{E}$, entonces $C$ es un LTS legal para $E$ con respecto a $(A, \widehat{A})$ y $c$ y $e$ son legales con respecto a $(A, \widehat{A})$. Esto muestra que son también legales con respecto a $(A_u, \widehat{A_u})$, porque $C$ y $E$ no tienen transiciones etiquetadas con $A_u \setminus A$ o $\widehat{A_u} \setminus \widehat{A}$.

Si *hotSwap* $\in \pi$ entonces podemos construir $\overline{\pi}$ que alcanza al estado $(c, e) \in C_u \| E_u$ a travez de, simplemente, reetiquetar cada etiqueta $\ell$ que aparece en $\pi$ antes de *hotSwap* con $f(\ell)$. Como $C_u$ es un LTS legal para $E_u$, entonces, $c$ y $e$ son legales con

respecto a $(A_u, \widehat{A_u} \cup \{\textit{hotSwap}\})$. Como no hay una transición *hotSwap* habilitada desde estos estados, entonces $c$ y $e$ son también legal con respecto a $(A_u, \widehat{A_u})$. □

*Prueba de Completitud.* Suponiendo una solución $(C', Y)$ a $\mathcal{S}$, debemos probar la existencia de un LTS $X$ que es una solución al problema de control $\mathcal{E}_u$. Sea $X$ un LTS tal que $X \triangleq ([E\|C]_f \not\downarrow^{hs}_{Y'} C')$ donde $((e, c), c') \in Y'$ sii $(c, c') \in Y$. En otras palabras que:

(a) $X\|E_u \models_{\overline{d_u}} G_u$.
(b) $X\|E_u$ es libre de deadlocks, y
(c) $X$ es un LTS legal para $E_u$ con respecto a $(A_u, \widehat{A_u} \cup \{hotSwap\})$,

Primero probamos el punto (a):
Como $(C', Y)$ es una solución a $\mathcal{S}$ podemos afirmar que

$$(C \not\downarrow^{hs}_{Y} C') \parallel (E \not\downarrow^{rec}_{R} E') \models_{d_u} G_u,$$

donde $Y$ vincula cada estado de $C$ a estados de $C'$.

Vamos a usar como antes el Lema 3.3.3 para descomponer $E$:

$$(C \not\downarrow^{hs}_{Y} C') \parallel (E \not\downarrow^{hs}_{Id} E \not\downarrow^{rec}_{R} E') \models_{d_u} G_u.$$

donde $Id$ es la relación identidad.

Luego, usando la Propiedad 2.1.3, tenemos que:

$$(E\|C \not\downarrow^{hs}_{Y'} C') \parallel (E\|C \not\downarrow^{hs}_{I} E \not\downarrow^{rec}_{R} E') \models_{d_u} G_u.$$

donde $((e, c), e) \in I$ sii $(e, e) \in Id$.

Podemos reetiquetar los eventos controlables en $E\|C$ con $f$ y seguir garantizando $G_u$ siempre y cuando cambie la fluent definition $d_u$ por $\overline{d_u}$. Por lo tanto, lo siguiente vale:

$$([E\|C]_f \; \lightning^{hs}_{Y'} \; C') \;\|\; ([E\|C]_f \; \lightning^{hs}_{I} \; E \; \lightning^{rec}_{R} \; E') \models_{\overline{d_u}} G_u.$$

Como antes, tenemos que $([E\|C]_f \; \lightning^{hs}_{Y'} \; C') \|O$ es bisimilar a $([E\|C]_f \; \lightning^{hs}_{Y'} \; C')$ donde $O$ es un observador introducido en la Definición 3.3.1. Esto se debe a que $O$ prohíbe la ejecución de *stopOldSpec* y *startNewSpec* antes de *hotSwap*, y habilita a cada uno a suceder a lo sumo una vez. Por lo tanto, tenemos que:

$$([E\|C]_f \; \lightning^{hs}_{Y'} \; C') \;\|\; O \;\|\; ([E\|C]_f \; \lightning^{hs}_{I} \; E \; \lightning^{rec}_{R} \; E') \models_{\overline{d_u}} G_u.$$

Finalmente, usando la definición de $E_u$ el punto (a) está probado:

$$([E\|C]_f \; \lightning^{hs}_{Y'} \; C') \;\|\; E_u \;\models_{\overline{d_u}}\; G_u$$

La prueba del punto (b) sigue el mismo razonamiento que lo anterior. Empezando por el hecho de que $(C \; \lightning^{hs}_{Y} \; C') \| (E \; \lightning^{hs}_{Id} \; E \; \lightning^{rec}_{R} \; E')$ es libre de deadlock y utilizando las mismas transformaciones de bisimulación, llego a que $([E\|C]_f \; \lightning^{hs}_{Y'} \; C') \| E_u$ es también libre de deadlock.

Probamos el punto (c) de una manera sencilla. Para cualquier traza $\overline{\pi}$ en $X\|E_u$ que llega a un estado $(x, e)$, puedo afirmar lo siguiente:

Si *hotSwap* $\notin \overline{\pi}$ entonces $x \in [E\|C]_f$, y, por definición de $E_u$, tenemos que $e \in [E\|C]_f$. Como $[E\|C]_f$ es un LTS deterministico, entonces, $x = e$. Por lo tanto, tenemos que $\Delta_X(x) = \Delta_{E_u}(e)$. Si esto es cierto, $x$ y $e$ deben ser legal con respecto a cualquier par de conjuntos.

Si *hotSwap* $\in \overline{\pi}$ entonces podemos construir $\pi$ para que alcance el mismo par $(x, e)$ pero en $(C \; \lightning^{hs}_{Y} \; C') \|((E\|C) \lightning^{hs}_{I} E \; \lightning^{rec}_{R} \; E')$, simplemente, reemplazando cada evento

reetiquetado $f(\ell)$ que aparece en $\overline{\pi}$ antes de *hotSwap* por $\ell$. Además, $\pi$ hasta *hotSwap* ejecuta en el término $(E\|C)\lightning_I^{hs}E$ para llegar al mismo estado en $E$ como cuando $\pi$ es ejecutado directamente en $E$. Por lo tanto, $\pi$ alcanza $(x, e)$ en $(C \lightning_Y^{hs} C') \| (E \lightning_R^{rec} E')$. Como $(C \lightning_Y^{hs} C')$ es un LTS legal para $(E \lightning_R^{rec} E')$, entonces, $x$ y $e$ son legales con respecto a $(A_u, \widehat{A_u})$. Como, además, no hay una transición *hotSwap* habilitada a partir de estos estados, entonces, $x$ y $e$ son también legal con respecto a $(A_u, \widehat{A_u} \cup \{\textit{hotSwap}\})$. □

Las pruebas de los lemas que fueron utilizados en la demostración de arriba, son las siguientes:

**Lema 3.3.1.** *Sea $C_u$ una solución al problema de control LTS con especificación $\mathcal{E}_u$, entonces, existe $C'_u$ tal que $C_u \sim ([C_u]_{cut}\lightning_Q^{hotSwap}C'_u)$ con $Q$ una función total.*

*Demostración.* Necesitamos mostrar que $C_u$ puede ser dividido por *hotSwap*. Notar que si existe una solución al problema de control LTS $C_u$, entonces, $C_u$ es un LTS legal para $E_u$ con respecto a $(A_u, \overline{A_u} \cup \{\textit{hotSwap}\})$ (i.e. debe garantizar $G_u$ solo controlando eventos en $A_u$). En particular, *hotSwap* no está en $A_u$ y está habilitado en todo estado de $E_u$ antes de que sucede el evento. Por lo tanto, como $C_u$ no puede recortar las transiciones *hotSwap*, entonces $Q$ es una función total. □

**Lema 3.3.2.** *Sea $C_u$ un resultado del problema de control LTS con especificación $\mathcal{E}_u$, entonces, $[C_u]_{cut} \sim [E\|C]_f \sim [C]_f$.*

*Demostración.* Primero, sabemos que $C_u = ([C_u]_{cut}\lightning_Q^{hotSwap}C'_u)$ utilizando el Lema 3.3.1. Como $C_u$ no controla ningún evento previo a *hotSwap* porque $E_u$ fue definido para que así sea (ver Definición 3.3.1), entonces, $[C_u]_{cut} = [E\|C]_f$. Además, usando Propiedad 2.3.1, sabemos que $E\|C \sim C$, y por lo tanto, $[E\|C]_f \sim [C]_f$. Finalmente, $[C_u]_{cut} \sim [E\|C]_f \sim [C]_f$ □

**Lema 3.3.3.** *Sean $(C \lightning_H^{hs} C'_u) \| (E \lightning_{Id}^{hs} E \lightning_R^{rec} E')$ y $(C \lightning_H^{hs} C'_u) \| (E \lightning_R^{rec} E')$ LTS, entonces, son bisimilares.*

*Demostración.* Notar que la única diferencia entre los dos LTS son la parte inicial de los ambientes. En ambos LTS cuando sucede el evento *hotSwap* se hace un cambio de controlador (de $C$ a $C'_u$), sin embargo, la parte del ambiente es solo intercambiada en el primero de los dos LTS. Este cambio es un cambio en vano, ya que el LTS $E$ es reemplazado mediante la función identidad por el mismo LTS $E$. Este intercambio es igual a no efectuar ningún evento. En el ambiente del segundo LTS, el evento *hotSwap* no forma parte del alfabeto. Por lo tanto, cuando suceda, la parte del ambiente en el segundo LTS queda inmutable generando el mismo efecto que en el primer LTS. □

En esta sección hemos presentado un enfoque correcto y completo, de complejidad lineal, con requerimientos de transición como propiedades de safety, que resuelve problemas de síntesis de ADC convirtiéndolos a un problema de control LTS.

# 4 Soporte de Herramienta

> *Life gives each of us the tools to go around the world.*
>
> — **Juan Manuel Fangio**
> (Automovilista)

La herramienta MTSA es la que utilizamos para poder producir todos los resultados presentados en esta tesis. Primero explicaremos como se sintetiza un controlador que es solución a un problema de control LTS (Definición 2.3.2). Mas adelante explicaremos como se define un problema de síntesis de controlador actualizador introduciendo los valores de entrada de la Definición 3.2.3. Esta herramienta es la utilizada para las aplicaciones que seguirán a continuación (Capítulo 5 y Capítulo 6). Los casos de estudio explorados en cada una de estas aplicaciones pueden descargarse de [Myt]

## 4.1 Sintetizando un Controlador

Lo necesario para realizar un síntesis de controladores es definir un problema de control tal y como fue planteado en la Definición 2.3.2. Para esto, necesitamos explicar como se construye el problema $\mathcal{E} = (E, G, d, A, \widehat{A})$. Necesariamente, debemos construir los siguientes valores de entrada: un ambiente $E$, un conjunto de requerimientos $G$ y un conjunto de acciones controlables $A$. La fluent definition $d$ y el conjunto de acciones no controlables $\widehat{A}$ se infieren a partir de la definición de $G$ y de $A$. Para el primero todos los fluentes que se utilizan en las formulas $G$ van a ser parte de la fluent definition $d$ y los eventos no controlables se definen como aquellos que no están en el conjunto de acciones controlables.

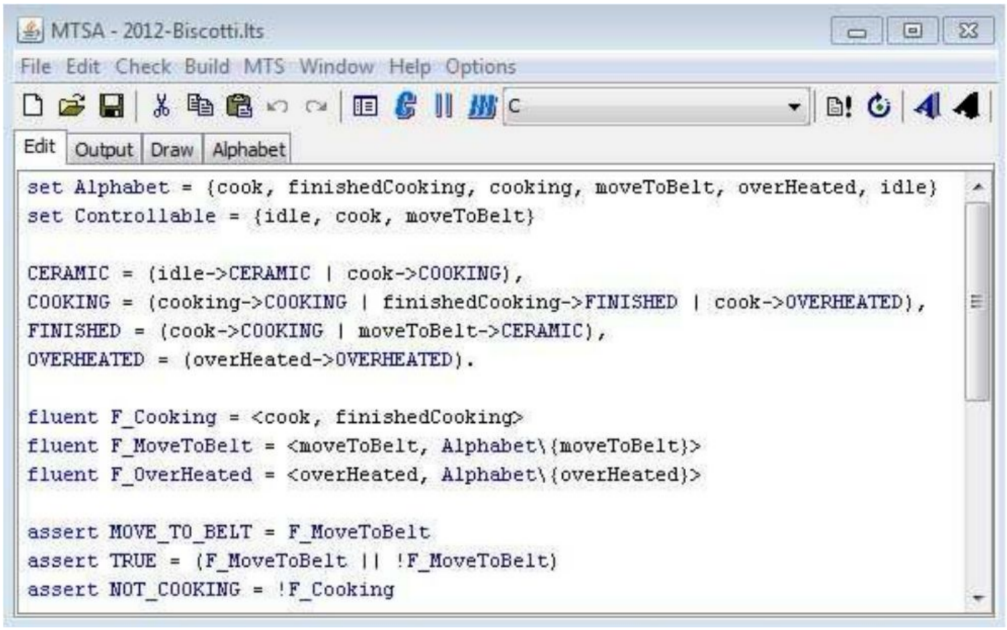


**Fig. 4.1:** Captura de pantalla de la herramienta MTSA donde se muestra la definición de un ambiente utilizando el lenguaje FSP.

### 4.1.1 Modelando el Ambiente

En MTSA los modelos del ambientes se describen con una extensión del lenguaje FSP (Finite State Process). Esta extensión de FSP provista por la herramienta MTSA incluye los operadores tradicionales para describir modelos de comportamiento, como por ejemplo, el prefijo de acción (->), elección (|), composición secuencial (;), y composición en paralelo (||). Además, posee operadores que extienden el lenguaje FSP tradicional como el hiding (\), utilizado para ocultar los eventos de disable y $enableAll$ que introduciremos en el Capítulo 6, entre otros operadores. Para más detalles ver [D'I+08].

La herramienta MTSA integra funcionalidad para construir, analizar y elaborar modelos LTS y provee un ambiente gráfico que apunta a facilitar estas tareas. Los ambientes $E$ se describen en MTSA como un FSP que se compilan a un LTS. En la Figura 4.1, muestro una captura de pantalla de la herramienta en la solapa de edición de FSP.

El código corresponde a un ejemplo de cocción de cerámicas. Inicialmente se define el alfabeto del ambiente tras listar todos los eventos (etiquetas del LTS) que tendrá el ambiente. En la segunda linea se define el subconjunto de eventos del alfabeto que son parte del conjunto de eventos controlables. El resto del código define inicialmente el proceso CERAMIC que permite tanto ejecutar el evento *idle* (haciendo un loop al ir al proceso CERAMIC), o, empezar a cocinar ejecutando el evento *cook* yendo hacia el proceso auxiliar COOKING. Similarmente, COOKING puede hacer un loop mientras el horno está cocinando (etiqueta *cooking*), o bien, puede terminar de cocinar ejecutando el evento *finishedCooking*, continuando la ejecución con el proceso auxiliar FINISHED, o, por última opción, volver a cocinar la cerámica que nos lleva al proceso auxiliar de OVERHEATED. Luego, el proceso FINISHED transiciona o bien por *cook* al proceso COOKING, o bien, por *moveToBelt* al proceso CERAMIC volviendo a empezar. Finalmente, el subproceso OVERHEATED está definido para que sea un estado donde solo se permite ejecutar el evento overHeated haciendo un loop.

### 4.1.2 Modelando los Requerimientos del Controlador

Para modelar requerimientos, la herramienta posee un conjunto de palabras clave para que FSP pueda soportar la síntesis de controladores LTS y la síntesis de problemas de actualización dinámica de controladores.

En la Figura 4.1 puede verse la definición de fluents y la definición de ciertas propiedades que predican sobre los fluents. Por ejemplo, la propiedad **MOVE_TO_BELT** va a ser válida si y sólo si el fluent **F_MoveToBelt** esta prendido (i.e. si sucedió el evento *moveToBelt* y no sucedió otro evento todavía). Por otro lado, la propiedad **NOT_COOKING** va a ser válida si y sólo si el fluent **F_Cooking** está apagado (i.e. desde el estado inicial hasta que suceda el evento *cook* y volverá a prenderse cuando suceda el evento *finishedCooking*).

La especificación del problema de control de LTS se completa con el código de la Figura 4.2, donde se muestra la definición del **controllerSpec** G1. Esta especificación

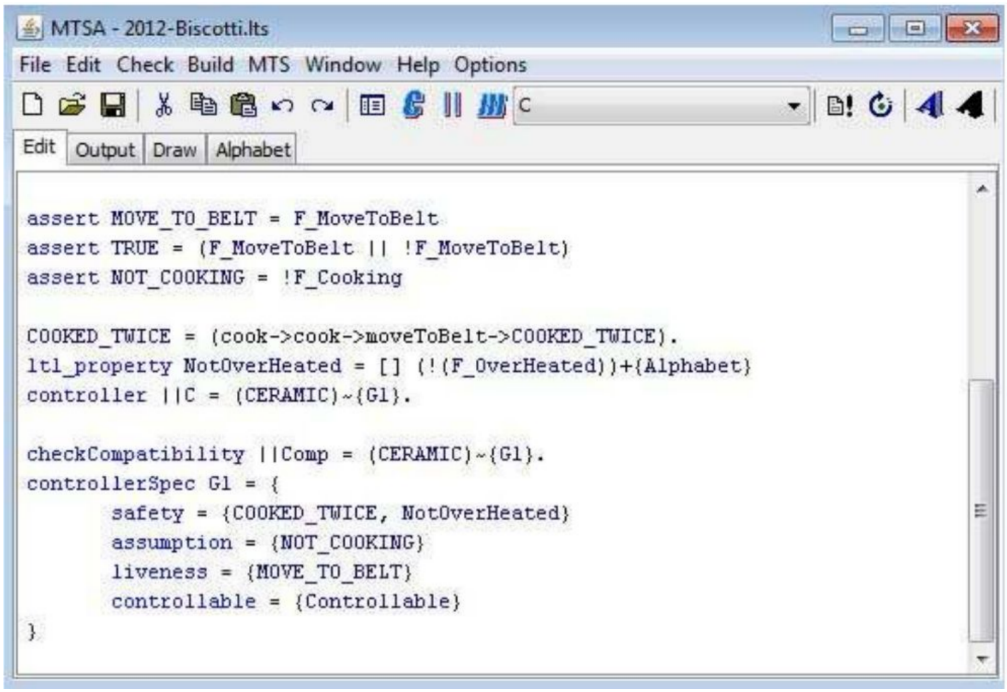


**Fig. 4.2:** Captura de pantalla de la herramienta MTSA donde se muestra la de un problema de control LTS.

define la otra parte relativa al problema de control, definiendo los requerimientos $G$ que se introducen por las palabras clave **safety**, **assumption** y **liveness**. Con **safety** se puede declarar propiedades de safety, ya sea mediante una propiedad LTL como **NotOverHeated** o mediante una expresión FSP que compila a un LTS como **COOKED_TWICE**. Las palabras clave **assumption** y **liveness** definen propiedades del tipo $(\square\Diamond assumptions) \rightarrow (\square\Diamond guarantees)$. Pueden las assumptions no estar declaradas y se asume que $assumptions = \top$ Por último, G1 posee la palabra clave **controllable** que define el conjunto de eventos controlables $A$.

### 4.1.3 Ejecutando la Síntesis

La especificación del problema de control de LTS se completa con el uso de la palabra clave **controller** con la siguiente sentencia.

```
controller ||C = CERAMIC~{G1}.
```

donde C es el nombre del controlador resultante luego de realizar la síntesis. Para ejecutar la síntesis, alcanza con buscar C en el menu dropdown de la barra superior

de MTSA. Si queremos obtener la composición paralela entre C y el ambiente E ($C \| E$), tendremos que escribir la siguiente sentencia.

```
||ControllerAndEnvironment = (C || CERAMIC).
```

Cabe destacar que con CERAMIC y G1 sólo hemos definido el ambiente $E$, los requerimientos $G$ y el conjunto de eventos controlables $A$. El conjunto de eventos no controlables $\widehat{A}$ y la fluent definition $d$ quedan definidos implícitamente por el complemento de los eventos controlables ($\widehat{A} = A_E \setminus A$) y por el conjunto de fluents utilizados para definir $G$, respectivamente. Con estos cinco elementos tenemos todos los valores de entrada de un problema de control LTS.

## 4.2 Sintetizando un Controlador Actualizador

Ahora que podemos especificar un problema de control LTS y sintetizar el controlador resultante en la herramienta MTSA, vamos a explicar cómo se debe especificar problema de actualización dinámica de controladores. El resultado de resolver este problema es un controlador $C_u$ que cumple con el criterio de correctitud introducido en la Sección 3.2.3.

### 4.2.1 Modelando un Problema de Actualización Dinámica

En esta tesis, definimos los valores de entrada para el problema de síntesis de actualización dinámica de controladores. Estos valores podrían simplificarse a el viejo controlador ($E \| C$), el mapping entre $E$ y $E'$ ($R$), los requerimientos viejos ($G$), los requerimientos nuevos ($G'$) y los requerimientos de transición ($T$). En la herramienta MTSA este problema se especifica con el código de la Figura 4.3.

Los elementos en rojo son los valores de entradas presentados en la tesis. Por ejemplo, con la sentencia oldController debemos especificar cual es el controlador viejo, componiendo en paralelo el ambiente con el controlador ($E \| C$). Los elementos en azul informa cual es el tipo de dato que hay que especificar en cada sentencia.

```
updatingController UpdCont = {
    oldController = E||C : Process,
    mapping = E ----reconfigure ----> E' : FSP,
    oldGoal = G : controllerSpec,
    newGoal = G' : controllerSpec,
    transition = T : ltl_property,
    nonblocking
}
```

**Fig. 4.3:** Especificación de un problema de actualización dinámica de controladores en la herramienta MTSA.

Por lo tanto, con oldController debemos colocar un proceso que es resultado de una composición en paralelo, mientras que, el requerimiento de transición $T$ debe ser una propiedad expresada con la sentencia ltl_property. Finalmente, $G$ y $G'$ deben ser una especificación de un problema de control mediante la sentencia controllerSpec. El más complejo de todos los valores de entrada es el mapping. Pedimos que el usuario escriba un FSP donde explique como se realiza este mapeo, es decir, presentar un FSP que se comporta originalmente como el ambiente $E$ y todo estado debe tener una forma de saltar a $E'$ mediante el evento *reconfigure*. El LTS que se obtiene por compilar el FSP representa perfectamente el mapeo de estado de $E$ a estados de $E'$.

### 4.2.2 Solución al Problema de Actualización Dinámica

La salida de la herramienta, como hemos introducido en el Capítulo 3, puede ser o bien un mensaje de que el controlador no pudo ser computado (“There is no controller”) o un controlador ($C_u$) en forma de LTS que representa la estrategia de actualización de controladores o de reconfiguración de procesos de negocio. Como es esperado, el LTS suele tener un tamaño bastante grande, y por lo tanto, es probable que la solapa “draw” no pueda mostrarnos esta salida. Afortunadamente, el controlador obtenido puede ser animado escribiendo la siguiente linea

```
||UPDATE_CONTROLLER = UpdCont.
```

y corriendo la composición en paralelo para UPDATE_CONTROLLER (buscar este nombre en el menú del dropdown). Para animar el controlador se debe presionar el

botón de animación en la barra superior de la herramienta (botón con una A azul en la Figura 4.2). En el menú que se despliega pueden ver una lista de todos los eventos que pueden ejecutarse desde el estado inicial. Esta lista va a actualizarse cada vez que se elige un evento. Esto se debe a que el estado actual del LTS cambió y son otros los eventos habilitados para su ejecución.

Otra validación que puede hacerse para obtener mayor confianza del resultado obtenido por la síntesis, es la evaluación de una propiedad. Previo al computo de la síntesis, podemos escribir una propiedad mediante la palabra reservada “assert”. La sintaxis es la siguiente:

```
assert "nombre de propiedad" = <<formula LTL>>
```

Luego de escribir la propiedad deseada (pudiendo ser esta una propiedad de safety o de liveness), y luego de computar la síntesis de actualización de controladores, podemos pedirle a la herramienta que verifique que la propiedad escrita vale en el modelo LTS obtenido. Para esto debemos utilizar el menú superior de la herramienta:

```
Check >> LTL property >> "nombre de propiedad"
```

# 5 Aplicación a Orquestaciones de Software

> *The difficult thing about searching phrases on internet is to verify their authenticity.*
>
> — **Luis Federico Leloir**
> (Médico, bioquímico y farmacéutico)

En este capítulo detallamos un conjunto de escenarios que utilizaremos para evaluar la técnica desarrollada. El objetivo es poder analizar estos casos de estudio en diversos aspectos. Obviamente, nos interesa mostrar que la solución es una que concuerda con las expectativas del usuario, pero además, nos interesa mostrar cual es el rendimiento de la técnica, en cuanto a la cantidad de estados y transiciones que debe manejar. Otro aspecto de esta técnica que deseamos mostrar es la utilización de los requerimientos de transición como valor de entrada indispensable para el usuario. Dependiendo del dominio, el usuario podría querer generar diferentes estrategias para actualizar/reconfigurar un controlador/proceso de negocio.

Todos los casos de estudios que utilizaremos a continuación son sistemas que han sido diseñados como un software orquestado. Esto significa, que tiene una pieza de software que orquesta una cantidad finita de eventos el cual llamaremos controlador. Utilizaremos la técnica de Actualización Dinámica de Controladores del Capítulo 3. El propósito de esta validación es poder mostrar aplicabilidad y generalidad del enfoque con respecto al trabajo existente usando casos de estudio tomados de la literatura. Como ya he discutido, la complejidad del problema de síntesis de ADC es lineal y, por lo tanto, la escalabilidad no es de extrema preocupación.

Todos los casos de estudio fueron ejecutados usando una extensión de la herramienta MTSA [D'I+08], que es, a su vez, una extensión de LTSA [MK06] (ver Capítulo 4) Como LTSA, MTSA soporta nativamente especificaciones de LTS [Kel76] y propie-

dades usando una álgebra de proceso de texto y FLTL [GM03]. La herramienta también soporta síntesis de controladores para problemas de control de SGR(1) que son estrictamente más complejos computacionalmente que los problemas de control de ADC [D'I+10]. La herramienta fue extendida para soportar definiciones de problemas de control de ADC, computando automáticamente $A_u$, $G_u$ y $E_u$, y resolviendo el problema de control ADC. La versión extendida de la herramienta y los casos de estudios pueden encontrarse en [Myt].

Los casos de estudio seleccionados son el Rail Cab [Ghe+12], la Planta de Energia [PLM+13], el protocolo GSM-oriented [ZC06], MetaSocket [ZC05] y la cadena de montaje [LL95], que permiten una comparación con los trabajos más parecidos a los presentados en esta tesis. Finalmente, elegimos una variedad de controladores a actualizar que comandan las actividades de un UAV (Unnamed Aerial Vehicle) inspirado en [Kon+04]. Estos casos de estudio apuntan a mostrar una aplicación punta a punta de la técnica, desde la síntesis a la ejecución en una infraestructura adaptable.

Para cada caso de estudio experimentamos diferentes situaciones. Fijando la vieja y nueva especificación para actualizar y explorar el uso de varios requerimientos de transición. Además de los requerimientos de transición específicos de dominio definidos por cada caso de estudio, usamos dos requerimientos de transición default: $T_\top$ y $T_\emptyset$ en todos los casos de estudio.

$T_\top = \top$ es un requerimiento de transición no restrictivo que permite, que deje de valer la vieja especificación en cualquier momento, y también, un período en el cual todo está permitido antes de que empiece a valer la nueva especificación. Como los controladores que construyo son activos para alcanzar liveness (i.e., desean lograr que la nueva especificación valga tan pronto como sea posible), el período en el cual todo está permitido es minimal en cuanto a que el controlador solo hará lo suficiente para alcanzar un estado desde el cual la nueva especificación puede garantizarse. El segundo requerimiento default que usé prohíbe que sucedan eventos durante el período en el cual ninguna especificación vale ($T_\emptyset =$

$\Box((\neg \mathit{NewSpecStarted} \wedge \mathit{OldSpecStopped}) \Rightarrow \neg Event)$, donde $Event$ es una disjunción de todos los eventos).

Para todos los casos de estudio y requerimientos de transición también construimos un controlador que es safe pero no garantiza progreso en cuanto al cambio de especificación. El propósito de esto es comparar los controladores safe y live contra los que son safe pero no live. Esto nos permite asegurarnos que los requerimientos de liveness inducen un controlador de actualización que guía al sistema a un estado safe en el cual los requerimientos de transición pueden satisfacerse y que la nueva especificación pueda garantizarse. Así mismo, producir los dos controladores permiten medir al costo adicional de resolver un problema de actualización de controladores dinámico live. Hacemos finalmente una discusión acerca de las diferencias entre los controladores live y no-live para el primer caso de estudio, la cadena de montaje. Para el resto, solamente muestro información cuantitativa: costo computacional y el tamaño de los controladores (ver Tabla 5.1)

## 5.1 Cadena de Montaje

La configuración general para la cadena de montaje, discutida en la Sección 3.1 y basada en [LL95], presenta una automatización industrial en la cual la cadena de montaje está siendo controlada para producir productos. Esta debe ser actualizada para acomodarse a nuevas reglas de producción. Para esto, hemos modelado varios requerimientos de transición y construimos controladores de actualización para cada uno de ellos.

El primer requerimiento de transición fue uno en el cual la nueva especificación debe empezar a valer cuando la cadena de montaje no tiene productos que están siendo procesados ($\Box(\mathit{startNewSpec} \implies Empty)$). Como es de esperarse, la herramienta reporta que no hay un controlador actualizador que puede satisfacer este requerimiento. Esto es razonable ya que *in(x)* es un evento que el controlador no controla: si nuevos productos llegan regularmente, la cadena de montaje podría nunca llegar

a estar vacía y por lo tanto la planta nunca llegaría a un estado en donde las nuevas reglas empiezan a valer.

Un requerimiento de transición más débil es forzar al plan a que siempre cumpla con una de las dos especificaciones ($T_\emptyset$). El resultado es un controlador que demora el cambio de las reglas de producción hasta que todos los elementos que estén en la cadena de montaje estén crudos (i.e., no se les aplicó ninguna herramienta a los productos). Esto es porque en este caso particular, la reglas para procesar productos de las dos especificaciones son incompatibles. Si un producto ha empezado a ser procesado usando las reglas viejas, entonces, es imposible que pueda cumplir con las nuevas. La única estrategia para tratar un producto parcialmente procesado es finalizarlo y demorar el procesamiento de cualquier producto nuevo que puede llegar a ser introducido a la cadena de montaje (*in(x)*).

En la Figura 5.1 se muestra un controlador actualizador para $T_\emptyset$. Como hemos explicado en la Sección 3.3.2, la caja de linea punteadas de la izquierda $C_G$ contiene los estados del controlador de actualización que son estructuralmente equivalente para controlar la producción siguiendo las reglas de producción viejas. La caja de la derecha $C_{G'}$ contiene todos los estados de un controlador actualizador que describe el comportamiento de un controlador una vez que ha finalizado el período de transición entre las especificaciones y garantiza las nuevas reglas de producción. El comportamiento descrito entre las dos cajas describen la estrategia del controlador actualizador para alcanzar un estado en el cual puede efectivamente cambiar la especificación. Nótese que el comportamiento entre las cajas no tiene ciclos, y por lo tanto, garantiza progreso hacia conseguir la actualización.

En contraste, y para enfatizar la importancia de garantizar progreso en actualizaciones de controladores dinámicos, la Figura 5.2 muestra el resultado de construir un controlador con requerimiento de transición $T_\emptyset$ sin el objetivo de liveness para actualización de controladores dinámicos (ítem 4. de la Definición 3.2.1). El controlador safe pero no live permite comportamiento cíclico una vez que el controlador fue intercambiado. Este ciclo, que se muestra en rojo, permite que el controlador

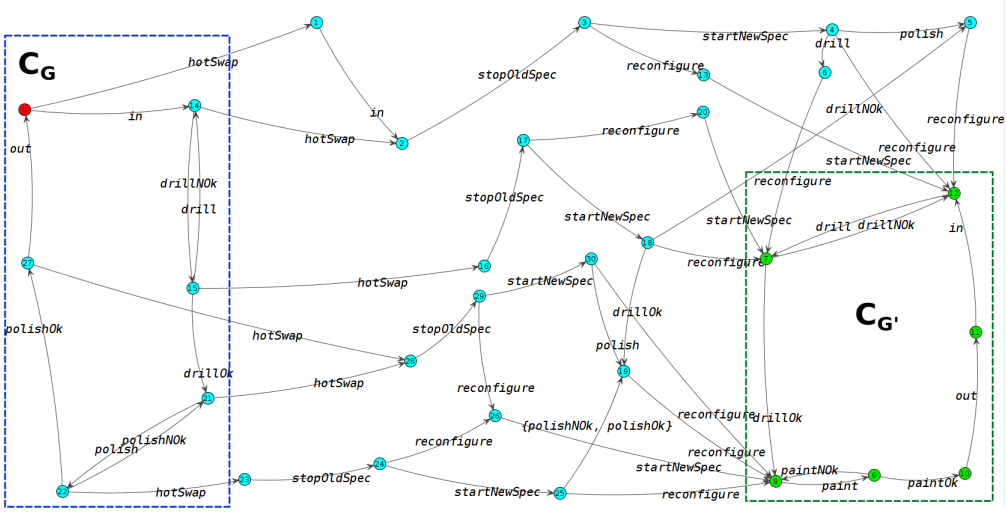


**Fig. 5.1:** Controlador actualizador para un versión reducida de una cadena de montaje (solo dos herramientas) con sus tres fases. $C_G$ y $C_{G'}$ garantizan la vieja y la nueva especificación respectivamente. La parte del medio garantiza progreso entre las dos mientras satisface $T_\emptyset$.

continúe procesando cualquier producto nuevo que entre a la cadena de montaje con las reglas de producción viejas (*in, drill, drillOk, polish, polishOk, out*), logrando así, que nunca se procesen productos usando la nueva especificación.

Como se discutió en la Sección 3.1, estudiamos la utilización de un requerimiento de transición ($T_1$) que especifica que el cambio de reglas de producción podrían ocurrir solo cuando todos los productos de la linea de producción no han sido pulidos. Para lograr este cambio, el controlador actualizador debe continuar produciendo productos que han sido pulidos pero debe parar de procesar productos que no han alcanzado la fase de pulido para asegurarse que eventualmente un estado es alcanzando en el cual $T_1$ vale. Quitar el requerimiento de liveness para estos valores de entradas produce un controlador mucho mas grande (537 estados en vez de 107) que no garantiza que la actualización suceda efectivamente.

Finalmente, computamos el controlador actualizador para $T_2$ (ver Sección 3.1) que permite un período en el cual ninguna de las dos especificaciones valen pero requiere que durante este período los productos sean o bien procesados con la nueva especificación (solo en caso que puedan continuarse para satisfacer las nuevas reglas)

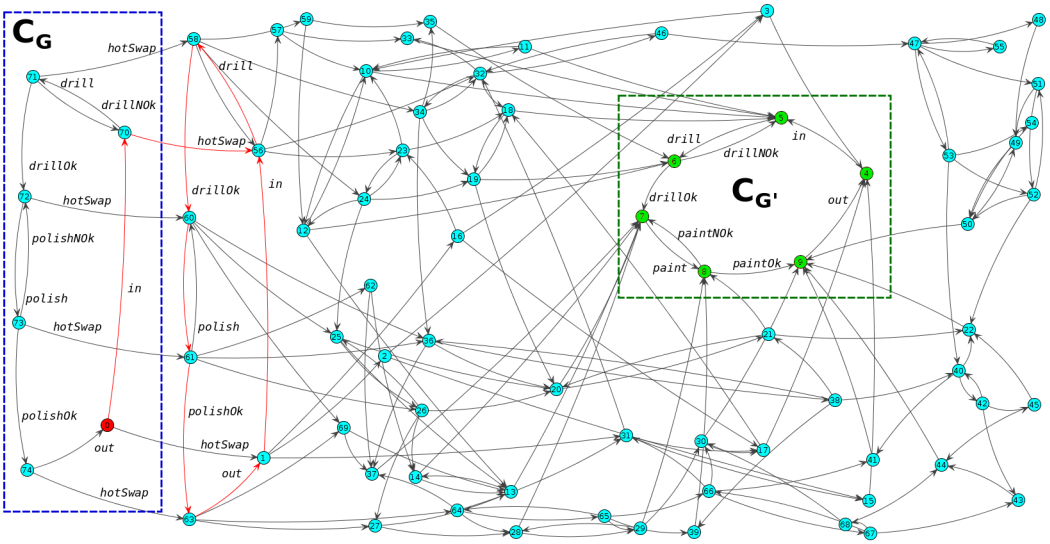


**Fig. 5.2:** Controlador Safe pero no live $C_u^S$ para un versión reducida de una cadena de montaje (sólo con dos herramientas) con sus tres fases. A diferencia de la Figura 5.1, la fase intermedia tiene ciclos (e.g., transiciones en rojo) donde no se cumple la finalización de la actualización.

o descartados en caso contrario. El controlador safe y live es aproximadamente un cuarto del tamaño del controlador que es safe pero no live.

## 5.2 Planta de Energía

En [PLM+13] se discute un Controlador para el sistema de enfriamiento de un planta de energía nuclear. Para poder atender a los servicios de mantenimiento, el controlador existente primero detiene la bomba que enfría la planta y luego del mantenimiento la reinicia. En la especificación del nuevo controlador, ya no se necesita apagar y prender la bomba, y hay un invariante del sistema que propone que la bomba de enfriamiento nunca debe ser detenida indefinidamente. Los autores muestran que si una actualización se realiza de manera ingenua en un estado en el cual el controlador actual ha apagado la bomba y no la ha reiniciado, la planta corre el riesgo de que suceda un incidente muy peligroso ya que el nuevo controlador no es el encargado de reiniciar la bomba. Una forma de satisfacer el invariante del sistema, dicen los autores, es pidiendo a la actualización dinámica que preserve el comportamiento de una actualización offline (i.e. equivalente a actualizar cuando la bomba ha sido reiniciada).

Consideramos tres diferentes problema de control de ADC para este caso de estudio. Tanto con $T_\top$ y $T_\emptyset$ el sistema muestra el comportamiento invalido descrito en [PLM+13]. Utilicé $T = T_\emptyset \wedge \Box(\mathit{startNewSpec} \Rightarrow \mathit{PumpOn})$ para exigir que si durante el período de transición, la bomba esta apagada, entonces la bomba DEBE prenderse antes de que termine el procedimiento de mantenimiento. Este requerimiento evita dejar la bomba apagada de manera inintencionada. Además, este controlador es menos restrictivo que el que es "equivalente a una actualización offline" de [Ghe+12; PLM+13]. No solo $T$ permite actualizar el sistema en estrictamente más estados que la solución propuesta en [Ghe+12], mientras se satisface el requerimiento deseable ($\neg\Diamond\Box\mathit{PumpOff}$), sino que tambien evita la validación de correctitud manual que se solicita en [PLM+13] para un requerimiento de transición mas débil que [Ghe+12].

## 5.3 RailCab

El sistema de RailCab [Pad14] consiste en vehículos autónomos que coordinan el transporte de pasajeros y bienes a demanda. El subsistema discutido en [Ghe+12] hace foco en el control de los RailCabs cuando se aproxima a un cruce. El RailCab puede monitorear eventos como *endOfTrunkSection* y que pasó las zonas de *lastBrake* o *lastEmergencyBrake*. El controlador controla el freno (*brake*) y el freno de emergencia (*emergencyBrake*) y tambien puede recibir respuestas a peticiones que controla como *requestEnter*.

El objetivo del controlador actual y el nuevo es poder garantizar que el RailCab entre al cruce solo si le han garantizado el acceso. Hay también otras restricciones a cerca de cuando se pueden pedir los permisos y cuando cada tipo de freno puede ser aplicado. También hay supuestos del ambiente que supone que las respuestas a los eventos controlables sucedan. Los eventos monitoreables como el ingreso a las zonas y respuestas del sistema estan dibujadas en la Figura 5.3 en letra negra y roja respectivamente. Las acciones controlables se muestran en azul.

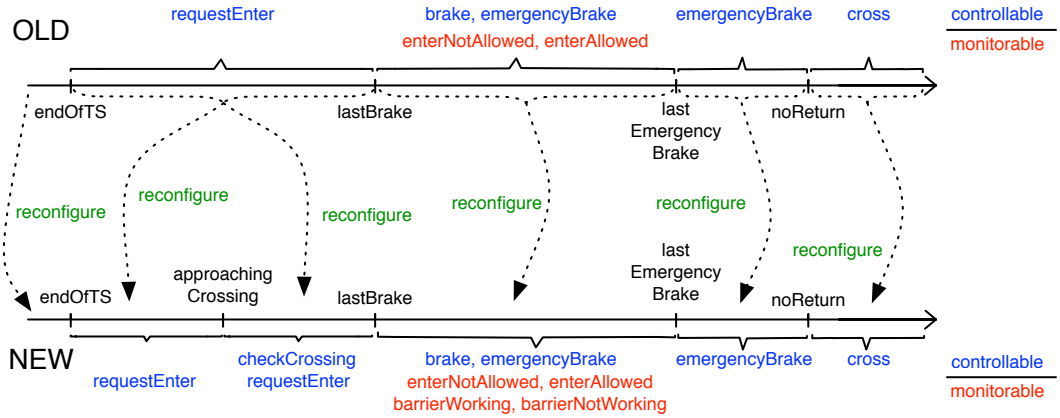


**Fig. 5.3:** Linea de tiempo del Railcab. Zonas en letra negra, eventos controlables en letra azul y eventos monitoreables en letra roja.

La diferencia entre las especificaciones del controlador actual y el nuevo es que en la nueva especificación se introduce un nuevo evento monitoreable y objetivos nuevos. El controlador nuevo monitorea el evento *approachingCrossing* que se supone que será recibido por el RailCab en una zona entre *endOfTrunkSection* y *lastbrake*. El nuevo controlador debe realizar un chequeo adicional (*checkCrossing*) para saber el estado en el que se encuentra la barrera entre *approachingCrossing* y la zona de *lastbrake*.

La dificultad con este cambio de especificación es qué hacer cuando el sistema a ser actualizado ha pasado *endOfTrunkSection* pero no a alcanzado *lastbrake* (ver Figura 5.3). En este escenario, el sistema actual puede pasar la zona de *approaching-Crossing* sin saberlo porque no estuvo monitoreando el evento *approachingCrossing*. Si el sistema fuese actualizado entre estas zonas, el sistema actualizado no puede saber si debe empezar a realizar los chequeos adicionales que son requeridos. Hacerlo sin haber pasado por *approachingCrossing*, violaría la intención de la nueva especificación (i.e. que *checkCrossing* debe ocurrir luego de pasar *approachingCrossing*).

En [Ghe+12] el problema es resuelto simplemente no permitiendo la actualización si es requerido mientras el RailCab recibió la señal de *endOfTrunkSection*. En este caso, la actualización se pospone hasta luego de que el Railcab pase el cruce. Esta demora es insatisfactoria ya que una actualización de seguridad para un sistema crítico no debería ser pospuesto a menos que sea estrictamente necesario.

Usando el enfoque descrito en este artículo, pudimos actualizar el sistema durante el período en el cual el RailCab está en la zona *endOfTrunkSection-lastbrake*. El controlador resultante reconfigurará (*reconfigure*) el sistema para habilitar el monitoreo de los eventos *approachingCrossing* pero seguirá teniendo cierta incertidumbre en cuanto a si el evento *approachingCrossing* fue recibido antes que *lastbrake*. La clave es que si el controlador actualizador luego de reconfigurar recibe la señal de *approachingCrossing*, entonces puede satisfacer la nueva especificación haciendo el chequeo (*checkCrossing*) Caso contrario, si el controlador actualizador recibe *lastbrake* no hay opción más que continuar con la vieja especificación ejecutando un comportamiento similar al de [Ghe+12].

Para producir el controlador actualizador descrito anteriormente, use una relación no-deterministica entre los estados de los modelos del viejo y nuevo ambiente. Esto es, la relación $R$ debe vincular el estado de $E$ que corresponda con la zona de *endOfTrunkSection-lastbrake* a dos estados en $E'$, uno modelando que el RailCab está en la zona *endOfTrunkSection-approachingCrossing* y otro para la zona *approachingCrossing-lastbrake* (ver Figura 3.3).

Resolvimos problemas de control ADC para tres requerimientos de transición distintos. Para $T_\top$, el controlador exhibe un comportamiento no seguro. Usando $T_\emptyset$ el resultado es un controlador que puede realizar una actualización segura en mas estados que la técnica propuesta en [Ghe+12]: si una actualización es requerida mientras el RailCab está entre *endOfTrunkSection* and *lastBrake*. El problema con $T_\emptyset$ es que permite al controlador actualizador intentar un *approachingCrossing* luego de *reconfigure* pero antes de *startNewSpec* lo que en la práctica nos lleva a una violación de la nueva especificación. Por lo tanto, usé $T = T_\emptyset \wedge \Box(\neg \mathit{OldSpecStopped} \Rightarrow \neg CheckCrossing)$ como una posible especificación comprensiva del problema de actualizar un Railcab.

## 5.4 Protocolo GSM-oriented

En [ZC06] presentan un caso de estudio basado en un protocolo orientado a GSM que se utiliza para transferir audio en una red wireless con perdida de paquetes. Los autores describen una actualización entre dos implementaciones de este protocolo que usa diferente estrategias de piggybacking de packetes para mejorar la transferencia de datos. Aumentar la cantidad de piggybacking nos da soporte para manejar redes de alto porcentaje de perdida de paquetes. Consecuentemente, es necesario realizar una actualización dinámica entre controladores que implementen distintas estrategias entre emisor y receptor para adaptar su comportamiento dependiendo la taza de falla de la red.

Un problema surge al actualizar el sistema mientras un mensaje fue transmitido. En esta situación hay un mensaje enviado con la estrategia vieja de piggybacking, pero el receptor, ha sido actualizado, esperando un mensaje usando la nueva estrategia. Estos problemas se exhiben por controladores construidos usando $T_\top$ y $T_\emptyset$. Sin embargo, la solución manual de [ZC06] puede ser sintetizada utilizando $T = T_\emptyset \wedge \Box((\mathit{startNewSpec} \vee \mathit{stopOldSpec}) \implies \neg SendingMessage)$. El controlador resultante tiene las mismas garantías que las descritas en [ZC06], haciendo la salvedad de que en este caso, la estrategia de actualización es construida automáticamente.

## 5.5 MetaSocket

El MetaSocket [ZC05] es un socket que puede ser adaptado para agregar y quitar filtros. En [ZC05] se presentan cuatro combinaciones de configuraciones de filtros: DES64, DES128, DES64COM y DES128COM. Como en [ZC05] generamos un controlador que inicialmente corre en DES64 y luego produce actualizaciones que pueden ser agregar o quitar filtros, de a uno a la vez. Modelamos diferentes escenarios de actualización y ejecutamos varias actualizaciones encadenadas, agregando y removiendo filtros para ver si la técnica produce controladores cada vez más grandes. Como hemos destacado en la Sección 3.3.2, el controlador resultante y

los problemas de control no aumentaron en tamaño a medida que fuimos realizando actualizaciones.

## 5.6 Surveillance

Consideremos un escenario de un UAV (Unmanned Aereal Vehicle) necesita buscar por actividad sospechosa en un área crítica Se requiere que el UAV vuele a una altura entre 40-50 metros y transmitir una foto por cada objetivo potencial mientras evita obstaculos. En baja bateria, el UAV debe volver a cualquier base para recarga. Durante esta misión, se decide que el UAV debe seguir inmediatamente la primera aparición que corresponda a un objetivo especifico en vez de continuar con la búsqueda. La nueva misión requiere que el UAV vuele en un rango de 20 a 30 metros y use un algoritmo de reconocimiento de imagen mejorado. Además, como ahora vuela bajo, el UAV debe poder pronosticar problemas que pueden ocurrir por el clima actual. El nuevo software debe cargarse en el UAV (el nuevo módulo de procesamiento de imagen y el cliente para el servicio de pronostico del clima) junto con el controlador actualizado para alcanzar los nuevos objetivos.

Notar que el no solapamiento de las alturas de vuelo implica que si la actualización sucede durante el vuelo, entonces un *período de transición* no vacío es necesario entre las dos misiones (el descenso desde 40 a 30 metros de altura) en donde ninguna especificación vale.

Produjimos diferentes controladores proponiendo variaciones de las propiedades de transición. Primero construimos requerimientos de transición para replicar la condición de [Ghe+12]. El resultado es un controlador actualizador que fuerza al UAV a volver a la base para actualizar y solo luego de que cambie la misión empieza de vuelta. El requerimiento de transición $T_{\emptyset}$ es un requerimiento levemente más débil que fuerza el cambio de misión cuando está en la base (no necesariamente la base inicial) ya que este es el único estado en el cual los requerimientos de vuelo de las dos misiones son satisfacibles. Por último produje controladores

utilizando requerimientos de transición más débil que permiten actualizar mientras está en vuelo. Primero exigiendo que mientras ninguna especificación está siendo garantizada, el UAV no cambia sus cordenadas $x$ e $y$, produciendo un controlador que evita perderse una actividad sospechoza mientras cambia la altitud: $\Box((\mathit{OldSpecStopped} \wedge \neg \mathit{NewSpecStarted}) \implies \neg move)$. Tambien incluí un requerimiento adicional que fuerza que cuando la especificación cambia no haya fotos de objetivos sospechosos pendientes: $\Box(\mathit{stopOldSpec} \implies \neg PendingTransmissions)$.

## 5.7 Tasa de Consumo de Energía Inesperada

Exploramos un escenario diferente de actualización del plan de misión: misión degradación debido a circunstancias imprevistas. Asumimos una misión original que implica cubrir un área A (es decir, visitar cada ubicación discreta en A) para fines de mapeo.

En grandes regiones discretas, las misiones de cobertura con ubicación explícita el modelado no escala bien, ya que un fluent para cada ubicación es necesario para rastrear si se ha cubierto. En cambio, usamos una alternativa estrategia de abstracción discreta nativa: planificación basada en iteradores. Esto sigue la idea de la planificación basada en sensores en la que se proporciona un sensor (es decir, código) para identificar si una ubicación corresponde a la zona a cubrir.

Suponga que debido a las condiciones del viento o al mal funcionamiento del motor, la tasa de consumo de la batería es más alta de lo predicho por un monitor. En el panel de control esto podría disparar una alarma que indica que el UAV será incapaz de cubrir completamente la región $A$. Simulamos este escenario e hicimos que el usuario interviniera (con el UAV aún en vuelo) produciendo un plan de misión degradado: el usuario elige reducir a la mitad la región original $A$, para tener al menos una región contigua completamente cubierta. Esto se hizo primero definiendo $B$ y luego especificando los nuevos requisitos de la misión, junto con el requisito de transición $T_b = \Box(\mathit{reconfigure} \rightarrow \neg SensingA)$. Este requisito permite que ocurra

la reconfiguración sólo cuando el sensor A está en su estado inicial. El módulo Discretizer genera automáticamente un código Python que implementa el nuevo sensor $B$ y lo carga al UAV que está en vuelo. Mientras tanto, el sintetizador produjo un controlador de actualización que cuando se carga y se hotswapea con el Enactor se detiene la antigua misión (*stopOldSpec*), se ejecuta *reconfigure* desencadenando el enlace del módulo $sensor.B$ al Enactor (y desenlazando al $sensor.A$), y señala el inicio de la nueva misión (*startNewSpec*).

## 5.8 Monitoreo de Incendios

Los UAV se utilizan para ayudar a los bomberos mediante el monitoreo y seguimiento de incendios. Una misión de monitoreo de incendios puede ser tan simple como una vigilancia de incendios [Kuc+17] entre dos lugares muy separados el uno del otro. Tal misión es adecuada para un rotor múltiple con capacidades de vuelo estacionarias.

La misión que simulamos consiste en visitar dos áreas A y B, haciendo un giro lento completo en cada una de ellas para tener una vista de $360°$ de los alrededores. Si el quadcopter tiene una cámara montada a bordo y transmite su video a un estación de monitoreo remoto, un humano puede ver estas imágenes para detectar la presencia de fuego. Ejecutamos esta misión usando un ArduCopter simulado SITL.

Nuestra adaptación es útil en este escenario si el humano necesita una mirada más cercana a cierta área donde se sospecha que hay fuego presente. El usuario puede seleccionar una nueva región para cubrir y generar, con ayuda del Discretizador, el sensor requerido. Para apuntar la cámara a bordo del UAV hacia abajo, el usuario modifica manualmente el código generado para mover la cámara que controla el servo cuando se inicializa el módulo híbrido. La nueva misión consiste en cubriendo el área C y volver al lanzamiento cuando haya terminado.

El requisito de transición es una combinación de $T_b$ adaptada para incluir el sensor B y la capacidad de giro en sus estados iniciales, $T_1$ y $T_2$ para restringir las acciones

entre las especificaciones, y $T_3$ para forzar un reinicio del iterador antes de comenzar el nuevo especificación (para garantizar la cobertura total de la región C):

$T_1 = \Box((OldStopped \wedge \neg NewStarted) \rightarrow \neg SenseOrMove)$

$T_2 = \Box(\textit{reconfigure} \rightarrow NewStarted)$

$T_3 = \Box(\textit{startNewSpec} \rightarrow Reset), Reset = \langle reset, has.next?, \bot \rangle$

donde $SenseOrMove$ es inicialmente falso, se vuelve verdadero cuando el UAV se mueve o sensa (e.g. $takeOff, go$, etc) y vuelve a ser falso con el resto de los eventos.

Como es esperado después de que el sistema carga el código del sensor $C$ y el plan de actualización, el UAV actualiza correctamente su plan de misión, cubre la región $C$ y vuelve a iniciarse.

## 5.9 Resumen de las Experiencias

En total, 21 problemas de síntesis de ADC, en 8 dominios distintos, fueron definidos y resueltos, correspondientes a diferentes elecciones de requerimientos de transición para cada caso de estudio. Además, corrimos el ejemplo del Metasocket con el objetivo de verificar el escenario de múltiples actualizaciones. Mostramos que los controladores resultantes luego de cada actualización no incrementan la cantidad de estados, evitando así controladores que crecen en numero de estados a medida que se aplican múltiples actualizaciones.

Otro de los experimentos fue cubrir varios aspectos de la arquitectura definida en el trabajo [Bra+15]. Los casos de estudio de la Tasa de Consumo de Energía Inesperado y el Monitoreo de Incendios apuntan a lograr un caso de estudio complejo, donde efectivamente un robot funcionando está cumpliendo una misión que es intercambiada por otra debido al cambio de requerimientos. Podríamos decir que la actualización dinámica de controladores de eventos discretos, es una pieza fundamental a la hora de cubrir todos los nodos de la arquitectura definida en [Bra+15].

| Caso de Estudio | Requerimiento de Transición | $\|E\|+\|E'\|$ | $\|d\|+\|d'\|$ | $\|G\|+\|G'\|$ | $\|T\|$ | $\|R\|$ | $\|E_u\|$ | $\|E_u\|\|G_u\|$ | $\|C\|$ | $\|C_u^S\|$ | $C_u^S$ ms | $\|C_u\|$ | $C_u$ ms |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Planta de Energía** | $T_\top$ | 8 | 13 | 10 | 0 | 4 | 16 | 80 | 8 | 88 | 141 | 97 | 294 |
| | $T_\emptyset$ | | | | 8 | | | 66 | | 72 | 153 | 52 | 241 |
| | Deja bomba prendida | | | | 11 | | | 49 | | 56 | 146 | 34 | 319 |
| **Railcab** | $T_\top$ | 34 | 18 | 42 | 0 | 21 | 63 | 728 | 21 | 731 | 482 | 346 | 730 |
| | $T_\emptyset$ | | | | 18 | | | 663 | | 668 | 570 | 276 | 466246 |
| | frena al actualizar si es tarde | | | | 5 | | | 750 | | 744 | 289 | 360 | 763 |
| **Cadena de Montaje** | $T_\top$ | 14 | 21 | 43 | 0 | 7 | 23 | 469 | 9 | 467 | 542 | 189 | 551 |
| | No hay pulidos al actualizar | | | | 3 | | | 419 | | 411 | 270 | 174 | 617 |
| | Quitar pulidos o $G'$ | | | | 5 | | | 798 | | 795 | 440 | 227 | 5961 |
| | Actualizar si vacía | | | | 2 | | | 291 | | 293 | 217 | - | 787 |
| | $T_\emptyset$ | | | | 10 | | | 163 | | 164 | 263 | 103 | 541 |
| | Mantener Spec vieja | | | | 7 | | | 397 | | 396 | 252 | 180 | 459 |
| **Protocolo GSM** | $T_\top$ | 17 | 2 | 2 | 0 | 8 | 25 | 81 | 8 | 76 | 99 | 83 | 117 |
| | $T_\emptyset$ | | | | 13 | | | 83 | | 76 | 866 | 75 | 896 |
| | No enviar mientras actualiza | | | | 6 | | | 78 | | 74 | 123 | 74 | 143 |
| **Surveillance** | $T_\top$ | 60 | 26 | 34 | 0 | 30 | 117 | 3226 | 56 | 3231 | 518 | 1287 | 992 |
| | $T_\emptyset$ | | | | 16 | | | 611 | | 615 | 29960 | 365 | 30057 |
| | No mover mientras actualiza | | | | 7 | | | 2900 | | 2907 | 696 | 1236 | 1005 |
| | No foto mientras actualiza | | | | 7 | | | 1714 | | 1718 | 684 | 1047 | 863 |
| **Tasa de Consumo de Energía** | $T_b$ | 63 | 42 | 31 | 7 | 34 | 187 | 355 | 56 | 3233 | 603 | 1340 | 981 |
| **Monitoreo de Incendios** | $T_b \wedge T_1 \wedge T_2 \wedge T_3$ | 83 | 41 | 35 | 10 | 65 | 123 | 320 | 72 | 3582 | 701 | 1602 | 1009 |

**Tab. 5.1:** Reporte de la síntesis de la actualización dinámica controladores. $|d|$ y $|d'|$ son los números de fluents en las especificaciones vieja y nueva, respectivamente. $|G|$ y $|G'|$ son los números de operadores lógicos en las especificaciones vieja y nueva, respectivamente. $|T|$ es el número de operadores lógicos del requerimiento de transición. $|E_u||G_u|$ es el número de estados de la composición de $E_u$ y la representación de $G_u$ en forma de autómata. El guión (-) indica un problema de control no realizable.

La Tabla 5.1 resume los casos de estudio, el tamaño del modelo de los ambientes que los describe, el tamaño de los controladores resultado safe y live ($C_u$) y el tiempo que llevó computarlos. La tabla también muestra el tamaño y el tiempo de síntesis de los controladores safe pero no live ($C_u^S$) para cada caso de estudio.

La mayoría de los controladores safe y live fueron sintetizados en unos pocos segundos. Vale la pena notar que los controladores de ejemplos más complejos computacionalmente fueron aquellos que tienen un $E_u$ de más de 100 estados. El tiempo máximo de síntesis fue cercano a los 5 minutos.

Notar que los controladores sintetizados sin el objetivo de liveness no garantizaron que la actualización suceda eventualmente. En otras palabras, en todos los casos de estudios, para garantizar que la actualización dinámica sucede eventualmente, se necesita un controlador actualizador que activamente guie al sistema para hacer una transición segura. El costo de computar controladores live comparados con los safe pero no live varía significativamente. En la mayoría de los casos, computar liveness no incrementó demasiado el tiempo versus computar un controlador safe, sin embargo, el caso de estudio del workflow, que es el peor caso, el tiempo de

la síntesis de un controlador live fue 20 veces mas que el de controlador safe, aproximadamente.

Además, note que el algoritmo de síntesis es propenso a intentar alcanzar los objetivos de liveness. En otras palabras, los controladores van a intentar ejecutar una actualización con la menor cantidad de eventos. Esto es muy distinto a controladores que solo tienen requerimientos de safety. Los últimos fueron sintetizados para ser maximales (permitir cualquier transición safe). Esto explica porque el tamaño de los controladores live fueron en promedio 60 % más chicos que los safe. Un ejemplo destacable es el caso del protocolo GSM en el cual para uno de los requerimientos de transición no hubo reducción grande, solo unos pocos estados de diferencia.

# Aplicación a Procesos de Negocio

# 6

> *My model for business is The Beatles.*
>
> — **Steve Jobs**
> (Empresario)

La finalidad de este capítulo es poder resolver el problema de reconfigurar un proceso de negocio cuando los requerimientos cambian. Como la técnica desarrollada en el Capítulo 3 es una técnica general que funciona para cualquier proceso que pueda ser modelado como un controlador de eventos discretos, creímos que era posible resolver dicho problema utilizando nuestra técnica. Sin embargo, poder modelar un proceso de negocio como un controlador de eventos discretos será uno de los grandes desafíos de este capítulo.

Para la reconfiguración de procesos de negocio vamos a necesitar plantear dicho problema como un problema de actualización dinámica de controladores (ADC). Para poder reutilizar el algoritmo de la Sección 3.3.2 necesitamos traducir una especificación de un proceso de negocio a un problema de control LTS (ver Definición 2.3.2). El resultado que obtendríamos al resolver este último, es un controlador que define la estrategia del proceso de negocio (i.e., las ejecuciones posibles deben ser exactamente las mismas que son permitidas por la especificación del proceso de negocio). Una vez que podamos traducir la especificación vieja y nueva de los procesos de negocio, tendremos todos los valores de entrada necesarios para plantear el problema de ADC.

Para obtener mayor garantías de esta traducción, es necesario dar una prueba de correctitud y completitud de la misma. Esto significaría probar que hay una equivalencia entre las ejecuciones de un modelo del proceso de negocio y el controlador sintetizado por un problema de control LTS.

Luego de traducir ambos modelos de procesos de negocio, y ejecutar nuestra solución al problema de ADC, se obtiene un autómata que representa la estrategia necesaria para reconfigurar un proceso de negocio debido al cambio de requerimientos para todo estado en que esté cualquier instancia viva. Finalmente, vamos a producir un Grafo Dinámico de Condición y Respuesta (DCR) especial al que llamaremos DCR de Reconfiguración. Lo definiremos para una instancia viva particular extendiendo su historia (ejecutada bajo los viejos requerimientos) para que garanticen los requerimientos de transición y alcancen los nuevos requerimientos. Gran parte del plan descrito para poder resolver la reconfiguración de procesos de negocio fue publicada en [Nah+19].

El Capítulo estará estructurado de la siguiente manera: Inicialmente, presentamos un ejemplo motivador que detalla el proceso de negocio que se quiere reconfigurar. Luego, se muestra una sección donde se explica como realizar la traducción de procesos de negocio a problema de control LTS, probando correctitud y maximalidad de la traducción. El corriente Capítulo continua con una sección que explica como utilizar los resultados de la traducción como valores de entrada para un problema de ADC. Así mismo, también se explica como producir un DCR de Reconfiguración, el cual le da semántica de ejecución a una instancia viva del proceso viejo para que esta cumpla con los requerimientos de transición y alcance los nuevos requerimientos. Todo este cómputo debe ser probado, y por lo tanto, probamos la correctitud de la construcción del DCR de Reconfiguración. La última Sección de este Capítulo muestra un conjunto de casos de estudio en los cuales hemos utilizado la técnica.

## 6.1 Ejemplo Motivador: Hospital Oncológico

Tomemos como escenario a un proceso diseñado para aplicar drogas oncológicas en hospitales daneses de la vida real [HM11]. Este workflow tiene las actividades *prescribir medicina* (*prescribe medicine*) y *firmar* (*sign*), representando al doctor agregando y firmando una prescripción a la historia clínica de un paciente. Además, un enfermero, es capaz de *dar medicina* (*give medicine*) en respuesta de la

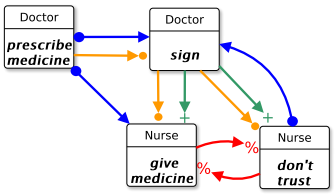


**Fig. 6.1:** DCR graph para el proceso del hospital.

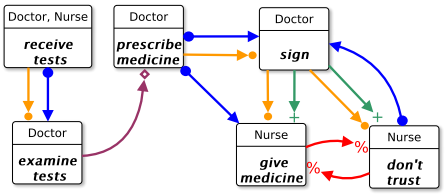


**Fig. 6.2:** DCR graph model para el nuevo procesos del hospital.

prescripción del doctor, o, en contraposición, el enfermero podría indicar que no confía en la prescripción o la firma, haciendo la actividad *no confiar* (*don't trust*). Los requerimientos del workflow incluye que I) el doctor debe prescribir medicina (*prescribe medicine*) a un paciente antes de firmarla (*sign*), II) el enfermero no puede dar medicina (*give medicine*) ni tampoco no confiar (*don't trust*) si el doctor no firmó (*sign*) la prescripción y III) el enfermero no puede dar la medicina y no confiar a la vez (*give medicine* y *don't trust*), solo puede hacer una de ellas.

La Figura 6.1 muestra estos requerimientos modelados usando Dynamic Condition Response (DCR) graphs tal cual está presentado originalmente en [HM11]. Un workflow que satisface estos requerimientos está dibujado en la Figura 6.3 donde las etiquetas *pm*, *s*, *gm* y *dt* se refieren a las actividades *prescribe medicine*, *sign*, *give medicine* y *don't trust*, respectivamente. El workflow es la semántica subyacente del modelo en la Figura 6.1 y puede construirse automáticamente usando síntesis de controladores como esta explicado en la Sección 6.2.

Supongamos ahora un escenario en donde mientras los pacientes están siendo atendidos, el proceso debe cambiar (tomado de [Muk12]). Por ejemplo, supongamos que una regulación interna empieza a implementarse que obliga a los doctores a no *prescribir medicina* (*prescribe medicine*) si nuevos exámenes han llegado (*receive tests*) pero no han sido examinados (*examine tests*). También, como es esperado, los exámenes deben llegar (*receive tests*) antes de ser examinados (*examine tests*). Este cambio involucra dos nuevas actividades y reglas adicionales como son modeladas en la Figura 6.2 que describe un workflow significativamente más complejo (Figura 6.4

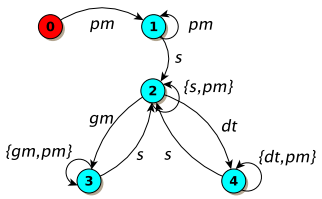


**Fig. 6.3:** Workflow del proceso del hospital.

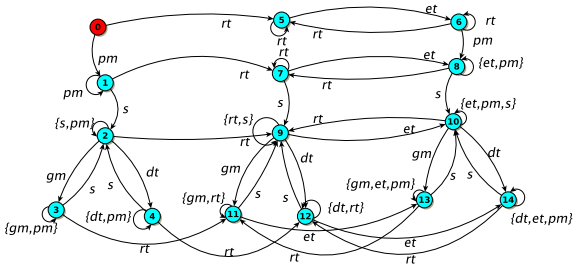


**Fig. 6.4:** Workflow del nuevo proceso del hospital.

pm: “prescribe medicine”, s: “sign”, gm: “give medicine”, dt: “don’t trust”, et: “examine test”, rt: “receive test”.

donde las etiquetas *rt* y *et* se refieren a *receive tests* y *examine tests*, respectivamente) que puede ser automáticamente sintetizado.

Una decisión crucial a tomar es cómo reconfigurar una instancia viva del workflow viejo, que está corriendo, al nuevo. Un enfoque ingenuo es pedir una reconfiguración inmediata [Ell+95] sin importar el estado actual de la instancia viva. Por lo tanto, si la instancia viva está en el estado $2$ de la Figura 6.3 debería evolucionar para llegar al estado $2$ de la Figura 6.4 (i.e., $2 \rightsquigarrow 2$). Sin embargo, esto pone al paciente en riesgo: el viejo workflow no registró la ocurrencia de *receive tests*. Nuevos exámenes pueden ser recibidos y nuevas prescripciones no deberían haberse hecho sin haber visto los exámenes primero. Sería mas seguro suponer, al momento de reconfigurar, que puede haber exámenes pendientes y forzar a revisarlos en vez de ignorarlos. Consecuentemente, es mas apropiado actualizar el viejo workflow a uno nuevo utilizando la siguiente correspondencia: $(0 \rightsquigarrow 5), (1 \rightsquigarrow 7), (2 \rightsquigarrow 9), (3 \rightsquigarrow 11), (4 \rightsquigarrow 12)$.

La tarea de definir una correspondencia entre los estados de los workflows, garantizando que el requerimiento de transición valga, puede ser muy difícil para workflows complejos. Una alternativa es permitir una descripción declarativa de los requerimientos de transición y computar la correspondencia automáticamente. Para el ejemplo, lo que se necesita es forzar *examine tests* cuando se reconfigura. Notar que es inconsistente con los requerimientos del viejo y el nuevo workflow a la vez. En el workflow viejo, no hay actividad *examine tests* y en los requerimientos del

nuevo workflow, *examine tests* debe suceder después de *receive tests*. Por lo tanto, lo que se necesita expresar es que hay un período durante la reconfiguración en donde ningún requerimiento vale y en donde *examine tests* (y nada más) debe ocurrir. En esta tesis, voy a mostrar como estos requerimientos de transición, específicos del dominio, pueden modelarse y como construir una estrategia automática para que las instancias vivan del viejo workflow empiecen a garantizar el nuevo workflow cumpliendo con todos los requerimientos de transición.

En síntesis, necesito resolver el problema de reconfiguración teniendo como valores de entrada las especificaciones de workflow actual y la del workflow nuevo, un requerimiento de transición representando las propiedades que la reconfiguración debe garantizar, y, la historia de lo que fue ejecutado bajo los requerimientos del workflow actual. La técnica propuesta debe construir una especificación de workflow que describa como continuar la historia dada, garantizando que el proceso de reconfiguración va a eventualmente terminar satisfaciendo los nuevos requerimientos. Para lograr esto, voy a presentar como modelar un requerimiento de transición especifico de dominio y como construir automáticamente una estrategia que toma a cualquier instancia viva corriendo en el workflow a un nuevo workflow garantizando todos los requerimientos de transición. Luego de tener esta estrategia, puedo extraer automáticamente una especificación de workflow que produce el proceso de reconfiguración para la historia dada (recordar Figura 1.1).

## 6.2 Semántica de DCR como Problema de Control

Mostraremos a continuación como extraer a partir de un DCR graph un conjunto de eventos controlables $L_C$, un conjunto de eventos no controlables $\widehat{L_C}$, un LTS $E$, una fluent definition $d$, y una formula LTL $G$ tal que el resultado de síntesis de controladores (Definición 2.3.2) devuelve un controlador que habilita y deshabilita actividades de forma tal que su ambiente, ejecutando solamente actividades habilitadas, satisface los requerimientos del proceso de negocio tal y como fueron

descritos en el DCR Graph. Por lo tanto, $L_C$ va a contener los eventos que habilitan y deshabilitan actividades, mientras que en $\widehat{L_C}$ tendremos los eventos que modelan la ejecución, por parte del equipo, de una actividad. Estos serán monitoreables pero no controlables. El LTS $E$ va a modelar los supuestos en los cuales el controlador va a tener que confiar para garantizar los requerimientos del workflow. El fluent definition $d$, definirá la valuación de un conjunto de fluents utilizados para definir $G$. Finalmente, la fórmula $G$ codifica los aspectos específicos del dominio del DCR graph. Específicamente, las flechas que establecen dependencias entre las actividades.

### 6.2.1 Eventos Controlables y Monitoreables

El conjunto de eventos que describe el problema de control está definido por las actividades que aparecen en el DCR graph (i.e., el conjunto $A$ del DCR graph). Introducimos dos eventos por cada actividad $a \in A$: $a\_disable$ y $a\_happened$. El primero es un evento controlable por el controlador. El segundo, es un evento que va a ser seleccionado por el ambiente (e.g., el/la enfermero/a y el/la doctor/a) para indicar que la actividad fue ejecutada. Vamos a decir que $a\_happened$ es monitoreable o no-controlable. Notar que no introducimos $a\_enabled$, ya que suponemos un evento $enableAll$ para reducir el número de eventos y estados del problema de control. El controlador va a habilitar todas las actividades ($enableAll$), luego, seleccionará cuales de esas deshabilitar. De esta forma, si el ambiente ejecuta una de las actividades habilitadas, la ejecución será consistente con los requerimientos del proceso de negocio.

Introducimos un evento extra, $menu$, para modelar la interacción basada en turnos donde el controlador ofrece a su ambiente un menú de actividades para realizar. Primero, el controlador seleccionará qué actividades deshabilitar, luego, indicará usando $menu$ que es el turno del ambiente para decidir que actividad ejecutar.

En conclusión, el conjunto de eventos controlables y no-controlables se define de la siguiente manera:

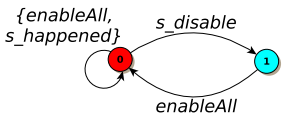


**Fig. 6.5:** LTS $Happens(s)$ que restringe la ocurrencia de $s\_happened$.

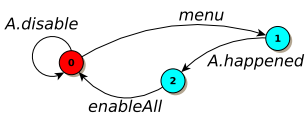


**Fig. 6.6:** LTS $Turns$ que define los turnos del controlador y el ambiente.

**Definición 6.2.1.** *Sea $DG = (A, R, M)$ un DCR graph. El conjunto de eventos controlables del DCR graph $DG$ es $L_C = \{a\_disable \,|\, a \in A\} \cup \{menu, enableAll\}$ y el conjunto de eventos monitoreables (no-controlables) del DCR Graph $DG$ es $\widehat{L_C} = \{a\_happened \,|\, a \in A\}$.*

### 6.2.2 Modelo del Ambiente

El LTS $E$ modela los dos supuestos en los cuales el controlador debe confiar para garantizar los requerimientos del proceso de negocio.

El primer supuesto es que las actividades solo pueden suceder cuando están habilitadas. Esto puede modelarse usando un modelo LTS por cada actividad que luego se compondrán en paralelo. En la Figura 6.5 muestro un LTS, $Happens(x)$, que modela el supuesto para la actividad *sign*. El estado $0$ modela que la actividad *sign* está habilitada (por lo tanto, posee una transición de salida $s\_happened$), mientras que, el estado $1$ modela que la actividad está deshabilitada (i.e., no posee una transición de salida $s\_happened$). Los eventos $enableAll$ y $s\_disable$ cambian el estado entre $0$ y $1$. Supongo que la actividad está inicialmente habilitada.

El segundo supuesto es que el ambiente jugará en turnos con el controlador. El controlador elige que actividades pueden ser ejecutadas sin violar los requerimientos del proceso de negocio, y luego, el ambiente elige cual de todas las actividades habilitadas ejecutar. Uso un sólo LTS, $Turns$, que está dibujado en la Figura 6.6, para modelar este supuesto. El estado inicial $(0)$ modela el turno del controlador donde cualquier actividad de $A$ puede ser deshabilitada. El evento $menu$ modela cuando el controlador cede su turno ofreciendo un menú de actividades a realizar. El estado $1$ es el turno del ambiente, en el cual, se selecciona solo una actividad de

$A$ a ser ejecutada, alcanzando el estado 2. Aquí, todas las actividades son habilitadas con el evento $enableAll$ para empezar de nuevo con el turno del controlador en el estado 0.

Los supuestos reflejan la operación del motor del proceso de negocio que será controlado. En el ejemplo del hospital de la Sección 6.1, el controlador primero decide que actividades deberían estar habilitadas ($enableAll$ y $a\_disabled$), y luego, presenta estas actividades al equipo de trabajo ($menu$). Se supone que los enfermeros y los doctores realizarán solamente aquellas actividades que son ofrecidas por el motor, y luego de su ejecución, el equipo reportará la actividad desarrollada al motor ($a\_happened$). A este punto, el controlador decidirá de nuevo que actividades se habilitarán para luego actualizar la oferta de actividades. Obviamente, el evento $menu$ debe solamente suceder cuando el controlador haya habilitado exactamente aquellas actividades que, en caso de ser ejecutadas, no deberían violar los requerimientos del proceso de negocio. El comportamiento del controlador es sintetizado automáticamente basado en la formalización de los objetivos que detallo a continuación

En conclusión, el LTS del ambiente $E$ se define de la siguiente manera:

**Definición 6.2.2.** *Sea $DG = (A, R, M)$ un DCR Graph con $A = \{a_0, \ldots, a_n\}$, $Turns$ el LTS de la Figura 6.6, $Happens(a_0), \ldots, Happens(a_n)$ los LTS de la Figura 6.5 para las actividades $a_0, \ldots, a_n$. El ambiente $E$ para el DCR graph $DG$ es $E = Turns \parallel Happens(a_1) \parallel \ldots \parallel Happens(a_n)$.*

### 6.2.3 Objetivos del Controlador

Los objetivos $G$ deben modelar las restricciones entre las actividades que se expresan en los DCR graphs con flechas entre actividades. Nuestra codificación sigue la misma lógica que la de [PA06] donde se utiliza fórmulas LTL para formalizar restricciones entre actividades de la misma forma que se restringe en los DCR graphs.

Primero definiremos el set de fluents necesarios para poder expresar los objetivos del controlador. Estos fluents, estarán comprendidos en la fluent definition $d$. Introduzco tres fluents por cada actividad $a \in A$ modelando si $a$ pertenece a los conjuntos $Ex$, $Re$, o $In$ siguiendo la Definición 2.4.3. Recordemos que suponemos que el marking inicial de un DCR graph es aquel en el que $In = A$ y $Re = Ex = \emptyset$.

- $d(a.Executed)$ modela si $a \in Ex$ y se define como $\langle\{a\_happened\}, \emptyset, \bot\}\rangle$. En otras palabras, utilizando la suposicion anterior, inicialmente ninguna actividad está en $Ex$ y una vez que una actividad está en $Ex$ no se quita más de allí (ver Definición 2.4.3(A)).
- $d(a.Required)$ modela si $a \in Re$ y se define como $\langle\{a'\_happened \mid a' \in (\bullet\!\!\rightarrow a)\}, a\_happened, \bot\rangle$. Esto es que toda actividad está inicialmente no requerida y la ejecución de la actividad $a$ hace que deje de estar requerida. La actividad $a$ pasa a estar requerida cuando cualquier actividad que espera la respuesta de $a$ es ejecutada (ver Definición 2.4.3(B)). En el ejemplo del hospital, el fluent $s.Required$ se define como $\langle\{pm\_happened, dt\_happened\},\ s\_happened,\ \bot\rangle$ porque la actividad *sign* es una respuesta de *don't trust* y *prescribe medicine* según la Figura 6.1. Notar que para aquellos casos donde $a\bullet\!\!\rightarrow a$, defino $d(a.Required) = \langle\{a'\_happened \mid a' \in (\bullet\!\!\rightarrow a)\}, \emptyset, \bot\rangle$ porque la ejecución de $a$ no hace falso al valor del fluent.
- $d(a.In)$ modela si $a \in In$ y está definido como $\langle\{a'\_happened \mid a' \in (\rightarrow_+ a)\}, \{a'.happened \mid a' \in (\rightarrow\!\%a)\}, \top\rangle$, que copia la Definición 2.4.3(C). Basado en las relaciones modeladas en la Figura 6.1, el fluent $gm.In$ está definido como $\langle\{s\_happened\}, \{dt\_happened\}, \top\rangle$.

Habiendo introducido un conjunto de fluents que registra la inclusión de cada actividad en los conjuntos $Ex$, $Re$ e $In$ copiando así los valores de los posibles marking de un DCR Graph, introducimos ahora, las fórmulas FLTL que preserva las reglas que definen cuando una actividad puede ser ejecutada (i.e., está habilitada) siguiendo la Definición 2.4.2. En otras palabras, las fórmulas relacionan la ocurrencia de $a\_happened$ con el valor de verdad de los fluents $a'.Executed$, $a'.Required$, y $a'.In$ para todo $a' \in A$.

- Para la regla $(a)$ de la Definición 2.4.2 introduzco para cada actividad $a \in A$ una fórmula $\alpha_a = (a\_happened \rightarrow a.In)$.
- Para la regla $(b)$ de la Definición 2.4.2 introduzco para cada actividad $a \in A$ una fórmula $\beta_a = (a\_happened \rightarrow \bigwedge_{a' \in (\rightarrow \bullet a)} (a'.In \rightarrow a'.Executed))$. Por ejemplo, para *sign*, siguiendo la Figura 6.1, tengo que $\beta_s = (s\_happened \rightarrow (pm.In \rightarrow pm.Executed))$.
- Para la regla $(c)$ de la Definición 2.4.2 introduzco para cada $a \in A$ una fórmula $\kappa_a = (a\_happened \rightarrow \bigwedge_{a' \in (\rightarrow \Diamond a)} (\neg a'.Required \vee \neg a'.In))$. Por ejemplo, $\kappa_{pm} = (pm\_happened \rightarrow (\neg et.Required \vee \neg et.In))$ para la Figura 6.2.

En conclusión, la definición de fluent $d$ y los objetivos $G$ para un DCR Graph $DG$ se definen de la siguiente manera:

**Definición 6.2.3.** *Sea $DG = (A, R, M)$ un DCR Graph.*

*La fluent definition $d$ para $DG$ define para cada $a \in A$ los siguiente fluents $d(a.Executed) = \langle \{a\_happened\}, \emptyset, \bot \} \rangle$, $d(a.Required) = \langle \{a'\_happened \mid a' \in (\bullet\rightarrow a)\}, \emptyset, \bot \rangle$ sii $a \bullet\rightarrow a$ ó $d(a.Required) = \langle \{a'\_happened \mid a' \in (\bullet\rightarrow a)\}, a\_happened, \bot \rangle$ en otro caso, y $d(a.In) = \langle \{a'\_happened \mid a' \in (\rightarrow_+ a)\}, \{a'\_happened \mid a' \in (\rightarrow \% a)\}, \top \rangle$.*

*Los objetivos $G$ para el DCR graph $DG$ son $\Box G = \Box \bigwedge_{a \in A} \alpha_a \wedge \beta_a \wedge \kappa_a$, donde, $\alpha_a, \beta_a$ y $\kappa_a$ son fórmulas LTL para cada actividad $a \in A$, tal que, $\alpha_a = (a\_happened \rightarrow a.In)$, $\beta_a = (a\_happened \rightarrow \bigwedge_{a' \in (\rightarrow \bullet a)} (a'.In \rightarrow a'.Executed))$ y $\kappa_a = (a\_happened \rightarrow \bigwedge_{a' \in (\rightarrow \Diamond a)} (\neg a'.Required \vee \neg a'.In))$.*

### 6.2.4 Síntesis de Workflows

Anteriormente hemos descrito como construir a partir de un modelo DCR graph $D$, el conjunto de eventos controlables $L_C$, el conjunto de eventos no controlables $\widehat{L_C}$, el LTS del ambiente $E$, la fluent definition $d$ y la fórmula FLTL $G$ que pueden ser usadas para definir un problema de control $\mathcal{E} = (E, G, d, L_C, \widehat{L_C})$. Una solución a este problema es un controlador LTS $C$ que decide cuando habilitar y deshabilitar actividades (que corresponden a eventos en $L_C$) tal que, al momento en que empieza a correr el ambiente, que juega en turnos, solo pueda ejecutar actividades habilitadas

(como se describe en $E$), satisfaciendo así todos los requerimientos del proceso de negocio (capturados por $G$). En otras palabras: $E \| C \models_d G$ (Definición 2.3.2).

Nótese que $E \| C \models_d G$ no es suficiente. Necesitamos que el controlador sea maximal en el sentido de que para toda ejecución de $menu$, el controlador habilite el conjunto mas grande de actividades que no violan $G$. Supongamos un workflow de un hospital en el cual siempre después de una firma del doctor (*sign*) solo se habilita la actividad de dar medicina (*give medicine*). La secuencia *sign*, *give medicine* no viola $G$, pero *sign* seguido de la actividad de no confiar en la firma del doctor (*don't trust*) debería ser posible también. Para garantizar maximalidad explotamos una característica particular del algoritmo de síntesis implementado en la herramienta MTSA [D'I+08]: MTSA construye componentes que intentan agresivamente satisfacer los requerimientos, construyendo así los caminos más cortos posibles hacia los requerimientos. Como el controlador es forzado a habilitar ($enableAll$), el algoritmo de síntesis tratará de deshabilitar lo menos posible para seguir garantizando $G$. Por lo tanto, el numero maximal de actividades estarán siempre habilitadas.

Los controladores para el DCR graph dibujados en las Figuras 6.1 y 6.2 tienen $188$ y $2291$ estados respectivamente y son muy grandes para ser dibujados en esta presentación. Sin embargo, muestro una versión abstracta de estos controladores (Figura 6.3 y 6.4) en el cual los eventos que habilitan ($enableAll$), los que deshabilitan y los $menu$ están ocultos. Esto representa una vista similar a lo que el/la Doctor/a y el/la enfermero/a deberían ver. Solo las actividades que están habilitadas y no las decisiones incrementales del controlador de habilitar y deshabilitar actividades. Nótese que los controladores abstractos se construyen automáticamente por la herramienta MTSA usando un operador de ocultamiento y minimización por bisimilaridad debil [Mil80].

### 6.2.5 Correctitud y Maximalidad de la Traducción

Esta sección explica porque la elección de $L_C$, $\widehat{L_C}$, $E$, $d$ y $G$ a partir de un DCR Graph $D$ es correcta y completa. Esto significa probar que si estos elementos son

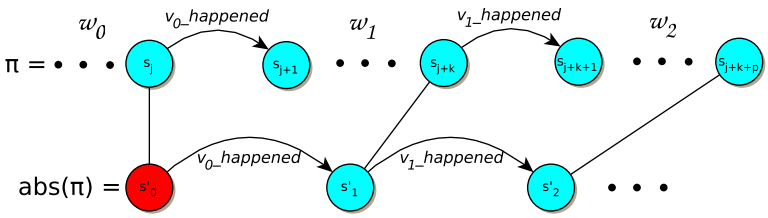


**Fig. 6.7:** Si $C$ es un controlador, entonces cada traza $\pi \in C$ tiene una traza abstracta $abs(\pi)$. Para todo $i \in \mathbb{N}$, $w_i$ es una subtraza que solo tiene eventos disable, $enableAll$ o $menu$.

utilizados como valores de entrada de un problema de control LTS, el resultado obtenido al resolver este problema mediante síntesis es exactamente un LTS cuyas trazas son las mismas trazas del DCR Graph $D$. La traducción es correcta si la versión abstracta de cualquier traza es una solución de la solución al problema de control traducido es una traza en el DCR graph. Por otro lado, la traducción es maximal si por cualquier traza del DCR graph hay una traza abstracta en una solución maximal del problema de control traducido. Por lo tanto, necesito definir qué es una traza abstracta de un LTS solución a un problema de control.

**Definición 6.2.4** (Traza Abstracta de un Controlador)**.** *Sean $\mathcal{E}_D = (E, \Box G, d, L_C, \widehat{L_C})$ un problema de control donde $E$, $G$, $d$, $L_C$ y $\widehat{L_C}$ fueron extraídos de $D$ como fue descrito anteriormente (Definiciones 6.2.1- 6.2.3), $C$ una solución a $\mathcal{E}_D$, y, $\pi$ una traza en $C$. La Traza Abstracta de $\pi$ en el controlador $C$ es la traza $abs(\pi)$ en la cual los eventos disable, $enableAll$ y $menu$ están ocultos (ver Figura 6.7)*

Luego, el teorema que necesitamos probar es el siguiente:

**Teorema 6.2.1.** *Sean $D = (A, R, M_0)$ un DCR graph y $\mathcal{E}_D = (E, \Box G, d, L_C, \widehat{L_C})$ un problema de control extraído de $D$ como describimos en las definiciones 6.2.1- 6.2.3.*

*CORRECTITUD. Si $C$ es una solución a $\mathcal{E}_D$ y $\pi$ es una traza en $C$, donde $abs(\pi) = v_0\_happened, v_1\_happened, \ldots$, entonces $v_0, v_1, \ldots \in traces(D)$.*

*MAXIMALIDAD. Si $C$ es una solución maximal a $\mathcal{E}_D$ y $v_0, v_1, \ldots \in traces(D)$, entonces, existe una traza $\pi \in C$ tal que $abs(\pi) = v_0\_happened, v_1\_happened, \ldots$.*

*CORRECTITUD.* Como $C$ es un LTS que es solución al problema de control $\mathcal{E}_D$, sabemos que $E \| C \models \Box G$ (ver Definición 2.3.2). Como $C$ es legal con respecto a $E$ (Definición 2.3.1), sus trazas deben ser un subconjunto de las trazas de $E$. Luego, también sucede que $C \models \Box G$. Por lo tanto, para todo $j \in \mathbb{N}_0$, $\pi, j \models G$. Por Lema 6.2.1 sabemos que $v_0.v_1, \cdots \in traces(D)$. □

*MAXIMALIDAD.* Asumiendo $D_0 \xrightarrow{v_0} D_1 \xrightarrow{v_1} \dots$ es una corrida en $D$, necesitamos probar que existe una traza $\pi \in C$ tal que $\pi = w_0.v_0\_happened.w_1.v_1\_happened, \dots$ donde las subtrazas $w_i$ no contienen eventos happened (ver Figura 6.7). Si esto es así, entonces sabremos que $abs(\pi) = v_0\_happened.v_1\_happened \dots$ basándonos en la Definición 6.2.4.

Para probar la existencia de $\pi$, necesitamos probar que (BASE) existe $w_0$ sin eventos happened tal que $w_0.v_0\_happened$ es prefijo de $\pi$ y (IND) dado $w_0 \dots w_{i-1}$ sin eventos happened tal que $w_0.v_0\_happened \dots w_{i-1}v_{i-1}\_happened$ es prefijo de $\pi$, entonces existe una subtraza $w_i$ sin eventos happened tal que $w_0.v_0\_happened \dots w_i.v_i\_happened$ es un prefijo de $\pi$.

Para probar (BASE), sea $\pi$ una traza en $C$. Como $C$ es legal con respecto a $E$ (Definición 2.3.2) tengo que $\pi$ debe ser una traza en $E$. Por construcción de $E$, $\pi$ debe ser una traza que empieza con una cantidad finita de eventos disable (entre 0 y $|A| - 1$), seguido de un evento $menu$. Llamo a este prefijo $w$. El estado de $C$ luego de ejecutar $w$ tiene transiciones habilitadas etiquetadas con $a\_happened$ para todo evento $a$ que no fue deshabilitado en $w$ (ver Figura 6.5).

Ahora debemos mostrar que $v_0\_happened$ está habilitada en este estado. Asumimos que no lo está y llegaremos a una contradicción. Como $C$ es maximal, si $v_0\_happened$ no está habilitado, entonces no existe una estrategia para el controlador en el estado $E$ alcanzado luego de $w.v_0\_happened$. Esto es equivalente a decir que el problema de control extraído desde $D_1$ no tiene solución. Sin embargo, existe una traza $v_1.v_2 \cdots \in traces(D_1)$ y por Lema 6.2.3 el problema de control extraído a partir de $D_1$ tiene una solución, alcanzando así una contradicción.

Ahora probaremos (IND): tengo que $D_0 \xrightarrow{v_0} \dots D_{i-1} \xrightarrow{v_{i-1}} D_i \xrightarrow{v_i}$ es un prefijo de una corrida infinita en $D$. Asumimos que $C$ tiene una traza $\pi$ con prefijo $\overline{\pi} = w_0.v_0\_happened.w_1.v_1\_happened \dots v_{i-1}\_happened$ donde $w_j$ no tiene eventos happened. Voy a mostrar $\overline{\pi}.w_i.v_i\_happened$ es un prefijo de una traza de $C$.

Sea $c$ el estado alcanzando en $C$ luego de $\overline{\pi}$. Como en el caso BASE, todas las trazas en $C$ que empiecen con $\overline{\pi}$ tienen una secuencia finita de eventos disable más $enableAll$ y $menu$ antes del próximo evento happened. Sea $w_i$ la secuencia mas larga tal que $\overline{\pi}.w_i$ es un prefijo de $\pi$. Sea $c'$ el estado alcanzado en $C$ luego de haber corrido $\overline{\pi}.w_i$. Asumo que $c'$ no tiene $v_i\_happened$ habilitado y llego a una contradicción al igual que como llegué para el caso BASE. □

**Lema 6.2.1.** *Sean $D = (A, R, M_0)$ un DCR graph, $\pi$ una traza con alfabeto definido como $L_C \cup \widehat{L_C}$ de la definición 6.2.1, y $k \in \mathbb{N}_0$. Si para todo $i \leq k$, $\pi, i \models G$ y $abs(\pi) = v_0\_happened, \dots, v_k\_happened$, entonces, $v_0, \dots, v_k \in traces(D)$, donde la definición de los fluents $d$ están inicializados de acuerdo a $M_0$ (como en Definición 6.2.3).*

*Demostración.* Voy a probar este Lema usando inducción sobre la longitud de una corrida en $D$. Quiero probar que (BASE) $D \xrightarrow{v_0}$, y el paso inductivo: (IND) si $D_0 \xrightarrow{v_0} D_1 \xrightarrow{v_1} \dots D_{i-1} \xrightarrow{v_{i-1}} D_i$ es una ejecución en $D$ con $i \leq k$, entonces, $D_i \xrightarrow{v_i}$. Voy a referirme como $M_i$ al marking del DCR graph $D_i$

Para probar (BASE) voy a mostrar que $v_0$ está habilitado en el marking $M_0$ porque se satisfacen las reglas (a)-(c) de la Definición 2.4.2.

La regla (a) vale sii $v_0 \in In_0$. Como el marking inicial fue definido para que tenga todas las actividades incluidas, tenemos en particular que $v_0 \in In_0$.

La regla (b) vale sii $(In_0 \cap (\rightarrow\bullet v_0) \subseteq Ex_0)$. Como $In_0 = A$ y $Ex_0 = \emptyset$ esta regla vale sii $(\rightarrow\bullet v_0) = \emptyset$. Supongamos que existe una actividad $b$ tal que $b \rightarrow\bullet v_0$. Como para todo $j \in \mathbb{N}_0$, sabemos que $\pi, j \models G$, entonces, para toda $a \in A$, la fórmula $\alpha_a \wedge \beta_a \wedge \kappa_a$ debe valer (ver Definición 6.2.3). En particular, $\beta_{v_0}$ debe valer luego de

la ejecución del primer evento happened (i.e. el evento $|w_0|+1$ de $\pi$, ver Figura 6.7): $\pi, |w_0|+1 \models \beta_{v_0}$. Recordemos que $\beta_{v_0} = v_0\_happened \rightarrow (b.In \rightarrow b.Executed)$ y que $\pi, |w_0|+1 \models v_0\_happened$. Por lo tanto, el consecuente de $\beta_{v_0}$ debe valer. Como $w_0$ no tiene ningún evento happened, entonces, en la posición $|w_0|+1$, los fluents $b.In$ y $b.Executed$ todavía tienen sus valores iniciales (verdadero y falso respectivamente) para todo $b \neq v_0$. Por lo tanto, si $b \neq v_0$ el consecuente no vale, alcanzando así a una contradicción. Si $b = v_0$, entonces, $v_0 \rightarrow\!\!\bullet v_0$ que no está permitido según la Definición 2.4.1.

La regla (c) es verdadera sii $Re_0 \cap In_0 \cap (\rightarrow_\Diamond v_0) = \emptyset$. Como $Re_0$ fue definido para estar vació en el marking inicial, entonces esta regla vale.

Por lo tanto, las reglas (a)-(c) de la Definición 2.4.2 estan garantizadas para la actividad $v_0$ en el marking $M_0$. Entonces, $M_0 \xrightarrow{v_0}$.

Ahora pasamos a probar (IND): $M_i \xrightarrow{v_i}$, suponiendo $D = D_0 \xrightarrow{v_0} D_1 \xrightarrow{v_1} \ldots D_{i-1} \xrightarrow{v_{i-1}} D_i$ con $i \leq k$. Luego, necesito probar que $v_i$ está habilitada en el marking $M_i$. Por Definición 2.4.2, necesito probar que:

(a) $v_i \in In_i$
(b) $In_i \cap (\rightarrow\!\!\bullet v_i) \subseteq Ex_i$
(c) $Re_i \cap In_i \cap (\rightarrow_\Diamond v_i) = \emptyset$

Como $abs(\pi) = v_0\_happened, \ldots, v_i\_happened, \ldots, v_k\_happened$, entonces debe haber una posición $j$ tal que $v_i\_happened$ es el evento $j$ en $\pi$ (i.e. $\pi, j \models v_i\_happened$). También, sabemos que $\pi, j \models G$. Por lo tanto, el consecuente de las fórmulas $\alpha_{v_i}, \beta_{v_i}$ y $\kappa_{v_i}$ deben valer en la posición $j+1$ (ver Definición 6.2.3).

Entonces, tenemos que:

I) $\pi, j+1 \models v_i.In$
II) $\pi, j+1 \models \bigwedge_{a \in (\rightarrow\!\!\bullet v_i)} (a.In \rightarrow a.Executed)$
III) $\pi, j+1 \models \bigwedge_{a \in (\rightarrow_\Diamond v_i)} (\neg a.Required \ \vee \ \neg a.In)$

Por I) y por Lema 6.2.2 sabemos que la actividad $v_i$ está incluida en el marking $M_i$ ($v_i \in In_i$). Luego, la Regla (a) de la Definición 2.4.2 está garantizada.

Por II) y por Lema 6.2.2 sabemos que para toda actividad $a \in (\rightarrow\bullet v_i)$ o bien $a \notin In_i$ ó $a \in Ex_i$. Por lo tanto, lo siguiente vale: $In_i \cap (\rightarrow\bullet v_i) \subseteq Ex_i$.

Por III) y por Lema 6.2.2 sabemos que para toda actividad $a \in (\rightarrow_{\Diamond}\ v_i)$ o bien $a \notin Re_i$ ó $a \notin In_i$. Por lo tanto, es verdad lo siguiente: $Re_i \cap In_i = \emptyset$.

Como las reglas (a), (b) y (c) de la Definición 2.4.2 valen para el marking $M_i$ y la actividad $v_i$, concluyo que $v_i$ está habilitada para su ejecución en el marking $M_i$: $D_i \xrightarrow{v_i}$. □

**Lema 6.2.2.** *Sean $\pi$ una traza en $C$ y $abs(\pi) =_0 \_happened, v_1\_happened, \ldots,$ $D$ un DCR graph tal que $D_0 \xrightarrow{v_0} D_1 \ldots D_{i-1} \xrightarrow{v_1-1} D_i$ es una corrida de $D$ que alcanza el marking $M_i$, y $j \in \mathbb{N}_0$ tal que $j = 0$ o $\pi, j \models v_i\_happened$. Para toda actividad $a \in A$:*

- $\pi, j \models a.Executed$ *sii* $a \in Ex_i$
- $\pi, j \models a.Required$ *sii* $a \in Re_i$
- $\pi, j \models a.In$ *sii* $a \in In_i$

*Demostración.* Esto se prueba trivialmente por el hecho de que la definición de los fluents $a.Executed$, $a.Required$ y $a.In$ (Definición 6.2.3) actualizan sus valores copiando la semántica de ejecución de los DCR graphs (Definición 2.4.3). □

**Lema 6.2.3.** *Sea $D$ un DCR graph, y $\mathcal{E}_D$ el problema de control extraído a partir de $D$. Si hay una traza infinita en $traces(D)$, entonces hay una solución a $\mathcal{E}_D$.*

*Demostración.* Dada una corrida infinita $D_0 \xrightarrow{v_0} D_1 \ldots$ construimos un LTS que es solución al problema de control $\mathcal{E}_D$ que tiene como única traza abstracta: $v_0\_happened.$ $v_1\_happened \ldots$. Construimos el LTS en dos pasos. Primero construimos un LTS que solo tenga eventos happened en su alfabeto y que además tiene una sola traza:

$v_0\_happened.v_1\_happened\ldots$. Esto se puede hacer con una cantidad finita de estados ya que cada estado modela un marking $M_0, M_1\ldots$ donde $M_i$ es el marking de $D_i$ (Note que existe una cantidad finita de markings posibles para un DCR graph). Teniendo este LTS le agregamos entre los eventos $v_i\_happened, v_{i+1}\_happened$ la subtraza $enableAll\ .w.menu$ donde $w$ tiene exactamente tantos eventos disable como eventos en $A$ exceptuando al evento $v_{i+1}$. El LTS resultante es una solución a $\mathcal{E}_D$. □

## 6.3 Controlando la Reconfiguración de Procesos de Negocio

En esta sección mostraremos como computar, usando Actualización Dinámica de Controladores (Definición 3.2.3), una estrategia de reconfiguración que guía la ejecución de las instancias del workflow que estan satisfaciendo los requerimientos actuales para que garanticen los nuevos requerimientos mientras se garantizan todos los requerimientos de transición.

Para este propósito, primero detallaremos como describir los requerimientos de transición específicos de dominio para una reconfiguración de procesos de negocio utilizando FLTL. Esto involucra utilizar los eventos previamente mencionados *startNewSpec* y *stopOldSpec*. Luego mostraremos cual es la pinta que tiene una solución a un problema de reconfiguración de procesos de negocio y como estas soluciones pueden construirse automáticamente resolviendo el problema de Actualización Dinámica de Controladores. Finalmente, presentamos un algoritmo que extrae a partir de esta solución, un DCR graph, que representa la reconfiguración dinámica de un proceso de negocio para una instancia viva particular.

### 6.3.1 La Especificación de un Requerimiento de Transición

Retomando el ejemplo del workflow del Hospital introducido en la Sección 6.1 donde un requerimiento de transición fuerza a que suceda la actividad de examinar un test (*examine tests*) durante la reconfiguración. Mas precisamente, justo antes

del momento en el que el nuevo proceso de negocio empiece a valer, la actividad de *examine tests* es requerida. El motivo por el cual requerir "justo antes" es porque ejecutar *examine tests* sin un previo *receive tests* no es permitido en el nuevo proceso de negocio.

Para formalizar este requerimiento de transición necesitamos referirnos al momento en el cual los viejos requerimientos del negocio dejan de valer (*stopOldSpec*) y el momento en el cual los nuevos requerimientos del negocio empiezan a valer (*startNewSpec*). Con estos dos eventos, el requerimiento de transición puede escribirse como se muestra a continuación. $T_h = \Box(\textit{stopOldSpec} \rightarrow ((\bigwedge_{a \in A \setminus \{et\}} \neg a.happened)$ W $(et.Executed \wedge \textit{startNewSpec})))$. Nótese que garantizar esta formula requiere habilitar y deshabilitar actividades de manera tal que los eventos no controlables $a.happened$ ocurran o no como se especifica en $T_h$ Un requerimiento de transición estándar, independiente de dominio, que fuerza a que en todo punto valga uno de los dos requerimientos (viejo o nuevo) puede especificarse de la siguiente manera: $T_\emptyset = \Box((\textit{OldSpecStopped} \wedge \neg \textit{NewSpecStarted}) \rightarrow \bigwedge_{a \in A} \neg a.happened)$. Nótese que esta configuración indica que no hay período de transición en donde no vale ninguno de los dos.

Formalmente, restringimos requerimientos de transición que sean propiedades FLTL de safety que solo prediquen sobre los eventos happened más *stopOldSpec* y *startNewSpec* y que no utilice el operador temporal X (i.e. son invariantes tartamudas [Lam94]).

### 6.3.2 Reconfiguración de Workflows

Volviendo a $T_h$, ¿cómo sería la solución al problema de reconfiguración? Supongamos que una instancia del workflow de la Figura 6.3 está en estado $2$. Una solución a la reconfiguración es instalando un workflow que fuerce la actividad *examine tests* y luego alcance al estado $10$ de la Figura 6.4. En otras palabras, necesito construir un workflow que administre la transición desde el viejo al nuevo workflow. Llamo a este workflow el *workflow de reconfiguración*.

Este workflow de reconfiguración supone que la vieja instancia que está en el estado $2$ es inadecuado para tomar otra actividad del viejo workflow (e.g., *give medicine*), para saltar así al estado $3$ (Figura 6.3). En cambio, la reconfiguración debe forzar a que sucede la actividad *examine tests* y luego mover al estado $13$ de la Figura 6.4 en vez de llegar al $10$. Por lo tanto, el objetivo es construir un workflow de reconfiguración que pueda administrar la transición desde cualquier estado del viejo workflow.

Conceptualmente, la solución construye un workflow de reconfiguración que consiste en tres fases como las explicadas en la Sección 3.3.2. La primera es estructuralmente equivalente al viejo workflow. La segunda fase es donde la reconfiguración puede empezar a diferenciarse del comportamiento del viejo workflow para empezar a garantizar los requerimientos de transición. La tercer fase es una en la cual los nuevos requerimientos de transición se satisfacen. Reutilizamos los eventos *hotSwap*, *stopOldSpec* y *startNewSpec* tal y como fueron utilizados para la solución del problema de Actualización Dinámica de Controladores (ver Figura 3.6).

En la Figura 6.8 se muestra un workflow de reconfiguración abstracto (los eventos de habilitación, deshabilitación y $menu$ son ocultados) que implementa la reconfiguración de los requerimientos de los procesos de negocio de la Figura 6.1 a los de la Figura 6.2 bajo el requerimiento de transición $T_h$. El rectángulo azul a la izquierda representa la primera fase de la reconfiguración de workflow. Nótese que la estructura de los estados y transición es exactamente igual al workflow que será reemplazado (Figura 6.3), por lo tanto, hacer un hotswap de este workflow es trivial. Nótese que todos los estados de la región en azul tienen transiciones de salida con etiqueta *hotSwap*. Cuando *hotSwap* se ejecuta, sin importar en que estado de la ejecución esté, va a haber un camino hasta la región amarilla en la derecha. La región amarilla representa el nuevo workflow tal y como se muestra en la Figura 6.4. La transición desde los requerimientos viejos a los nuevos, mientras se satisface los requerimientos de transición están representados por los estados fuera de los dos rectángulos. Cabe destacar que no hay ciclos durante esta fase de transición, lo que

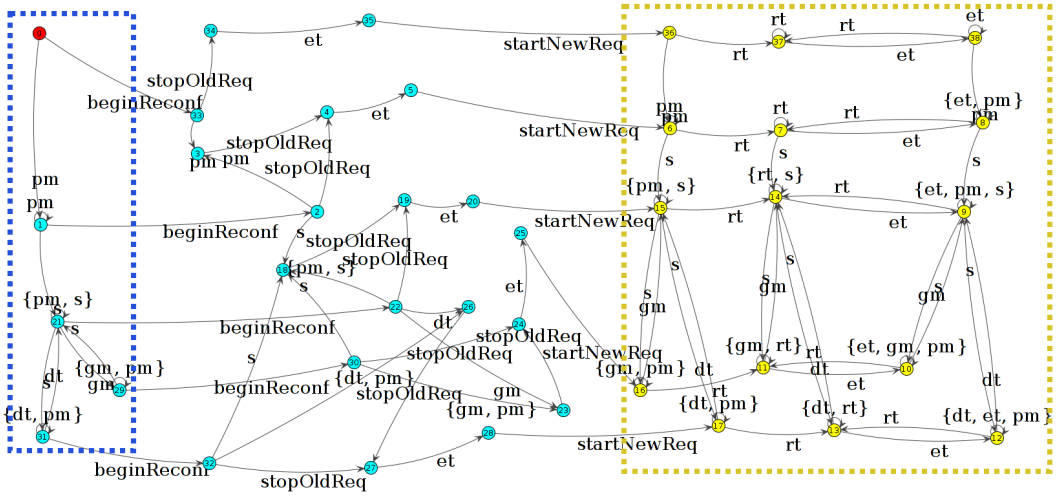


**Fig. 6.8:** Workflow de reconfiguración con requerimiento de transición $T_h$. pm: "prescribe medicine", s: "sign", gm: "give medicine", dt: "don't trust", et: "examine test", rt: "receive test".

garantiza que eventualmente los nuevos requerimientos del proceso de negocio van a valer.

### 6.3.3 Construcción Automática del Workflow de Reconfiguración

Concluyendo, la Figura 6.8 representa una solución al problema de reconfiguración de requerimientos de procesos de negocio desde aquellos de la Figura 6.1 a aquellos de la Figura 6.2 bajo requerimientos de transición $T_h$. Ahora explicaremos como se construye dicha solución a través de resolver un problema de Actualización Dinámica de Controladores (ADC) de la Definición 3.2.3. El problema de ADC (definido por $\mathcal{E}, C, \mathcal{E}', T, R$ y $\overline{d_u}$) requiere de dos problemas de control $\mathcal{E} = (E, G, d, L_C, \widehat{L_C})$ y $\mathcal{E}' = (E', G', d', L'_C, \widehat{L'_C})$ que representan en este caso los problemas de síntesis del viejo proceso de negocio y del nuevo como se describe en la Sección 6.2. La ADC también requiere del workflow viejo $C$, un requerimiento de transición $T$, un mapeo de estados $R$ desde los estados de $E$ a estado de $E'$ y una extensión de fluent definition $\overline{d_u}$ Ya he discutido $T$, $C$ viene dado ya que supongo que es el controlador del workflow viejo que se desea reemplazar y $\overline{d_u}$ se arma de manera trivial. Discutamos entonces $R$.

El propósito de la relación $R$ es poder explicar la relación entre los supuestos modelados en cada problema de control. El problema es que $E$ registra los supuestos para el controlador sintetizado $C$. Cuando se establece la reconfiguración no es posible saber en que estado de los supuestos de $E'$ se está. La relación $R$ debe ser provista por el usuario tratando de solucionar este problema. En esta configuración, la relación puede ser trivialmente definida como las únicas diferencias entre $E$ y $E'$ representando actividades que están presentes en un proceso de negocio pero no en el otro. Además, se sabe que cualquier nueva actividad puede no haber sido habilitada nunca por el controlador del viejo workflow. En consecuencia, $R$ podría definirse como la relación identidad de estado para todo LTS que está en $E$ y $E'$ y como una relación constante $0$ (i.e., el estado inicial) para LTLs representando nuevas actividades. Asumimos la unión disjunta de $D$ y $D'$. Aunque una actividad en $D$ y $D'$ pueden tener el mismo nombre (y representan la misma operación en el dominio del proceso de negocio), trato a cada actividad compartida entre $D$ y $D'$ como si fueran diferentes. La razón de hacer esto, es que cada actividad puede cambiar sus relaciones (flechas) con el resto de las actividades cuando se cambia de $D$ a $D'$. Por lo tanto, habilitar y deshabilitar una actividad puede cambiar en base al nuevo modelo $D'$. El mismo caso sucede cuando considero la definición de fluents en $d$ y en $d'$

Por lo tanto, dado dos DCR Graphs $D$ y $D'$ que describen los requerimientos del viejo y el nuevo proceso de negocio y un requerimiento de transición $T$, podemos automáticamente construir problemas de control $\mathcal{E} = (E, G, d, L_C, \widehat{L_C})$ y $\mathcal{E}' = (E', G', d, L'_C, \widehat{L'_C})$ como se describe en la Sección 6.2 y tomar $R$ para describir y solucionar un problema de ADC. Una abstracción a la solución del problema de ADC para la reconfiguración del hospital con $T_h$ se muestra en la Figura 6.8 y fue automáticamente construido por la herramienta MTSA.

Una nota metodológica importante es que no todo problema de ADC tiene una solución. Es posible tener dos problemas de control $\mathcal{E}$ y $\mathcal{E}'$ que son individualmente realizables pero para ciertos requerimientos de transición la reconfiguración sea

imposible. En términos de la reconfiguración de procesos de negocio esto significa que es posible empezar procesos de negocio con con uno de los dos conjuntos de requerimientos pero al proponer un requerimiento de transición muy complejo la reconfiguración correcta pasa a ser imposible. Un ejemplo de esto, para el ejemplo del Hospital, es requerir $T = \Box(\textit{startNewSpec} \rightarrow \neg pm.Executed)$. No hay una estrategia que pueda realizar una reconfiguración para garantizar los nuevos requerimientos *sin importar el estado actual*. Es decir, no hay estrategia de reconfiguración que pueda garantizar que los nuevos requerimientos de proceso de negocio solo empiezan a valer en aquellos estados donde la actividad *prescribe medicine* no sucedió.

### 6.3.4 Reconfiguración de DCR graphs

Habiendo discutido como construir un workflow de reconfiguración, debemos ahora mostrar como construir un DCR graph a partir de la estrategia de reconfiguración para una instancia particular del workflow, y lo llamaremos a este DCR graph como Reconfiguración de DCR. Definiré primero que es una Reconfiguración de DCR y luego presentaré un proceso automático que lo construye.

La reconfiguración de workflow discutida previamente codifica una estrategia de reconfiguración para cada estado en donde puede estar una instancia viva, que está siguiendo las viejas reglas de procesos de negocio. Para otorgar respuesta a un usuario que especificó las reglas del negocio en DCR graphs, es conveniente mostrarle como una instancia particular puede ser continuada para que adecuadamente reconfigure el proceso de negocio.

Por lo tanto, asumo que tengo una traza particular $\alpha$ que satisface que $\alpha \in traces(D)$ y corresponde a las actividades que atravesó una instancia del workflow. Dada esta traza, la ejecución de la Reconfiguración de DCR debe continuar $\alpha$ garantizando el criterio de correctitud de la Definición 3.2.2. Esto significa exigir que cualquier traza de la Reconfiguración de DCR debe continuar $\alpha$ con un comportamiento $c$ que mantiene las reglas establecidas en el DCR graph $D$ para luego continuar con un período de transición $t$, para luego finalizar un período $r$ cuyo comportamiento se

adaptan a las reglas de $D'$. Notar que $r$ necesita ser una traza de $D'$ ya que algunas actividades requeridas en $D'$ pueden haber ocurrido anteriormente. Restrinjo, por lo tanto, que $r$ sea una traza de $D'$ con el marking inicial actualizado para ser consistente con lo ocurrido anteriormente en $\alpha.c.t$.

**Definición 6.3.1.** *(Reconfiguración de DCR) Sean $D = (A, R, M_0)$ y $D' = (A', R', M'_0)$ dos DCR graphs para la vieja especificación y para la nueva, respectivamente, donde $M_0 = (Ex_0, Re_0, In_0)$ y $M'_0 = (Ex'_0, Re'_0, In'_0)$, $T$ una fórmula FLTL representando el requerimiento de transición, y, $\alpha$ una traza en $D$. Una Reconfiguración de DCR para la traza $\alpha$ es un DCR graph $DR^\alpha$ tal que $\forall \omega \in traces(DR^\alpha)$, existen las trazas finitas $c$ y $t$ y una traza infinita $r$ tal que $\omega = c.t.r$ con I) $\alpha.c \in traces(D)$, II) $\alpha.beginReconf.c.stopOldSpec.t.startNewSpec.r \models T$ y, III) $r$ puede ser ejecutado desde $(A', R', M')$ donde $M'$ es consistente con $D'$ y $\alpha.c.t$.*

Volviendo al ejemplo del hospital discutido en la Sección 6.1 y su workflow de reconfiguración dibujado en la Figura 6.8. Para el caso en donde la medicina fue prescrita pero nada mas fue ejecutado ($\alpha = pm$), una Reconfiguración de DCR está dibujada en la Figura 6.9. Nótese que este DCR graph fuerza *examine tests* antes de pasar a una corrida que satisface las nuevas reglas de negocio. Esto corresponde al período de transición dibujado en la Figura 6.8 como el camino entre los estados 1, 2, 4, 5 y 6. Nótese también que el nuevo DCR graph que está dibujado entre lineas punteadas tiene un marking distinto al DCR graph original (ver Figura 6.2): la actividad *prescribe medicine* está marcada como ejecutada. Esto es consistente con $\alpha$.

### 6.3.5 Construyendo una Reconfiguración de DCR

Construimos una Reconfiguración de DCR a partir del controlador de reconfiguración $C_u$ computado como describí anteriormente. Dada una historia $\alpha$, identifico una traza que representa el período de transición y produzco un DCR graph que fuerza esa traza. Luego agrego al DCR graph una copia del nuevo DCR graph con un

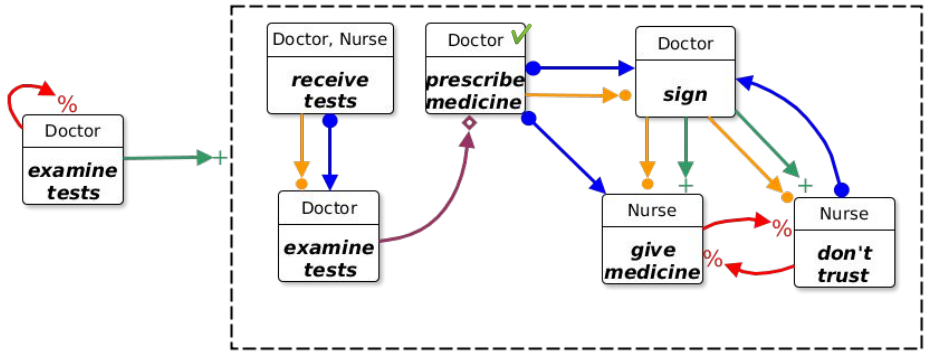


**Fig. 6.9:** Reconfiguración de DCR para requerimiento de transición $T_h$ y $\alpha = pm$.

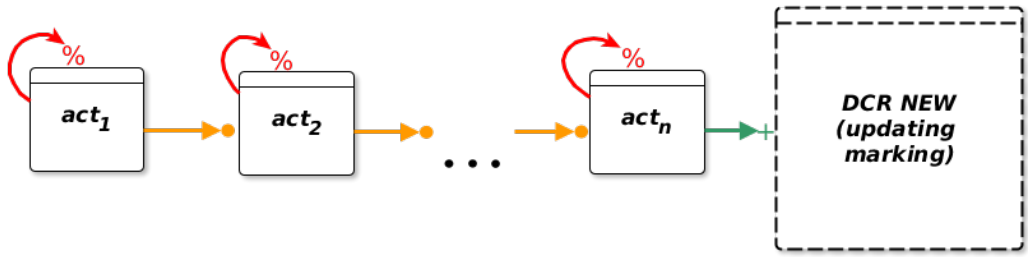


**Fig. 6.10:** Estructura de la Reconfiguración de DCR.

marking inicial que es consistente con $\alpha$. El DCR graph resultante sigue la estructura dibujada en la Figura 6.10.

En el Algoritmo 1 mostramos como producir una Reconfiguración de DCR usando el viejo y el nuevo DCR graph ($D$ y $D'$, respectivamente), una traza $\alpha \in traces(D)$, y una solución al problema de ADC ($C_u$) para $\mathcal{E}_D$, $\mathcal{E}_{D'}$ y un requerimiento de transición $T$.

Las Lineas 2-7 en Algoritmo 1 consume la traza $\alpha$ mientras cambia el marking de $D$ siguiendo la Definición 2.4.3 y tambien el estado actual de $C_u$. En la Linea 8, el estado actual de $C_u$ es aquel desde el cual la Reconfiguración de DCR debe ser construido.

El algoritmo continúa con un ciclo while que agrega a la Reconfiguración de DCR una actividad por vez siguiendo el camino en $C_u$ que empieza con *beginReconf*. Repite este ciclo hasta que finaliza el período de transición (esto es cuando se visitó los eventos

**ALGORITHM 2:** Pseudocodigo para construir la Reconfiguración de DCR.

```
1  DCRReconfiguration (α, C_u, D, D')
2      resultDCR ← DCR();
3      history ← α ;
4      foreach activity in α do
5          C_u.executeHappened(activity.toEvent());
6          D.execute(activity); // update marking as Definition 2.4.3
7      end
8      C_u.execute("beginReconf");
9      toConsume ← ["startNewReq", "stopOldReq"];
10     lastNode ← Null;
11     while ToConsume is not empty do
12         event ← C_u.getEnableAbsEvent();
13         if event in toConsume then
14             toConsume.remove(event);
15         else
16             D.execute(event.toActivity());
17             history.append(event.toActivity());
18             if lastNode is not Null then
19                 newNode ← resultDCR.addNode(event.toActivity());
20                 resultDCR.addCondition(lastNode, newNode);
21                 lastNode ← newNode;
22             else
23                 lastNode ← resultDCR.addNode(event.toActivity());
24             resultDCR.addExclusion(lastNode, lastNode);
25         C_u.execute(event);
26     end
27     D'.setMarkingConsistentTo(history);
28     if lastNode is Null then
29         return D';
30     dcrNewNode ← createNotIncludedNode(D');
31     resultDCR.addInclusion(lastNode, dcrNewNode);
32     return resultDCR;
```

$stopOldReq$ y $startNewReq$). Este ciclo utiliza los métodos $toEvent()$ y $toActivity()$ para convertir el evento $s\_happened$ hacia la actividad *sign* y viceversa.

En la Linea 12) obtenemos el primer evento alcanzable desde el estado actual de $C_u$ que no es un evento disable ni tampoco un $menu$ ni un $enableAll$. Si hubiese más de uno, se selecciona uno de ellos aleatoreamente. En otras palabras, obtenemos tanto un evento *stopOldSpec* o *startNewSpec* o un evento happened.

Por cada evento happened, agrego (Lineas 17-24) la actividad a la Reconfiguración de DCR. Al agregar estas actividades, creo una secuencia de actividades encadenadas por una relación de condición (i.e. $a_i \rightarrow\bullet a_{i+1}$ para todo $i \geq 0$). Luego, a cada una de estas actividades, le agrego una relación de exclusión hacia ellos mismos para prohibir que sean ejecutados mas de una vez (Linea 24).

Al alcanzar la Linea 27, el período de transición en $C_u$ ya terminó y, por lo tanto, agrego el nuevo DCR graph a la Reconfiguración de DCR. Sin embargo, primero debemos actualizar su marking basado en al historia de las actividades que fueron ejecutadas (ver Definición 2.4.6). El DCR $D'$ es agregado a la Reconfiguración de DCR usando $createNotIncludedNode$ que crea un nuevo nodo conteniendo $D'$ que está inicialmente no incluido, y por lo tanto, ninguna de las actividades en $D'$ pueden ser ejecutadas. También agrego una relación de Inclusión desde la última actividad de la secuencia de transición a este nuevo nodo, para garantizar que cuando el período de transición se acaba, $D'$ pasa a estar activo.

### 6.3.6 Correctitud del Algoritmo

A continuación presentamos y probamos la correctitud del Algoritmo 1. Por correcto queremos decir que el Algoritmo 1 produce, efectivamente, una Reconfiguración de DCR de la Definición 6.3.1.

**Teorema 6.3.1.** *Sea $D$ y $D'$ DCR graphs para la vieja especificación y para la nueva especificación, respectivamente, $T$ una fórmula FLTL representando un requerimiento de transición, $\alpha$ una traza en $D$, $C_u$ una solución al problema de ADC definido por*

*el requerimiento de transición $T$ y traduciendo $D$ y $D'$ a $\mathcal{E}_D$ y $\mathcal{E}_{D'}$, respectivamente segund definiciones 6.2.1- 6.2.3.*
*Si $DR^\alpha$ es la salida del Algoritmo 1 con inputs $(\alpha, abs(C_u), D, D')$, entonces $DR^\alpha$ es una Reconfiguración de DCR para la traza $\alpha$ de la Definición 6.3.1.*

*Demostración.* Sea $\omega$ una traza de $DR^\alpha$, $D = (A, R, M_0)$ y $D' = (A', R', M'_0)$. Quiero mostrar que existe trazas finitas $c$ y $t$ y una traza infinita $r$ tal que $\omega = c.t.r$ con I) $\alpha.c \in traces(D)$, II) $\alpha.beginReconf.c.$*stopOldSpec*$.t.$*startNewSpec*$.r \models T$, y, III) $r$ puede ser ejecutada desde $(A', R', M')$ donde $M'$ es consistente con $D'$ y $\alpha.c.t$.

Siguiendo las Lineas 2-27 del Algoritmo 1, es fácil ver que existe una traza $\pi \in C_u$ tal que $abs(\pi) = \alpha'.$*beginReconf*$.c'.$*stopOldSpec*$.\ t'.$*startNewSpec*$.r'$. Defino $\alpha$, $c$, y $t$ remplazando todo $a_i\_happened$ en $\alpha'$, $c'$, y $t'$ con $a_i$.

Siguiendo la Definición 3.2.3 sé que (a) $\pi \models G$ W *stopOldSpec*, (b) $\pi \models T$, (c) $\pi \models \Box($*startNewSpec* $\rightarrow G')$ y (d) $\pi \models \Box($*beginReconf* $\rightarrow (\Diamond$ *stopOldSpec* $\wedge \Diamond$ *startNewSpec*$))$.

Por (a) puedo afirmar que existe $k \in \mathbb{N}_0$ (posición en donde *stopOldSpec* sucede en $\pi$) tal que para toda posición $i \leq k$, $\pi, i \models G$. Por lo tanto, puedo utilizar el Lema 6.2.1 para decir que $\alpha'.$*beginReconf*$.c'$ es una traza con eventos happened que pueden ser ejecutados en $D$ sin *beginReconf*. Ergo, $\alpha.c \in traces(D)$.

Por (b) y por el hecho de que $T$ no refiere a eventos ocultados por $abs()$ y que tampoco utiliza el operador temporal X , puedo concluir que $abs(\pi) \models T$.

Finalmente, por (c) tengo que $\pi, j \models G'$ para todo $j > k$ donde $k$ es la posición de *startNewSpec* en $\pi$. Esto es equivalente a decir que $r', 0 \models G'$ con fluents inicializados para reflejar la valuación en la posición $k$ de $\pi$. Mas precisamente, si $\pi, j \models_d G'$ para todo $j > k$ entonces $r', 0 \models_{d'} G'$ donde la definición de fluents $d'$ es aquella en al que todos los fluents $f$, $\pi, k \models_d f$ si y solo si $f_{d'}.Init$.

Sea $M'_0 = (Ex'_0, Re'_0, In'_0)$. Defino $M' = (\delta_{Ex}(Ex'_0, \alpha.c.t), \delta_{Re}(Re'_0, \alpha.c.t), \delta_{In}(In'_0, \alpha.c.t))$. Por construcción $M'$ es consistente con $D'$ y $\alpha.c.t$. Lo que queda para probar es que $r$ es una corrida de $(A', R', M')$. Esto se prueba por el Lema 6.2.1 y por el hecho de que la definición de fluents $d'$ inicializa los fluents consistentemente con el marking $M'$. □

## 6.4 Casos de Estudio de Procesos de Negocio

El propósito de esta sección es mostrar aplicabilidad de nuestro enfoque usando, además del ejemplo motivacional de procesos de negocio, otros tres procesos de negocio tomados de BPM Academic Initiative [Bpma] y que a su vez fueron modelados en la herramienta de DCR Graph [Mar+16]. Elegimos estos para evitar parcialidad en producir nuestro propios DCR Graphs.

Cada caso de estudio necesitamos dos DCR Graphs, uno fuente y otro objetivo de la reconfiguración. Manualmente produjimos variantes para cada caso de estudio y usamos requerimientos de transición independientes de dominio tales como $T_\emptyset$ (ver Sección 6.3.1), además de los específicos de dominio. Todos los ejemplos fueron corridos usando una extensión de la herramienta MTSA [D'I+08] y pueden encontrarse en [Myt].

### 6.4.1 Hospital Oncológico

Este caso de estudio que ya discutido anteriormente en la Sección 6.1 es el único para el cual ambos modelos de DCR Graph de origen y destino ya existían. Ambos fueron tomados de [Muk12]. Modelamos varios requerimientos de transición alternativos y construí reconfiguraciones de procesos de negocio para cada uno de ellos.

Primero usé un requerimiento de transición trivial ($T_\top = \top$) para confirmar que existe una estrategia de reconfiguración pero esta misma permite efectuar un comportamiento indeseado. De hecho, el proceso de reconfiguración permite efectuar

el siguiente tipo de traza: *beginReconf*, *stopOldSpec*, *give medicine*, *startNewSpec* .... Esta traza es una en la cual una instancia viva que no ha efectuado ninguna actividad empieza a ser reconfigurada, los viejos requerimientos del proceso de negocio son descartados y, antes de empezar a garantizar los nuevos, el paciente puede recibir medicina (sin la prescripción firmada por el doctor!). Este problema surge porque $T_{\top}$ permite cualquier actividad durante la reconfiguración.

Decidimos utilizar otro requerimiento de transición estándar para este ejemplo. Utilizando $T_{\emptyset}$ obtengo una estrategia de reconfiguración que copia los dos esquemas propuestos en el artículo [Ell+95]. Es decir, una reconfiguración inmediata cuando es posible o, caso contrario, demorar la reconfiguración hasta el punto donde se pueda hacer inmediatamente.

Además, consideré dos requerimientos de transición específico de dominio probando que nuestra técnica puede computar otras estrategias de reconfiguración mas allá de [Ell+95]. Primero utilicé $T_h$ tal y como fue discutido en la Sección 6.3.1 y uno más que demora la reconfiguración cuando se da el caso en el que el/la enfermero/a ha indicado que no confía en la prescripción del/a médico/a: $T'_h = T_{\emptyset} \wedge \Box((dt.Executed \wedge \neg gm.Executed) \rightarrow \neg$*stopOldSpec* $)$. Como es esperado, el comportamiento de la reconfiguración resultante es muy parecido a la generada por $T_{\emptyset}$ con la excepción de que *stopOldSpec* es demorado para aquellas instancias que estén entre la ocurrencia de las actividades *don't trust* y *give medicine*. Por lo tanto, el proceso *forzará* al doctor a volver a firmar la prescripción o crear una prescripción nueva antes de poder terminar la reconfiguración. Este escenario, tal y como se plantea en [Ell+95], no puede ser considerado ni un cambio inmediato, ni un cambio demorado ya que se decide posponer una reconfiguración en un caso particular que el usuario definió en el requerimiento de transición. La demora no sucede por la naturaleza de la imposibilidad de poder hacerlo inmediato.

### 6.4.2 Proceso de evaluación de Doctores

Un proceso de evaluación de doctores de un hospital involucra a un gerente preguntándole a un experto que evalúe a cada doctor. Usamos el DCR Graph original como el modelo al cual se quiere llegar luego de la reconfiguración y removí una actividad para producir el DCR Graph fuente. Inicialmente el proceso no paga a los expertos por hacer la evaluación de los doctores y luego es reconfigurado a un proceso que remunera a los expertos por el trabajo realizado. Lo lógico de esta reconfiguración es evitar realizar un pago a aquellos expertos que aceptaron hacer el trabajo gratis (antes de la reconfiguración).

Nuevamente se puede argumentar que la utilización de $T_{\top}$ no es apropiado. El motivo, nuevamente, es que durante la reconfiguración hay un período en el cual cualquier cosa puede suceder (e.g., pagarle a un experto incluso sin que haya terminado la revisión).

Usamos, entonces, el requerimiento de transición $T_{\emptyset}$ obteniendo una estrategia de reconfiguración que puede ser efectuada inmediatamente en cualquier punto de la ejecución del primer proceso. Esto se debe a que la actividad de pagarle a los expertos se agrega simplemente al final del proceso fuente. Sin embargo, como destacamos anteriormente, una reconfiguración inmediata puede resultar en pagarle a expertos que han aceptado hacer la revisión gratis en el viejo proceso. Para evitar este escenario debemos formular un requerimiento de transición específico para este dominio. Este requerimiento de transición ($T_D$) va a predicar que si la reconfiguración es requerida luego de haber recibido la evaluación del experto, el experto no recibirá ninguna paga: $T_D = T_{\emptyset} \wedge (\Box(\mathit{startNewSpec} \wedge recExp.Executed) \rightarrow \Box\neg pay.Executed)$ donde $recExp$ es la actividad que representa la recepción de la evaluación del experto.

### 6.4.3 Proceso de Seguro

El proceso de negocio de una compañía de seguros incluye dos roles: agentes y empleados. Originalmente, el empleado, luego de recibir aun pedido de un nuevo cliente (*new customer claim*), debe llamar a un agente (*call the agent*) para que revise el pedido y debe crear un nuevo caso (*create a new customer case*). Para que el nuevo requerimiento empiece a valer, crear el nuevo caso (*create a new case*) debe suceder antes de el llamado al agente (*call the agent*). Este escenario, que corresponde a la clásica reconfiguración de paralelo a secuencial [VAS00], fue resuelto utilizando dos requerimientos de transición.

Usamos $T_\emptyset$ para computar un workflow de reconfiguración que demora la reconfiguración al punto en el que el llamado al agente ha sido ejecutado (*call the agent*) pero el caso no fue creado (*create a new case*). Para todos los otros escenarios, el workflow de reconfiguración hace un cambio inmediato.

Una alternativa diferente es modificar el modelo DCR Graph destino agregándole una actvidad *kill* que posee una relación de exclusión con todas las otras actividades del DCR Graph. Esta actividad modelará la eliminación de una instancia. Luego escribí un requerimiento de transición que fuerza a hacer *kill* cuando se llamó al agente (*call the agent*) antes de la creación del caso (*create a new case*). Este requerimiento se escribe de la siguiente manera: $T_I = T_\emptyset \wedge \Box((call.Executed \wedge \neg create.Executed \wedge \textit{startNewSpec}) \rightarrow (\neg ED \text{ W } kill.Executed))$, donde $ED$ es la disyunción de los eventos de deshabilitación de todas las actividades exceptuando la del *kill* más la actividad $enableAll$.

### 6.4.4 Proceso de Reparación de Computadoras

Un servicio de reparación de computadores empieza cuando un cliente trae una computadora defectuosa. Si el proveedor de servicios y el cliente acuerdan un presupuesto, entonces las actividades de reparar el hardware y reparar el software deben llevarse a cabo. Agregué un nuevo rol, que es de un supervisor, que debe

aprobar ese presupuesto antes de ser enviado al cliente. Usamos tres actividades para esto: enviar presupuesto al supervisor (*send to supervisor*), que el supervisor lo apruebe (*approve*) y que el supervisor lo rechace (*reject*).

Inicialmente, resolvimos este problema de reconfiguración con el requerimiento de transición estándar $T_{\emptyset}$. Como es esperado, para aquellas ejecuciones en las cuales la reconfiguración es requerida después de que el presupuesto sea enviado al cliente, la reconfiguración es demorada para no contradecir el requerimiento de aprobación del supervisor.

Modelé una alternativa a la reconfiguración anterior en la cual la reconfiguración fuerza a preguntar al supervisor por su aval en todas las instancias en las cuales el cliente ha recibido el presupuesto pero la reparación no ha empezado. Si el supervisor rechaza el presupuesto, entonces el cliente debe ser contactado para ofrecerle disculpas por lo ocurrido. La siguiente fórmula (donde $sup$ es la actividad *send to supervisor*) captura este requerimiento de reconfiguración: $T_C = \Box((\textit{stopOldSpec} \wedge \neg RepairStart) \rightarrow (\neg Happens \ \mathsf{W} \ (sup.Executed \wedge \textit{startNewSpec})))$ donde $Happens$ es la disyunción de los eventos de deshabilitación de todas las actividades exceptuando $yes$, $no$ y $sup$, más, el evento $enableAll$.

### 6.4.5 Procesos de Negocio: Resumen de las Experiencias

En total, $10$ reconfiguraciones fueron definidas y resueltas, correspondiente a diferentes elecciones de requerimientos de transición para cada caso de estudio. En la Tabla 6.1 reporto los ejemplos, el numero de distintas actividades y restricciones que están involucradas, el tamaño de los workflow de reconfiguración resultantes y el de la versión minimizada (estos ocultan las habilitaciones, deshabilitaciones y los eventos $menu$).

En la mayoría de los casos el requerimiento de transición $T_{\emptyset}$ reproduce la estrategia de reconfiguración como es explicada en [Ell+95] pero de manera automática. El algoritmo de síntesis pefectamente identifica cuando un cambio inmediato o

| Caso de Estudio | # Actividades | # Relaciones | Requerimiento de Transición | WF de Reconf. (# Estados) | WF de Reconf. Minimizado (# Estados) |
|---|---|---|---|---|---|
| **Hospital Oncológico** | 6 | 13 | $T_{\top}$ | 18667 | 54 |
| | | | $T_{\emptyset}$ | 9817 | 34 |
| | | | $T_h$ | 11155 | 39 |
| | | | $T'_h$ | 15094 | 54 |
| **Proceso de Evaluación de Doctores** | 10 | 25 | $T_{\emptyset}$ | 22448 | 39 |
| | | | $T_D$ | 27512 | 42 |
| **Proceso de Seguro** | 11 | 25 | $T_{\emptyset}$ | 15484 | 51 |
| | | | $T_I$ | 14233 | 48 |
| **Proceso de Reparación de Computadoras** | 18 | 26 | $T_{\emptyset}$ | 43307 | 59 |
| | | | $T_C$ | 52652 | 63 |

**Tab. 6.1:** Resumen de los casos de estudio ejecutados.

demorada debe hacerse. Para el hopsital oncológico y el proceso de evaluación de doctores usé el requerimiento de transición $T_{\top} = \top$ para mostrar que la no definición del requerimiento de transición causa comportamiento no deseado durante el proceso de reconfiguración. Como he mencionado en la Sección 6.2 utilicé cada modelo DCR Graph de la herramienta tal y como fue diseñado anteriormente.

# Discusión y Trabajo Relacionado

# 7

> *It is only discussed with whom you agree.*
>
> — **Guillermo Moreno**
> (Político y Economista)

En este capítulo es donde presento una discusión del trabajo relacionado con respecto a los tópicos estudiados en esta tesis poniendo el trabajo elaborado en la tesis en contexto.

## 7.1 Controladores de Eventos Discretos

Actualización dinámica de software ha sido estudiado extensivamente y hay una gran cantidad de diferentes problemas que deben ser estudiados dependiendo el dominio de la aplicación, la tecnología utilizada y el objetivo de la actualización (ver [Sei+13] para reconocerlos). Enfoques de actualizaciones dinámicas típicamente suponen que no hay un cambio de especificación provista [Nea+06; Che+07] y, por lo tanto, el comportamiento debe preservarse (solamente arreglar bugs), como por ejemplo [HC13], o, que la especificación es genérica y no es provista por el usuario, como en el caso de [She+05; Che+11; Ors+02; AR09; Gup+96; KM90]. Ejemplos de esto último, además de asegurar que las actualizaciones eviten crashes, poder asegurar safety (e.g., [Sub+09]) y aislamiento de datos entre versiones [Sto+07]. “Quiescencia” [KM90] y otras nociones relacionadas (e.g., [Van+07; AR09; Gup+96]) no resuelve originalmente la representación explícita de propiedades a ser preservadas durante la actualización, pero han sido usadas en conjunto con técnicas que aseguran consistencia de semántica (e.g., [Ban+10]).

La necesidad de propiedades de actualización especificadas por el usuario ha sido reconocida por [ZC05] donde se discute especificaciones de propiedades para ac-

tualizaciones. Luego, [Hay+12; Ram+10; ZC06], hacen construcciones manuales o semi-automáticas de actualizaciones para luego ser verificadas, mientras que [BG10; Ghe+12; PLM+13; An+15] proponen enfoques completamente automáticos basados en síntesis. Analizaré cada conjunto de trabajo relacionado separadamente.

### 7.1.1 Propiedades de Transición Especificadas por el Usuario

En [ZC05], se presentó una caracterización de patrones de diferentes propiedades de actualización. Los autores argumentan que diferentes patrones pueden aplicarse a diferentes dominios y diferentes escenarios de actualización. Por ejemplo, en un escenario de emergencia, se puede necesitar una actualización para que suceda a partir de cualquier momento de la ejecución actual y que ocurra tan pronto como sea posible. Este patrón corresponde a una propiedad de actualización "one-point". En una actualización planeada de, por ejemplo, una cadena de montaje automatizada, se puede requerir que las obligaciones de la especificación actual se completen antes de cambiar a la nueva especificación. Este escenario es referido en [ZC05] como "guided adaptation" ya que el sistema, satisfaciendo la especificación actual, debe ser guiado para completar las obligaciones pendientes sin adquirir nuevas. El artículo, sin embargo, no discute síntesis, es decir, la construcción de estrategias para conseguir estas actualizaciones.

En el trabajo presentado en esta tesis, no solo produzco una estrategia automática para varias propiedades de actualizaciones que identifican diferentes patrones de [ZC05] sino que también pude generar adicionalmente otros escenarios que no fueron considerados: Un período en el cual no vale la especificación actual ni la especificación nueva, sino que vale una propiedad temporaria alternativa.

Otros (e.g., [Nea+06; Gup94; Hay+12; MR07]) han considerado la necesidad de un período de transición entre especificaciones en donde ninguna especificación puede valer. Sin embargo, en esta tesis se provee un marco de trabajo para especificar formalmente los requerimientos para el período de transición y un algoritmo de

síntesis que garantiza su preservación. Por ejemplo, Neamtiu et. al. en [Nea+06] consideran un patrón de actualizaciones para programas C en el cual primero los usuarios envían una señal al programa que está corriendo; luego de eso, cuando el sistema que está corriendo alcanza un estado seguro el código de inicialización de la actualización se ejecuta.; y finalmente, este código "pega" el parche en el programa que está corriendo. Los requerimientos de transición de esta técnica son implícitos y el código de inicialización debe ser escrito manualmente. Makris y Ryu [MR07] también tienen una actualización con una estructura basada en fases, similar a la de Nematiu et al. y a la explicada en esta tesis. Sin embargo, como expliqué antes, la automatización de estas fases y la garantía de correctitud en la construcción es la diferencia clave.

Zhang y Cheng [ZC06] estudiaron el problema de construir estrategias para controlar las actualizaciones. Su enfoque es semi-automático ya que necesitan de una construcción manual de "adaptation models" que luego pueden ser verificados contra los requerimientos y usados para luego construir programas. Ramirez et al. [Ram+10] construyeron sobre este enfoque semi-automático una herramienta capaz de seleccionar y aplicar el mejor camino seguro de adaptación que balancea requerimientos no-funcionales, basandose en valores de costos. A modo de trabajo futuro, el trabajo presentado en esta tesis, podría ser extendido para que los requerimientos no-funcionales sean contemplados mientras se computa la estrategia de actualización utilizando, tal vez, técnicas de síntesis de controladores cuantitativos como en [Blo+09] y en [Syk+10].

### 7.1.2 Computación Automática de Estrategias de Actualización

Como en [BG10; Ghe+12; PLM+13; An+15] el enfoque de esta tesis es producir automáticamente un controlador actualizador en un sistema reactivo. En contraste con [Ghe+12; PLM+13], que han fijado la noción de correctitud, soporto los criterios especificados por el usuario para los requerimiento de transición. Además,

propongo una técnica para actualización dinámica que *garantiza* que el sistema va a alcanzar un estado seguro, incluso cuando el ambiente no es cooperativo, mientras que en [Ghe+12; PLM+13; NH09], se *supone* que los estados seguros serán alcanzados.

En [An+15], además de estos supuestos de liveness, también tienen fuertes restricciones en el tipo de patrones de actualizaciones que pueden suceder. Primero, requieren que los supuestos del nuevo ambiente a operar sea estrictamente más fuerte que el que estaba ejecutando como viejo ambiente. Segundo, los autores no permiten las diferentes semánticas de transiciones presentadas en [ZC05], sino que solo soportan transiciones "one-point". La noción de garantizar liveness (la actualización sucederá eventualmente) es una característica clave que distingue el enfoque presentado en esta tesis con respecto a los otros trabajos que computan automáticamente una estrategia de actualización. Notar que en [ZC05], esta característica de liveness es reconocida como un aspecto relevante de cualquier actualización, pero hasta donde pude explorar en la literatura, la técnica presentada aquí es la primera que introduce este concepto en la solución.

Tal vez, el trabajo más parecido al presentado en esta tesis es [Ghe+12] y [PLM+13]; es por eso que ya he hecho una comparación con ellos en el Capítulo 5 a través de casos de estudio. Tanto [Ghe+12], como [PLM+13], adoptan un criterio de correctitud general y muy natural que alivia el trabajo del ingeniero de especificar los requerimientos de transición (en contraste con el enfoque de la tesis), pero con el costo de limitar los tipos de actualizaciones que pueden soportarse. En [Ghe+12], si el sistema no puede volver a su estado inicial y no está ejecutando un comportamiento compatible con la nueva especificación desde el último estado inicial, entonces, no es posible realizar una actualización. El criterio de correctitud de actualizaciones en [Ghe+12] puede expresarse como un requerimiento de transición en el enfoque de esta tesis, pero adicionalmente, con mi técnica puedo garantizar progreso en busca de completar la actualización. Los autores en [PLM+13] debilitan el criterio de correctitud de una actualización con respecto a [Ghe+12], introduciendo la posi-

bilidad de actualizar sistemas donde el estado inicial no es re-visitado. Sin embargo, no hay garantías de que valga el criterio de correctitud original (equivalente a un actualización offline). La falta de garantías requiere que un ingeniero valide el controlador resultante. En el trabajo realizado en esta tesis, el ingeniero es involucrado a priori, definiendo la especificación del criterio de correctitud de la actualización ($T$) que luego es garantizada por construcción (e.g., ver el caso de estudio de la Planta de Energía de la Sección 5.2).

### 7.1.3 Sistemas de Eventos Discretos

El problema de actualización de controladores dinámicamente también se lo conoce en la literatura como sistemas de eventos discretos (Discrete Event Systems DES). Actualizar un DES, para esta comunidad significa resolver un problema de *reconfiguración* (e.g., [GR96; NS15; Sam+08]). Esta comunidad reconoce la necesidad de requerimientos explícitos que describan la transición de una configuración a la siguiente. Sin embargo, muchos trabajos en esta área suponen que las actualizaciones a ejecutarse son conocidas en tiempo de diseño, y por lo tanto, un sistema de eventos discretos es construido de forma tal que todos las posibles nuevas configuraciones son precomputadas. Por ejemplo, en [NS15], el DES actual tiene eventos especiales por cada nueva configuración disponible, de manera tal, que cuando estos eventos sean ejecutados, la actualización empieza. Tanto [GR96; NS15] precomputa reconfiguraciones que garantizan propiedades de safety que pueden ser expresadas en nuestro esquema de trabajo como parte de $T$ (ver Definición 3.2.1).

Por otro lado, en esta tesis no asumo que la nueva especificación la cual debe ser alcanzada luego de una actualización sea conocida en tiempo de diseño. Una de las dificultades de descartar este supuesto es que se necesita construir una estrategia que debe ser viable para todo estado actual del sistema que está ejecutando, y un mecanismo apropiado de cambio en caliente debe suceder. Todas estas características son lo novedoso del trabajo presentado en esta tesis.

Una notable excepción a suponer conocimiento en tiempo de diseño de actualización en la comunidad de sistemas de eventos discretos es [Sam+08] que está desarrollado bajo el formalismo de Petri nets. Sin embargo, los autores suponen que mientras la actualización se está computando el sistema actual está congelado. Esto es claramente irreal para muchos escenarios y es una restricción que nosotros superamos intercambiando en caliente por un controlador que en su fase inicial es capaz de emular al controlador actual y tiene una estrategia para actualizar desde cualquiera de esos estados (ver Figura 3.4).

### 7.1.4 Reconfiguración de Arquitecturas de Software

La actualización dinámica de controladores esta relacionada con la reconfiguración dinámica de arquitecturas de software. En la actualización de controladores, un componente (i.e., el controlador) es remplazado y el foco está puesto en el comportamiento que el sistema tiene como resultado de la coordinación que el controlador provee. Uno de los aspectos que puede ser coordinado es la reconfiguración arquitectural. En el trabajo de esta tesis, abstraje este concepto con el comando *reconfigure*. Sin embargo, el procedimiento de reconfiguración arquitectónico podría ser un proceso complejo que requiere de su propio planeamiento y actuación. La reconfiguración dinámica de arquitecturas de software es un campo activo de estudio (e.g., [Ars+07; Taj+10; Syk+08; Bar+16; Bra+15]) que complementa el campo de actualización de controladores.

### 7.1.5 Síntesis

Síntesis, la construcción automática de estrategias operacionales que satisfacen una especificación dada, ha sido estudia extensamente para garantizar código que es correcto por construcción (e.g., [Gre+13]). La completa automatización natural de la síntesis nos lleva naturalmente a su potente aplicación, no solo en tiempo de diseño, sino también en tiempo de ejecución para evolucionar sistemas de software. Esta evolución no está limitada exclusivamente a los sistemas adaptativos. Por ejemplo,

en [Pel+08] el problema de evolucionar conjuntos de componentes es solucionado a través de sintetizar "glue code" (i.e. controladores). Aunque la síntesis se produce sin parar el sistema, el nuevo controlador solo puede empezar a ejecutar una vez que el sistema esté en un estado de quietud.

La síntesis requiere algún tipo de especificación desde la cual, utilizando diferentes técnicas, se produce una solución. El resultado de una síntesis es correcto solo en la medida que la especificación sea válida. Por lo tanto, las técnicas de síntesis son, en principio, poco resistentes a errores en las especificaciones o ambientes que evolucionan y divergen de la especificación. El trabajo de esta tesis es también susceptible a especificaciones inválidas. En el dominio de sistemas adaptables, los enfoques que pueden detectar y corregir estas situaciones han sido estudiados (e.g., [D'I+14; Vro+11]), incluyendo como aprender nuevas especificaciones en tiempo de ejecución (e.g., [Syk+13]). El enfoque descrito aquí puede ser combinado con estas técnicas.

En muchas situaciones, un cambio no anunciado en el ambiente puede ocurrir y es deseable actualizar el controlador para acomodarse a este cambio. Por ejemplo, la comunicación entre un UAV y su base puede perderse y, como resultado, parte de la interfaz en la que el controlador de UAV confía, puede estar deshabilitada. En estos casos, debe realizarse una actualización del controlador inmediatamente ya de esta manera podría ser imposible seguir garantizando los objetivos actuales o incluso los nuevos. En [D'I+14] se presenta un trabajo para degradar cuidadosamente las garantías que están siendo otorgadas por el controlador. Sin embargo, la técnica requiere que el controlador y las especificaciones de todos los niveles de degradación preserve una relación de refinamiento con respecto al controlador y a la especificación actual. Este requerimiento de la técnica presentada puede ser muy restrictiva y no es necesaria bajo el enfoque de esta tesis. Incluso, ellos necesitan conocer a priori todas las capas de degradación, especificarlas y sintetizarlas en tiempo de diseño. Bajo el marco de trabajo de esta tesis, en tiempo de ejecución, podría decidirse una nueva degradación no anticipada, especificando (sin requerir ningún tipo de

refinamiento entre $E$ y $E'$), sintetizando e instalando el nuevo controlador $C_u$. En conclusión, el trabajo presentado aquí podría verse como una generalización de la degradación presentada en [D'I+14].

La complejidad de tiempo lineal de la solución al problema de ADC cuando es aplicado a ambientes determinísticos provee un argumento analítico a la escalabilidad. Sin embargo, queda por hacer una validación experimental, y en particular, evaluar la necesidad práctica de introducir no-determinismo, ya que esto, puede producir una explosión exponencial.

Este trabajo basado en actualizaciones dinámicas de controladores tiene el potencial de resolver varios de los problemas identificados en [Che+09; Fil+15; She+17], trabajando en un nivel de abstracción más alto, y se complementa con técnicas que logran adaptación utilizando control de variables continuas como en [Ber+15; Fil+14; Fil+11; Mag+13].

En el marco de esta tesis, hemos restringido la discusión para evitar objetivos de liveness en general como parte de la especificación actual o nueva. Esto simplifica la presentación y también ayuda a la resolución de complejidad lineal de los problemas de ADC mediante síntesis, siempre y cuando, los ambientes a controlar sean determinísticos. Sin embargo, es posible permitir más expresividad en $G$, $G'$ y $T$ sin incurrir en la penalidad completa de resolver problemas de control (2EXPTIME-COMPLETE). Sería posible, por ejemplo, reformular la Definición 3.2.1 para permitir especificaciones $G$, $G'$ y $T$ que incluyan subformulas de la forma $\Box\Diamond\varphi$. Este criterio de aceptación de büchi extiende significativamente la expresividad mientras que mantiene la complejidad a un costo polinomial.

Como ya hemos mencionado, la solución al problema de ADC fue originalmente propuesto en [Nah+16]. En esta presentación, reformulé completamente el problema de actualización dinámica con LTS y FLTL. En [Nah+16], lo definimos sobre labelled transition Kripke Structures (LTKS). La razón por la cual realizamos este cambio es que en la primera versión [Nah+16], se necesitaba que la nueva especificación

subsume a la vieja especificación (en términos de el universo de proposiciones necesitadas para identificar unívocamente un estado). Eso significa que por cada vez que se actualiza el sistema, las especificaciones van creciendo en cuanto a cantidad de proposiciones de los LTKS. Esta restricción no es necesaria en la formalización actual.

## 7.2 Procesos de Negocio

El problema de reconfigurar un proceso de negocio fue estudiado extensivamente bastante tiempo atrás [Ell+95]. En [VAS00] los autores proveen una clasificación de errores potenciales que resultan de realizar cambios de procesos. En [Rin+04] se presenta una clasificación de diferentes tipos de criterios de correctitud que son garantizados por distintas técnicas de cambios dinámicos. También en [Sch+08] se presenta una taxonomía de razones por las cuales se requeriría una reconfiguración de procesos. A su vez, se ha dado soporte metodológico y automatizado para reconfiguraciones de procesos de negocio. Trabajos como [Ell+95; Aal01; BO98] consideran a la reconfiguración de procesos como un problema de definir transiciones dinámicas desde un estado del workflow actual a otro estado en el nuevo workflow. Sin períodos de transición, los cambios pueden particionarse en inmediatos o demorados [Ell+95]. Una visión distinta sobre reconfiguraciones es versionado de workflows (e.g., [KG99; ZL]), donde muchas versiones de workflows son precomputados y ejecutados simultáneamente. En todos los casos, y en contraste con nuestro trabajo, no es considerada la noción del período de transición en la cual se necesitan implementar actividades remediadoras que no cumplen ni con el workflow actual ni con el workflow nuevo.

Para razonar sobre reconfiguraciones, nuestro enfoque supone una especificación declarativa de los requerimientos de procesos de negocio (en vez de una descripción operacional en el forma de un workflow). Anteriormente se han estudiado enfoques de modelado declarativo de procesos de negocio. El lenguaje ConDec [PA06] fue introducido para modelar procesos de negocio basándose en linear temporal logic

(LTL [Pnu77]). En [HM11], los autores proponen una semántica operacional para un lenguaje basado en grafos declarativos, con su respectiva herramienta [Mar+16]. Esta herramienta permite ejecutar el workflow subyacente a la especificación descrita con un modelo DCR Graph. Descripciones basadas en reglas para requerimientos de procesos de negocio también fueron propuestas en el pasado (e.g., [MB+10; Vas+16]). Estas descripciones son naturalmente ejecutables. Ambas soportan cambios de reglas mientras se está ejecutando el workflow, sin embargo, no dan soporte para entender o garantizar propiedades de reconfiguración. Por lo tanto, entender si es necesario un cambio inmediato o uno demorado debe hacerse previamente a introducir la nueva regla. El enfoque propuesto en esta presentación requiere una descripción declarativa de requerimientos de procesos de negocio en un lenguaje general (FLTL) y provee garantías sobre el proceso de reconfiguración. La elección de DCR Graphs como punto de partida es accidental, pudiéndose aplicar una traducción similar para cualquier otro lenguaje declarativo de especificación de procesos de negocio.

La construcción automática de modelos operacionales o ejecutables a partir de requerimientos declarativos también fue estudiado extensivamente, incluyendo trabajos en control de supervisión [RW89], síntesis de diseños reactivos [PR89] y planificación automática [Cim+03]. El trabajo presentado en esta tesis construye sobre síntesis de controladores de eventos discretos y en particular sobre el trabajo [D'I+10] que utiliza Labelled Transition Systems y Fluent Linear Temporal Logic como valor de entrada de la síntesis. Todos los resultados presentados en el área de procesos de negocios están intrínsecamente asociados a los resultados presentados en [Nah+18] donde se presenta la técnica de actualizar un sistema reactivo en tiempo de ejecución (similarmente a lo presentado en el Capítulo 3). Fue necesario adaptar y aplicar esta técnica en el contexto de reconfiguración de procesos de negocio para especificaciones de DCR Graphs.

# 8 Conclusiones

> *It was not magic!*
>
> — **Cristina Fernandez de Kirchner**
> (Abogada y política)

El objetivo general de esta tesis ha sido la exploración y el desarrollo de técnicas de síntesis de controladores para solucionar problemas de actualización de procesos en general, ya sea, procesos que describen el comportamiento de un sistema de software, un sistema reactivo o bien un proceso de negocio. Si bien todos ellos suelen tener una naturaleza distinta, todos ellos refieren a una estructura de actividades/eventos que deben ser llevadas a cabo en un orden especifico para satisfacer un requerimiento definido por los stakeholders del proyecto. Para lograr el conjunto de contribuciones de esta tesis, fue necesario entender los problemas que se generan usualmente a la hora de actualizar un sistema y también, entender cuales son los escenarios en los cuales es más necesario aplicar las técnicas desarrolladas. Entender cuales son los problemas de los que diseñan procesos manualmente para comprender donde la automatización puede sacar más ventaja fue otra forma de explorar el posible impacto de las técnicas desarrolladas en esta tesis.

## 8.1 Contribuciones

En esta tesis, definimos el criterio de correctitud para una actualización dinámica de controladores. Presentamos una solución al problema de actualizar dinámicamente un controlador para satisfacer este criterio basado en síntesis de controladores. Esta solución garantiza que se alcance un punto en donde la nueva especificación vale y, a su vez, se satisfacen los requerimientos de transición provistos por el usuario. Además, la técnica propuesta toma control del sistema que está ejecutando

satisfaciendo la vieja especificación y lo guía hacia un estado seguro desde el cual la actualización puede empezar, garantizando también, que la actualización va a ocurrir eventualmente satisfaciendo la nueva especificación.

En cuanto a los procesos de negocio logramos definir formalmente el problema de reconfiguración de procesos de negocio y presenté una técnica automática que construye una reconfiguración de workflows que garantiza los mismos objetivos detallados anteriormente. Lo interesante de este enfoque es la utilización, como punto de partida, la especificación en un lenguaje declarativo ajeno a nuestra técnica. El hecho de que el usuario utilice un lenguaje con el que está habituado para especificar un problema de reconfiguración acerca la técnica producida en esta tesis a los usuarios. Muchos proyectos científicos carecen del requerimiento no funcional de la usabilidad e intentamos acortar esta brecha realizando una traducción de DCR graphs a problemas de control. Por lo tanto, el usuario debe definir una descripción de los requerimientos del proceso de negocio actual y los requerimientos nuevos sumado a una propiedad LTL que actuará como requerimiento de transición. La técnica producida permite realizar cambios inmediatos y demorados tal y como el estado del arte los detalla. Sin embargo, también produce algunos escenarios de reconfiguración en los cuales hay un período entre ambos procesos de negocio en el cual se ejecutan actividades especificas del dominio preparatorias o remediadores. El resultado es un workflow que al ser intercambiado con el actual guía la transición hacia un nuevo procesos de negocio que propicia los nuevos requerimientos.

## 8.2 Limitaciones

Como todo trabajo de investigación esta tesis y sobretodo la técnica presentada tiene sus limitaciones. La más notoria es la limitante que proviene de elegir MTSA como herramienta para el computo de la síntesis. Si bien MTSA nos otorga una gran facilidad a la hora de definir problemas gracias a su integración con el lenguaje FLTL, también nos prohíbe especificar problemas en otro marco de trabajo. Por ejemplo, para computar reconfiguraciones de procesos de negocio tuvimos que hacer una tra-

ducción a problemas de control (como son planteados en la herramienta MTSA). Eso significa tener que utilizar tres fluents por cada actividad para mantener la valuación del marking del DCR graph. Como se puede ver en la Definición 2.4.3, luego de la ejecución de una actividad el marking puede variar mucho y esto significa tener que apagar/prender muchos fluents. Para representar este cambio significa que se debe transicionar por muchos eventos que apagan/prenden cada valor que es modificado. Simplificar todas estas transiciones a una sola que modifique todo lo necesario sería una optimización muy valiosa. Hemos explorado la utilización de otras herramientas como Ratsy o Spin sin éxito. La problemática descrita anteriormente sale a la luz cuando queremos hacer una reconfiguración de mas de 20 actividades en cada DCR graph (40 actividades total). El cómputo de la síntesis no escala perdiendo así la posibilidad de obtener un valor de salida.

## 8.3 Trabajo Futuro

Como trabajo futuro se puede planear la exploración de cómo aumentar la expresividad de los objetivos para incluir liveness pero sin tener que pagar el precio de la síntesis general. Muchas misiones en robótica necesitan actualizarse y estas poseen objetivos del sistema actual que son de liveness, por ejemplo, ir infinitamente a la base para recargar la batería. Actualmente no soportamos la actualización de este tipo de planes pero estamos explorando la posibilidad de hacerlo en la brevedad. La forma de lograrlo sería utilizando dos controladores iniciales: el controlador que solo proporciona safe y uno que proporciones los requerimientos safe y live. Al momento del *hotSwap* el controlador (live y safe) debe saltar a uno que solo garantiza las propiedades safe para iniciar la transición y llegar a un estado donde los nuevos requerimientos valen (estos deberían tener una parte safe y otra live también).

En cuanto al área de procesos de negocio, aun queda bastante por hacer. Aquí se presenta una traducción para poder utilizar la técnica de actualización dinámica de controladores de eventos discretos en procesos de negocios que fueron definidos en un lenguaje declarativo amigable para el usuario. Sin embargo, esta traducción

no es automática, debilitando el poder que tiene la técnica de ADC. Teniendo una traducción automática podríamos disparar una actualización directamente teniendo los DCR graphs.

Finalmente, también se puede apuntar los próximos pasos a investigar la integración con otros enfoques que proveen adaptación de capacidades de sistemas de software complejos de alto nivel, tales como técnicas de aprendizaje de comportamientos de ambientes en tiempo de ejecución y adaptación de propiedades cuantitativas.

# Bibliografía


[Aal01] Wil MP van der Aalst. „Exterminating the dynamic change bug: A concrete approach to support workflow change“. En: *Information Systems Frontiers* 3.3 (2001), págs. 297-317 (vid. pág. 121).

[AH01] Luca de Alfaro y Thomas A. Henzinger. „Interface automata“. En: *ESEC / SIGSOFT FSE*. ACM, 2001, págs. 109-120 (vid. págs. 19, 20).

[AH04] Wil van der Aalst y Kees van Hee. *Workflow Management: Models, Methods, and Systems*. Cambridge, MA, USA: MIT Press, 2004 (vid. pág. 3).

[Alv+17] Frederico Alvares, Eric Rutten y Lionel Seinturier. „A domain-specific language for the control of self-adaptive component-based architecture“. En: *Journal of Systems and Software* (2017) (vid. pág. 34).

[An+15] Shengwei An, Xiaoxing Ma, Chun Cao, Ping Yu y Chang Xu. „An event-based formal framework for dynamic software update“. En: *IEEE International Conference on Software Quality, Reliability and Security*. IEEE. 2015, págs. 173-182 (vid. págs. 114-116).

[AR09] Austin Anderson y Julian Rathke. „Migrating Protocols in Multi-threaded Message-passing Systems“. En: *Proc. of the 2Nd International Workshop on Hot Topics in Software Upgrades*. HotSWUp '09. ACM, 2009, 8:1-8:5 (vid. págs. 1, 113).

[Ars+07] Naveed Arshad, Dennis Heimbigner y Alexander L. Wolf. „Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems“. En: *Software Quality Journal* 15.3 (2007), págs. 265-281 (vid. pág. 118).

[Ban+10] Filippo Banno, Daniele Marletta, Giuseppe Pappalardo y Emiliano Tramontana. „Handling consistent dynamic updates on distributed systems“. En: *Proc. of the IEEE Symp. on Computers and Communications (ISCC)*, IEEE. 2010, págs. 471-476 (vid. pág. 113).

[Bar+16] Luciano Baresi, Carlo Ghezzi, Xiaoxing Ma y Valerio Panzica La Manna. „Efficient Dynamic Updates of Distributed Components through Version Consistency“. En: *IEEE Transactions on Software Engineering* (2016) (vid. pág. 118).

[Ber+15] Andreas Bergen, Nina Taherimakhsousi y Hausi A Müller. „Adaptive management of energy consumption using adaptive runtime models“. En: *Proc of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE. 2015, págs. 120-126 (vid. pág. 120).

[BG10] Luciano Baresi y Carlo Ghezzi. „The Disappearing Boundary Between Development-time and Run-time“. En: *Proc. of the FSE/SDP Workshop on Future of Software Engineering Research*. ACM, 2010, págs. 17-22 (vid. págs. 2, 114, 115).

[Blo+09] Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger y Barbara Jobstmann. „Better Quality in Synthesis through Quantitative Objectives“. En: *Proc. of the 21st International Conference on Computer on Aided Verification, CAV 2009, Grenoble, France.* Ed. por Ahmed Bouajjani y Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, págs. 140-156 (vid. pág. 115).

[BO98] Eric Badouel y Javier Oliver. „Reconfigurable nets, a class of high level Petri nets supporting dynamic changes within workflow systems“. Tesis doct. INRIA, 1998 (vid. pág. 121).

[Bpma] *Business Process Management Academic Initiative*. `https://bpmai.org/` (vid. págs. 8, 106).

[Bpmb] *International Conference on Business Process Management*. IEEE, 2003–2020 (vid. pág. 9).

[Bra+15] Víctor Braberman, Nicolás D'Ippolito, Jeff Kramer, Daniel Sykes y Sebastian Uchitel. „MORPH: A Reference Architecture for Configuration and Behaviour Self-adaptation“. En: *Proc. of the 1st International Workshop on Control Theory for Software Engineering*. CTSE 2015. 2015, págs. 9-16 (vid. págs. 76, 118).

[Che+07] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang y Pen-Chung Yew. „Polus: A powerful live updating system“. En: *Proc of the 29th International Conference on Software Engineering*. IEEE Computer Society. 2007, págs. 271-281 (vid. pág. 113).

[Che+09] Betty HC Cheng, Rogerio De Lemos, Holger Giese, Paola Inverardi, Jeff Magee y col. „Software engineering for self-adaptive systems: A research roadmap“. En: *Software engineering for self-adaptive systems*. Springer, 2009, págs. 1-26 (vid. pág. 120).

[Che+11] Haibo Chen, Jie Yu, Chengqun Hang, Binyu Zang y Pen-Chung Yew. „Dynamic Software Updating Using a Relaxed Consistency Model“. En: *IEEE Transactions on Software Engineering* 37.5 (2011), págs. 679-694 (vid. pág. 113).

[Cim+03] A. Cimatti, M. Pistore, M. Roveri y P. Traverso. „Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking“. En: *Artificial Intelligence* 147 (2003) (vid. pág. 122).

[Cio+17] Daniel Ciolek, Víctor A. Braberman, Nicolás D'Ippolito, Nir Piterman y Sebastián Uchitel. „Interaction Models and Automated Control under Partial Observable Environments“. En: *IEEE Transactions on Software Engineering* 43.1 (2017), págs. 19-33 (vid. pág. 46).

[D'I+08] Nicolás D'Ippolito, Darío Fischbein, Marsha Chechik y Sebastián Uchitel. „MTSA: The Modal Transition System Analyser“. En: *IEEE/ACM International Conference Automated Software Engineering*. 2008, págs. 475-476 (vid. págs. 46, 56, 63, 89, 106).

[D'I+10] Nicolás D'Ippolito, Víctor Braberman, Nir Piterman y Sebastián Uchitel. „Synthesis of Live Behaviour Models“. En: *Foundations of Software Engineering*. Santa Fe, New Mexico, USA: ACM, 2010, págs. 77-86 (vid. págs. 64, 122).

[D'I+13] Nicolás D'Ippolito, Víctor Braberman, Nir Piterman y Sebastián Uchitel. „Synthesising Non-Anomalous Event-Based Controllers for Liveness Goals". En: *ACM Transactions on Software Engineering and Methodology* 22 (1 2013) (vid. págs. 6-8, 20).

[D'I+14] Nicolás D'Ippolito, Víctor Braberman, Jeff Kramer y col. „Hope for the Best, Prepare for the Worst: Multi-tier Control for Adaptive Systems". En: *Proc. of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, págs. 688-699 (vid. págs. 119, 120).

[Dcr] *DCR Solutions* (vid. pág. 22).

[Ell+95] Clarence Ellis, Karim Keddara y Grzegorz Rozenberg. „Dynamic change within workflow systems". En: *Proceedings of conference on Organizational computing systems*. ACM. 1995, págs. 10-21 (vid. págs. 4, 82, 107, 110, 121).

[Fil+11] Antonio Filieri, Carlo Ghezzi, Alberto Leva y Martina Maggio. „Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements". En: *Proc. of the 26th IEEE/ACM International Conference on Automated Software Engineering, 2011*. IEEE Computer Society. 2011, págs. 283-292 (vid. pág. 120).

[Fil+14] Antonio Filieri, Henry Hoffmann y Martina Maggio. „Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees". En: *Proc of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, págs. 299-310 (vid. pág. 120).

[Fil+15] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos y col. „Software engineering meets control theory". En: *Proc. of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press. 2015, págs. 71-82 (vid. pág. 120).

[Ghe+12] Carlo Ghezzi, Joel Greenyer y Valerio Panzica La Manna. „Synthesizing dynamically updating controllers from changes in scenario-based specifications". En: *Proc. of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE. 2012, págs. 145-154 (vid. págs. 3, 34, 35, 64, 69-71, 73, 114-116).

[GM03] Dimitra Giannakopoulou y Jeff Magee. „Fluent model checking for event-based systems". En: *Proc. of the 11th ACM SIGSOFT International Symposium on Foundations of software engineering*. ESEC/FSE-11. 2003, págs. 257-266 (vid. págs. 16, 17, 64).

[GR96] Humberto E. Garcia y Asok Ray. „State-space supervisory control of reconfigurable discrete event systems". En: *International Journal of Control* 63.4 (1996), págs. 767-797 (vid. págs. 1, 117).

[Gre+13] Joel Greenyer, Christian Brenner, Maxime Cordy, Patrick Heymans y Erika Gressi. „Incrementally Synthesizing Controllers from Scenario-based Product Line Specifications". En: *Proc. of the 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. 2013, págs. 433-443 (vid. págs. 3, 118).

[Gup+96] Deepak Gupta, Pankaj Jalote y Gautam Barua. „A formal framework for on-line software version change". En: *IEEE Transactions on Software Engineering* 22.2 (1996) (vid. págs. 1, 113).

[Gup94] Deepak Gupta. „On-line software version change“. Tesis doct. 1994 (vid. pág. 114).

[Hay+12] Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster y Jeffrey S. Foster. „Specifying and Verifying the Correctness of Dynamic Software Updates“. En: *Proc. of the 4th International Conference on Verified Software: Theories, Tools, Experiments*. VSTTE'12. Philadelphia, PA: Springer-Verlag, 2012, págs. 278-293 (vid. pág. 114).

[HC13] Petr Hosek y Cristian Cadar. „Safe Software Updates via Multi-version Execution“. En: *Proc. of the International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, págs. 612-621 (vid. pág. 113).

[HM11] Thomas T. Hildebrandt y Raghava Rao Mukkamala. „Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs“. En: *Electronic Proceedings in Theoretical Computer Science* 69 (2011), 59–73 (vid. págs. 5, 8, 80, 81, 122).

[Ics] *International Conference on Software Engineering*. ACM/IEEE, 1975–2020 (vid. pág. 9).

[KC03] Jeffrey O. Kephart y David M. Chess. „The Vision of Autonomic Computing“. En: *Computer* 36.1 (ene. de 2003) (vid. pág. 30).

[Kel76] Robert M. Keller. „Formal verification of parallel programs“. En: *Communications of the ACM* 19 (7 1976), págs. 371-384 (vid. pág. 63).

[KG99] Markus Kradolfer y Andreas Geppert. „Dynamic workflow schema evolution based on workflow type versioning and workflow migration“. En: *International Journal of Cooperative Information Systems*. 1999 (vid. pág. 121).

[KGP08] Hadas Kress-Gazit y George J. Pappas. „Automatically synthesizing a planning and control subsystem for the DARPA urban challenge“. En: *2008 IEEE International Conference on Automation Science and Engineering*. 2008, págs. 766-771 (vid. pág. 2).

[KM90] Jeff Kramer y Jeff Magee. „The Evolving Philosophers Problem: Dynamic Change Management“. En: *IEEE Transactions on Software Engineering* 16.11 (nov. de 1990), págs. 1293-1306 (vid. págs. 1, 3, 113).

[Kon+04] Michael Kontitsis, Kimon P Valavanis y Nikos Tsourveloudis. „A UAV vision system for airborne surveillance“. En: *Proc. of the IEEE International Conference on Robotics and Automation, 2004. ICRA'04.* Vol. 1. 2004, págs. 77-83 (vid. pág. 64).

[Kuc+17] Omer Kucuk, Ozer Topaloglu, Arif Oguz Altunel y Mehmet Cetin. „Visibility analysis of fire lookout towers in the Boyabat State Forest Enterprise in Turkey“. En: *Environmental monitoring and assessment* 189.7 (2017), pág. 329 (vid. pág. 75).

[Lam94] Leslie Lamport. „The Temporal Logic of Actions“. En: *ACM Trans. Program. Lang. Syst.* 16.3 (mayo de 1994), 872–923 (vid. pág. 96).

[LL00] Axel van Lamsweerde y Emmanuel Letier. „Handling Obstacles in Goal-Oriented Requirements Engineering“. En: *IEEE Transactions on Software Engineering* 26.10 (oct. de 2000), págs. 978-1005 (vid. pág. 16).

[LL95] Claus Lewerentz y Thomas Lindner, eds. *Formal Development of Reactive Systems - Case Study Production Cell*. London, UK, UK: Springer-Verlag, 1995 (vid. págs. 25, 64, 65).

[Mag+13] Martina Maggio, Henry Hoffmann, Marco D. Santambrogio, Anant Agarwal y Alberto Leva. „Power Optimization in Embedded Systems via Feedback Control of Resource Allocation“. En: *IEEE Transactions on Control Systems Technology* 21.1 (2013), págs. 239-246 (vid. pág. 120).

[Mar+16] Morten Marquard, Muhammad Shahzad y Tijs Slaats. „Web-based modelling and collaborative simulation of declarative processes“. En: *International Conference on Business Process Management*. Springer. 2016, págs. 209-225 (vid. págs. 8, 106, 122).

[MB+10] Jose F Mejia Bernal, Paolo Falcarin, Maurizio Morisio y Jia Dai. „Dynamic context-aware business process: a rule-based approach supported by pattern identification“. En: *Symposium On Applied Computing*. 2010, págs. 470-474 (vid. pág. 122).

[Mil80] Robin Milner. „A calculus of communicating systems“. En: *LNCS* 92 (1980) (vid. pág. 89).

[MK06] Jeff Magee y Jeff Kramer. *Concurrency: state models & Java programs*. Wiley New York, 2006 (vid. pág. 63).

[MR07] Kristis Makris y Kyung Dong Ryu. „Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels“. En: *ACM SIGOPS Operating Systems Review* 41.3 (2007), págs. 327-340 (vid. págs. 114, 115).

[Muk12] Raghava Rao Mukkamala. „A Formal Model For Declarative Workflows“. Tesis doct. IT University of Copenhagen, 2012 (vid. págs. 22, 81, 106).

[Myt] *MTSA Synthesis Tool and Examples* (vid. págs. 46, 55, 64, 106).

[Nah+16] Leandro Nahabedian, Víctor Braberman, Nicolás D'Ippolito y col. „Assured and correct dynamic update of controllers“. En: *Proc. of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM. 2016, págs. 96-107 (vid. págs. 9, 25, 120).

[Nah+18] Leandro Nahabedian, Víctor Braberman, Nicolás D'Ippolito y col. „Dynamic Update of Discrete Event Controllers“. En: *IEEE Transaction on Software Engineering 2018* (2018), págs. 1-1 (vid. págs. 8, 9, 25, 122).

[Nah+19] Leandro Nahabedian, Víctor Braberman, Nicolás D'ippolito, Jeff Kramer y Sebastián Uchitel. „Dynamic Reconfiguration of Business Processes“. En: *International Conference on Business Process Management*. Springer. 2019, págs. 35-51 (vid. págs. 9, 80).

[Nah17] Leandro Nahabedian. „Dynamic Update of Business Process Management“. En: *IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, págs. 413-416 (vid. pág. 9).

[Nea+06] Iulian Neamtiu, Michael W. Hicks, Gareth Paul Stoyle y Manuel Oriol. „Practical dynamic software updating for C“. En: *Proc. of the Conference on Programming Language Design and Implementation*. 2006, págs. 72-83 (vid. págs. 113-115).

[NH09] Iulian Neamtiu y Michael Hicks. „Safe and timely updates to multi-threaded programs“. En: *ACM Sigplan Notices*. Vol. 44. 6. ACM. 2009, págs. 13-24 (vid. pág. 116).

[NS15] Anas Nooruldeen y Klaus W. Schmidt. „State Attraction Under Language Specification for the Reconfiguration of Discrete Event Systems“. En: *IEEE Transaction on Automatic Control* 60.6 (2015), págs. 1630-1634 (vid. págs. 1, 117).

[Ors+02] Alessandro Orso, Anup Rao y Mary Jean Harrold. „A technique for dynamic updating of Java software“. En: *Proc of the International Conference on Software Maintenance, 2002*. IEEE. 2002, págs. 649-658 (vid. pág. 113).

[PA06] Maja Pesic y Wil MP Van der Aalst. „A declarative approach for flexible business processes management“. En: *International Conference on Business Process Management*. 2006, págs. 169-180 (vid. págs. 5, 86, 121).

[Pad14] Paderborn. *New Rail Technology Paderborn*. http://www.railcab.de/. Ago. de 2014 (vid. pág. 69).

[Pel+08] Patrizio Pelliccione, Massimo Tivoli, Antonio Bucchiarone y Andrea Polini. „An Architectural Approach to the Correct and Automatic Assembly of Evolving Component-based Systems“. En: *J. Syst. Softw.* 81.12 (dic. de 2008), págs. 2237-2251 (vid. pág. 119).

[Pit+06] Nir Piterman, Amir Pnueli y Yaniv Sa'ar. „Synthesis of reactive (1) designs“. En: *Lecture notes in computer science* 3855 (2006), págs. 364-380 (vid. págs. 6, 7).

[PLM+13] Valerio Panzica La Manna, Joel Greenyer, Carlo Ghezzi y Christian Brenner. „Formalizing correctness criteria of dynamic updates derived from specification changes“. En: *Proc. of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press. 2013, págs. 63-72 (vid. págs. 64, 68, 69, 114-116).

[Pnu77] Amir Pnueli. „The temporal logic of programs“. En: *IEEE Symposium on Foundations of Computer Science*. 1977, págs. 46-57 (vid. págs. 8, 46, 122).

[PR89] Amir Pnueli y Roni Rosner. „On the synthesis of a reactive module“. En: *ACM SIGPLAN Symposium on Principles of Programming Languages*. 1989, págs. 179-190 (vid. pág. 122).

[Ram+10] Andres J. Ramirez, Betty H.C. Cheng, Philip K. McKinley y Benjamin E. Beckmann. „Automatically Generating Adaptive Logic to Balance Non-functional Tradeoffs During Reconfiguration“. En: *Proc. of the 7th International Conference on Autonomic Computing*. ICAC '10. Washington, DC, USA: ACM, 2010, págs. 225-234 (vid. págs. 114, 115).

[Rin+04] Stefanie Rinderle, Manfred Reichert y Peter Dadam. „Correctness criteria for dynamic changes in workflow systems—-a survey“. En: *Data & Knowledge Engineering* 50.1 (2004), págs. 9-34 (vid. pág. 121).

[RW89] Peter JG Ramadge y W Murray Wonham. „The control of discrete event systems“. En: *Proceedings of the IEEE* 77.1 (1989), págs. 81-98 (vid. págs. 6, 7, 122).

[Sam+08] Rupa Sampath, Houshang Darabi, Ugo Buy y Jing Liu. „Control reconfiguration of discrete event systems with dynamic control specifications“. En: *IEEE Transactions on Automation Science and Engineering* 5.1 (2008), págs. 84-100 (vid. págs. 1, 117, 118).

[Sch+08] Helen Schonenberg, Ronny Mans, Nick Russell, Nataliya Mulyar y Wil MP van der Aalst. „Towards a Taxonomy of Process Flexibility.“ En: *International Conference on Advanced Information Systems Engineering*. Vol. 344. 2008, págs. 81-84 (vid. pág. 121).

[Sea] *ICSE Symp. on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*. ACM/IEEE, 2006–2020 (vid. págs. 1, 9).

[Sei+13] Habib Seifzadeh, Hassan Abolhassani y Mohsen Sadighi Moshkenani. „A survey of dynamic software updating“. En: *Journal of Software: Evolution and Process* 25.5 (2013), págs. 535-568 (vid. pág. 113).

[She+05] Junrong Shen, Xi Sun, Gang Huang y col. „Towards a Unified Formal Model for Supporting Mechanisms of Dynamic Component Update“. En: *SIGSOFT Softw. Eng. Notes* 30.5 (sep. de 2005), págs. 80-89 (vid. pág. 113).

[She+17] Stepan Shevtsov, Mihaly Berekmeri, Danny Weyns y Martina Maggio. „Control-theoretical software adaptation: A systematic literature review“. En: *IEEE Transactions on Software Engineering* (2017) (vid. pág. 120).

[Sto+07] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell y Iulian Neamtiu. „Mutatis Mutandis: Safe and Predictable Dynamic Software Updating“. En: *ACM Transactions on Programming Languages and Systems* 29.4 (2007) (vid. pág. 113).

[Sub+09] Suriya Subramanian, Michael Hicks y Kathryn S. McKinley. „Dynamic Software Updates: A VM-centric Approach“. En: *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*. PLDI ’09. Dublin, Ireland: ACM, 2009, págs. 1-12 (vid. pág. 113).

[Syk+08] Daniel Sykes, William Heaven, Jeff Magee y Jeff Kramer. „From Goals to Components: A Combined Approach to Self-Management“. En: *Proc. of the Workshop on Software Engineering for Adaptive and Self-Managing Systems*. 2008 (vid. pág. 118).

[Syk+10] Daniel Sykes, William Heaven, Jeff Magee y Jeff Kramer. „Exploiting non-functional preferences in architectural adaptation for self-managed systems“. En: *Proc. of the 2010 ACM Symp. on Applied Computing*. ACM. 2010, págs. 431-438 (vid. pág. 115).

[Syk+13] Daniel Sykes, Domenico Corapi, Jeff Magee y col. „Learning Revised Models for Planning in Adaptive Systems“. En: *Proc. of the International Conference on Software Engineering*. San Francisco, CA, USA: IEEE Press, 2013, págs. 63-71 (vid. pág. 119).

[Taj+10] Hossein Tajalli, Joshua Garcia, George Edwards y Nenad Medvidovic. „PLASMA: a plan-based layered architecture for software model-driven adaptation“. En: *Proc. of the IEEE/ACM International Conference on Automated software engineering*. 2010, págs. 467-476 (vid. págs. 34, 118).

[Tse] *IEEE Transaction On Software Engineering*. IEEE, 1975–2020 (vid. pág. 9).

[Van+07] Yves Vandewoude, Peter Ebraert, Yolande Berbers y Theo D’Hondt. „Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates“. En: *IEEE Transactions on Software Engineering* 33.12 (2007), págs. 856-868 (vid. págs. 1, 113).

[Vas+16] Olegas Vasilecas, Diana Kalibatiene y Dejan Lavbič. „Rule-and context-based dynamic business process modelling and simulation“. En: *Journal of Systems and Software* (2016) (vid. pág. 122).

[VAS00] Wil MP V.D Aalst y J. Stefan. „Dealing with workflow change: identification of issues and solutions“. En: *Computer systems science and engineering* 15.5 (2000), págs. 267-276 (vid. págs. 3, 27, 109, 121).

[Vro+11] Pieter Vromant, Danny Weyns, Sam Malek y Jesper Andersson. „On Interacting Control Loops in Self-adaptive Systems“. En: *Proc. of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '11. Waikiki Honolulu, HI, USA: ACM, 2011, págs. 202-207 (vid. pág. 119).

[ZC05] Ji Zhang y Betty H. C. Cheng. „Specifying Adaptation Semantics“. En: *Proc. of the 2005 Workshop on Architecting Dependable Systems*. WADS '05. ACM, 2005, págs. 1-7 (vid. págs. 2-4, 64, 72, 113, 114, 116).

[ZC06] Ji Zhang y Betty HC Cheng. „Model-based development of dynamically adaptive software“. En: *Proc. of the 28th International Conference on Software engineering*. ACM. 2006, págs. 371-380 (vid. págs. 3, 64, 72, 114, 115).

[ZL] Xiaohui Zhao y Chengfei Liu. „Version management in the business process change context“. En: *International Conference on Business Process Management 2007*, págs. 198-213 (vid. pág. 121).

# Índice de figuras


1.1 Esquema de la reconfiguración de procesos de negocio. . . . . 9

2.1 Semántica del operador de satisfacción. . . . . 18

3.1 Actualización dinámica de controladores con diferentes requerimientos de transición (de arriba a abajo): *stopOldSpec* $\Rightarrow$ *NewSpecStarted*, *startNewSpec* $\Rightarrow$ *OldSpecStopped* y (*OldSpecStopped*$\wedge\neg$*NewSpecStarted*) $\Rightarrow \varphi$, donde $\varphi$ es una propiedad de safety. . . . . 30

3.2 Hotswap de controlador por otro para una cadena de montaje (una reducción – dos herramientas. Ningún controlador tiene herramienta limpiadora). $(C \not\downarrow_H^{hotSwap} C')$. Las lineas punteadas representan la transición de hotswap inducida por $H$. . . . . 32

3.3 Supuestos del ambiente $(E \not\downarrow_R^{reconfigure} E')$ para la actualización dinámica del caso de estudio del RailCab [Ghe+12]. Las transiciones de reconfiguración con linea punteada indican como los viejos supuestos son mapeados hacia los nuevos como es definido en la realación $R$. . . 35

3.4 Esquema informal de las tres fases del ambiente $E_u$. Primero como un ambiente controlado por $C$ pero no controlable por $C'$ ($[C\|E]_f$), luego como ambiente no controlado que se comporta como $E$ y finalmente como un ambiente no controlado que se comporta como $E'$. . . . . 41

3.5 LTS que define la ocurrencia de los eventos *stopOldSpec* y *startNewSpec*. Antes del evento *hotSwap* ninguno de los eventos *stopOldSpec* o *startNewSpec* puede suceder. Luego de este, pueden suceder en cualquier orden. . . . . 42

3.6 Dibujo informal de las tres fases del controlador $C_u$. Primero comportándose exactamente como $C$, y por lo tanto, garantizando $\Box G$ ($C_G$), luego controlando el período de transición ($C_T$), y luego cuando ambos eventos *stopOldSpec* y *startNewSpec* han sucedido, garantiza $\Box G'$ ($C_{G'}$). Notar que, por simplicidad, *reconfigure* no está dibujado. . . . . 45

3.7 Soporte para prueba. Dibujo informal de $C_u = ([C_u]_{cut} \not\downarrow_Q^{hs} C'_u)$. Muestro que para los estados $c$ y $q$ tal que $[C_u]_{cut}(q) \sim [C]_f(c)$, entonces, $H$ vincula a $c$ con un estado $c'_u$ en $C'_u$ de acuerdo con la relación $Q$. . . . . 48

4.1 Captura de pantalla de la herramienta MTSA donde se muestra la definición de un ambiente utilizando el lenguaje FSP. . . . . 56

4.2 Captura de pantalla de la herramienta MTSA donde se muestra la de un problema de control LTS. . . . . 58

4.3 Especificación de un problema de actualización dinámica de controladores en la herramienta MTSA. . . . . 60

5.1 Controlador actualizador para un versión reducida de una cadena de montaje (solo dos herramientas) con sus tres fases. $C_G$ y $C_{G'}$ garantizan la vieja y la nueva especificación respectivamente. La parte del medio garantiza progreso entre las dos mientras satisface $T_\emptyset$. . . . . 67
5.2 Controlador Safe pero no live $C_u^S$ para un versión reducida de una cadena de montaje (sólo con dos herramientas) con sus tres fases. A diferencia de la Figura 5.1, la fase intermedia tiene ciclos (e.g., transiciones en rojo) donde no se cumple la finalización de la actualización. 68
5.3 Linea de tiempo del Railcab. Zonas en letra negra, eventos controlables en letra azul y eventos monitoreables en letra roja. . . . . 70

6.1 DCR graph para el proceso del hospital. . . . . 81
6.2 DCR graph model para el nuevo procesos del hospital. . . . . 81
6.3 Workflow del proceso del hospital. . . . . 82
6.4 Workflow del nuevo proceso del hospital. . . . . 82
6.5 LTS $Happens(s)$ que restringe la ocurrencia de $s\_happened$. . . . . 85
6.6 LTS $Turns$ que define los turnos del controlador y el ambiente. . . . . 85
6.7 Si $C$ es un controlador, entonces cada traza $\pi \in C$ tiene una traza abstracta $abs(\pi)$. Para todo $i \in \mathbb{N}$, $w_i$ es una subtraza que solo tiene eventos disable, $enableAll$ o $menu$. . . . . 90
6.8 Workflow de reconfiguración con requerimiento de transición $T_h$. pm: “prescribe medicine”, s: “sign”, gm: “give medicine”, dt: “don’t trust”, et: “examine test”, rt: “receive test”. . . . . 98
6.9 Reconfiguración de DCR para requerimiento de transición $T_h$ y $\alpha = pm$. 102
6.10 Estructura de la Reconfiguración de DCR. . . . . 102

# Índice de cuadros

5.1 Reporte de la síntesis de la actualización dinámica controladores. $|d|$ y $|d'|$ son los números de fluents en las especificaciones vieja y nueva, respectivamente. $|G|$ y $|G'|$ son los números de operadores lógicos en las especificaciones vieja y nueva, respectivamente. $|T|$ es el número de operadores lógicos del requerimiento de transición. $|E_u||G_u|$ es el número de estados de la composición de $E_u$ y la representación de $G_u$ en forma de autómata. El guión (-) indica un problema de control no realizable. . . . . 77

6.1 Resumen de los casos de estudio ejecutados. . . . . 111