



UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE CIENCIAS EXACTAS Y NATURALES

DEPARTAMENTO DE COMPUTACIÓN

# Hot-swap. Una técnica para la generación y actualización automática de controladores discretos en tiempo de ejecución

Tesis presentada para optar al título de  
Licenciado en Ciencias de la Computación

Leandro Ezequiel Nahabedian

Director: Nicolás Roque D'Ippolito

Buenos Aires, Argentina 2014



# HOT-SWAP: UNA TÉCNICA PARA LA GENERACIÓN Y ACTUALIZACIÓN AUTOMÁTICA DE CONTROLADORES DISCRETOS EN TIEMPO DE EJECUCIÓN

Es esperado que muchos sistemas se ejecuten continuamente mientras el ambiente cambia y los requerimientos evolucionan, por lo tanto las implementaciones de dichos sistemas deben ser actualizados dinámicamente para satisfacer los cambios de requerimientos, respetando los cambios del ambiente. Lo complejo de este paso, es poder determinar en que puntos de la ejecución previa es seguro hacer la actualización, y si es seguro, como deberá seguir ejecutando el nuevo sistema. A su vez, existe la necesidad de desarrollar técnicas que permitan actualizar un sistema sin frenar o interrumpir la ejecución del sistema.

Tanto la máquina, como el ambiente y los requerimientos, pueden ser interpretados por modelos de comportamiento que son estructuras formales que definen acciones que pueden suceder. Luego, con la técnica de síntesis de controladores podremos obtener modelos de forma correcta debido a que son obtenidos mediante construcciones.

En esta tesis presentaremos una solución general, que no sólo produce un controlador para la nueva especificación y maneja la transición de uno a otro, sino que también, fuerza al sistema a que alcance un estado en el cual la transición puede ocurrir de manera segura.

Finalizando, desarrollaremos varios casos de estudio utilizando la herramienta MTSA (Modal Transition System Analyser) que nos permite efectuar la síntesis de controladores. Los casos de estudios fueron tomados de trabajos previos sobre actualización dinámica y sistemas adaptables lo que nos permite hacer un trabajo comparativo entre nuestros resultados y los obtenidos previamente.

**Palabras claves:** *Síntesis de controladores; LTS; Actualización dinámica; Sistemas Adaptables.*



## AGRADECIMIENTOS

En primer lugar, quisiera expresar mi gratitud al Dr. Nicolás D'Ippolito, quien me guió por este camino desde el inicio de mi tesis hasta estos días. Su experiencia y la forma de transmitirla me fueron de gran ayuda. Tanto él como el Dr. Sebastián Uchitel me generaron un gran sentimiento de admiración, que sumado a su trato afectuoso posibilitaron que diera lo mejor de mí para iniciarme en esta etapa académica.

Quiero también agradecer a mis padres que desde el inicio del CBC hasta hoy me dieron su apoyo para seguir siempre adelante a pesar de los malos momentos.

Agradezco a mi hermano Luciano y mi cuñada Alejandra quienes me dan una cuota de alegría muy importante cada fin de semana, compartiendo charlas, viendo series, películas o simplemente mediante un abrazo cuando se necesita.

Estoy agradecido a mis abuelos, ejemplo de perseverancia, que llevaron una larga vida juntos, y en especial a mi abuela Beba que siempre confió en mí. Ellos me dieron una infancia llena de juegos y aprendizaje, y un amor incondicional.

No quiero dejar de agradecer a mis amigos: Fer, “El tano”, Lucas, Ger, Pablito, Lea y Pablo, los amigos de la facu, los compañeros de la oficina del Ministerio de Interior, los compañeros de LaFHIS y compañeros de la primaria. Todos ellos logran que cada día de mi vida tenga una razón para reír.

Finalmente, agradezco a todas aquellas personas que me acompañan y acompañaron en el transcurso de este largo camino. Son ellos los que me demuestran que mediante el esfuerzo y el amor los sueños son posibles.



*A mi familia con todo mi amor.*





## Índice general

1..	Introducción . . . . .	1
1.1.	Motivación . . . . .	1
1.2.	Resumen de la contribución . . . . .	2
1.3.	Esquema de tesis . . . . .	3
2..	Fundamentos teóricos . . . . .	5
2.1.	El Mundo y la Máquina . . . . .	5
2.2.	Sistema de Transición Etiquetados (Labelled Transition System) . . . . .	7
2.3.	Lógica Lineal Temporal de Flujos (Fluent Linear Temporal Logic) . . . . .	8
2.4.	Problemas de síntesis de controladores . . . . .	9
2.5.	Juegos de dos jugadores . . . . .	11
2.6.	Resolviendo el problema de control LTS SGR(1) . . . . .	12
2.6.1.	Control LTS SGR(1) a juegos GR(1) . . . . .	13
2.6.2.	Traduciendo la estrategia a un Controlador LTS . . . . .	14
2.6.3.	Algoritmo . . . . .	15
2.7.	Procesos de estados Finitos (Finite State Process) . . . . .	17
3..	MTSA como herramienta de modelado y síntesis . . . . .	21
3.1.	Construcción . . . . .	21
3.2.	Análisis . . . . .	22
3.3.	Modelando problemas de controladores dinámicamente actualizables . . . . .	23
4..	Problema de actualización de controladores dinámicamente . . . . .	27
4.1.	Especificación . . . . .	27
4.2.	La actualización dinámica de controladores como un problema de síntesis de controladores . . . . .	28
4.2.1.	El objetivo del problema de control . . . . .	29
4.2.2.	Modelo del ambiente del problema de control . . . . .	30

4.2.3. El problema de control de la actualización de controladores dinámicamente . . . . .	33
4.3. Resolviendo el problema de actualización dinámica de controladores . . . .	35
5.. Validación . . . . .	37
5.1. Casos de Estudio . . . . .	39
5.1.1. Planta de energía nuclear . . . . .	39
5.1.2. RailCab . . . . .	40
5.1.3. Buscador UAV de vida salvaje . . . . .	40
5.1.4. Production Cell . . . . .	41
5.1.5. Cuadróptero . . . . .	42
5.2. Resultados . . . . .	42
5.2.1. Planta de energía nuclear . . . . .	43
5.2.2. RailCab . . . . .	43
5.2.3. Buscador UAV de vida salvaje . . . . .	44
5.2.4. Production Cell . . . . .	44
5.2.5. Cuadróptero . . . . .	44
6.. Conclusiones y trabajos futuros . . . . .	47
6.1. Discusión y trabajo futuro . . . . .	47
6.2. Conclusiones . . . . .	51

# 1. INTRODUCCIÓN

## 1.1. Motivación

La operación continua, sistemas donde cada una de sus componentes se mantienen operativos todo el tiempo, es un requerimiento común en muchas aplicaciones. Por lo tanto, es necesario desarrollar técnicas ingenieriles que puedan actualizar un sistema, tanto su ambiente como sus requerimientos, sin la necesidad de frenar o interrumpir sus operaciones. Este trabajo ha sido estudiado de diversas maneras, empezando por la actualización dinámica de software [KM90] y más recientemente con el diseño de software adaptable [SEA14].

La pregunta central que intenta solucionar este problema es ¿cuándo es seguro cambiar un componente de software en un sistema que esta corriendo? Una respuesta conservadora a esta pregunta es “cuando los componentes no están involucrados en alguna interacción”; esto fue formalizado introduciendo la noción de **quietud** (*quiescence*) [KM90] y luego **tranquilidad** (*tranquility*) [VEBD07]. Muchas otras técnicas han sido desarrolladas (como en [AR09] y [GJB96]) aunque estas nunca explican los requerimientos de actualización [BG10], ni indican cuando es correcto realizar el cambio a la nueva especificación. Para tal fin Ghezzi et al. [GGM12, MGGB13] estudiaron el problema de actualizar un controlador que esta monitoreando un ambiente de sistema reactivo mientras controla actuadores. La pregunta que persiguen contestar los autores es ¿cuándo es seguro reemplazar el controlador actual con uno nuevo, donde se cumple la nueva especificación?

Un problema en común que los trabajos existentes en actualización dinámica poseen, es que dichas técnicas necesitan asumir que el sistema que esta siendo ejecutado va a eventualmente alcanzar un estado seguro donde hacer la actualización. Estos estados son designados como *actualizables* y su identificación depende de cada técnica. Dichas técnicas, suelen tener un operador o una pieza de software especial designado a identificar dichos estados. Esta asunción, es algo que no depende del software, sino que depende del ambiente, el cual sabemos que no puede ser manipulado. A su vez, los trabajos existentes, tampoco

dan una técnica para guiar al sistema hacia un estado actualizable.

Por ejemplo, en [KM90], la expectativa sería que los componentes en un sistema distribuido deben ser diseñados, para que den información del momento en que dicho componente entró en estado de *quiescence*. A su vez, están diseñados para aceptar mensajes *pasivate* que, al deshabilitar componentes para inicializar nuevas transacciones, intentan, pero no garantizan que alcance *quiescence*. Similarmente, en [GGM12] requiere asumir que el controlador a ser actualizado va a volver, eventualmente, a su estado inicial (o asumir que la nueva especificación vale desde el último estado inicial). Estados actualizables son aquellos en los que el comportamiento del sistema desde el último estado inicial puede controlarse para satisfacer la nueva especificación. Por otra parte, en [MGGB13], el mismo autor relaja las condiciones necesarias para realizar una actualización, permitiendo más estados actualizables, al costo de violar la nueva especificación, y sin contemplar el problema de garantizar que dicho estado sea alcanzado.

## 1.2. Resumen de la contribución

En esta tesis, como en [GGM12, MGGB13] consideramos el problema de controladores actualizables en un sistema reactivo. Por otro lado, proponemos una técnica para actualización dinámica que *fuera* al sistema a alcanzar un estado actualizable en vez de asumir que el sistema va a llegar a dicho estado. Por lo tanto, nuestro enfoque no sólo garantiza que la nueva especificación va a valer *si* la actualización se produce, sino que también garantiza que la actualización *va a suceder*. Además, generalizamos la noción de estados actualizables en [GGM12, MGGB13] mientras mantenemos correctitud, que es mejor que requerir que la nueva especificación empiece a valer desde el estado inicial (o co-inicial [MGGB13]) del controlador actual. Proveemos también una especificación declarativa y general para señalar desde que punto los nuevos objetivos valen y designar propiedades que deben valer en el momento que el sistema está transicionando.

Una clase de sistemas que se amoldan perfectamente al trabajo presentado son los sistemas adaptables. Dichos sistemas están diseñados para que, mientras el proceso está corriendo, pueda soportar cambios en cuanto a la disponibilidad de recursos o necesidades del usuario y también soportar condiciones inesperadas del ambiente y fallas [SEA14].

Nuestro enfoque de actualización de controladores dinámicos forma parte junto con otros trabajos de la misma área para síntesis de controladores discretos (ej. [RW89], [PPS06],[DBPU13]). Síntesis de controladores automáticamente construye una estrategia operacional (en la forma de máquinas de estado) que es capaz de garantizar un objetivo bajo asunciones del ambiente.

El problema de controladores dinámicos actualizables puede ser expresado como un problema de síntesis de controladores en el cual el nuevo controlador cumple que:

- ★ Es una estructura equivalente al controlador actual hasta que recibe un mensaje (no controlable) llamado *beginUpdate*,
- ★ Satisface la especificación actual hasta que el evento (controlable) *stopOldSpec* sucede,
- ★ Garantiza a la nueva especificación desde que ocurre la acción (controlada) *startNewSpec*,
- ★ Proporciona que el comportamiento entre *beginUpdate*, *stopOldSpec* y *startNewSpec* satisface cualquier requerimiento de transición y
- ★ Garantiza que los eventos *startNewSpec* y *stopOldSpec* van a suceder eventualmente.

### 1.3. Esquema de tesis

Este trabajo esta organizado de la siguiente manera. Empezaremos dando una base de conocimientos teóricos en el capítulo 2. Luego de esto, en el capítulo 3 vamos a explicar el soporte que nos da la herramienta MTSA para poder manejar y controlar todos los conceptos que fueron definidos en el capítulo previo, como el modelado y la síntesis de controladores.

La técnica central de este trabajo y el aporte científico estará principalmente desarrollado en el capítulo 4. En este detallaremos la aplicación de la técnica de síntesis de controladores al problema de actualización dinámica. Explicaremos además, como construimos la solución propuesta y daremos las definiciones necesarias para finalmente demostrar la correctitud de nuestro enfoque.

En el capítulo 5, realizamos una trabajo comparativo entre las técnicas que existen actualmente y la propuesta por nosotros. Los resultados obtenidos destacando ventajas y

desventajas de cada estrategia son expuestos. Explicaremos además cual es la motivación de utilizar cada caso de estudio.

Finalmente, detallamos los límites y alcances de nuestro trabajo en el capítulo 6. Discutiremos otros enfoques comparando nuestra labor con trabajos ya existentes. Resaltaremos posibles trabajos futuros que pueden surgir a partir de este trabajo y concluiremos con un resumen.

## 2. FUNDAMENTOS TEÓRICOS

### 2.1. El Mundo y la Máquina

Los primeros conceptos que detallaremos estarán involucrados con la ingeniería de los requerimientos. Los puntos de vista mas relevantes son los de Zave y Jackson ([ZJ97, Jac95a, Jac95b]) por un lado, y los de Letier y Van Lamsweerde ([VLL00, VL01]) por el otro. Ambos puntos de vista distinguen a los problemas del *Mundo* y las soluciones de la *Máquina* como fundamentales para reconocer si las operaciones de la máquina solucionan los problemas planteados en el mundo. De hecho, el efecto de la máquina en el mundo y las suposiciones que hacemos acerca de este mundo son fundamentales para el proceso de toma de requerimientos. En el lado del mundo definimos una serie de problemas que existen en el mundo real que serán solucionados al construir una máquina. Fácilmente podremos notar que existen componentes en la máquina que interactúan directamente con el mundo siguiendo normas y procesos conocidos. Estas, forman parte de la intersección entre el mundo y la máquina. Por ejemplo un taladro, un brazo robótico o las reglas de procesamiento para cada elemento que entra en una linea de producción (ver la Figura 2.1).

Así mismo, es esperado que la máquina proponga una solución al problema. Por ejemplo, en la Figura 2.1 podemos ver que la célula de producción debe procesar cada producto, sólo si están disponible en la bandeja de entrada en ese momento. Con la sentencia  $EnBandeja[p] \rightarrow get.Bandeja[p]$  mostramos que se espera que el brazo del robot sólo podrá tomar los productos de la bandeja cuando estén listos. Finalizando, los fenómenos compartidos entre el mundo y máquina, es decir, los que se encuentran en la intersección, representa a la *interfaz*, donde la máquina interactúa con el mundo. También, podemos definir a los fenómenos del mundo como el *modelo del entorno* ya que el conjunto de estos describen los eventos que suceden en el mundo real.

Las sentencias que detallan los distintos fenómenos, tanto en el mundo como en la máquina pueden variar en *alcance* y en *forma* [PM95, Jac95a]. Además, estas pueden

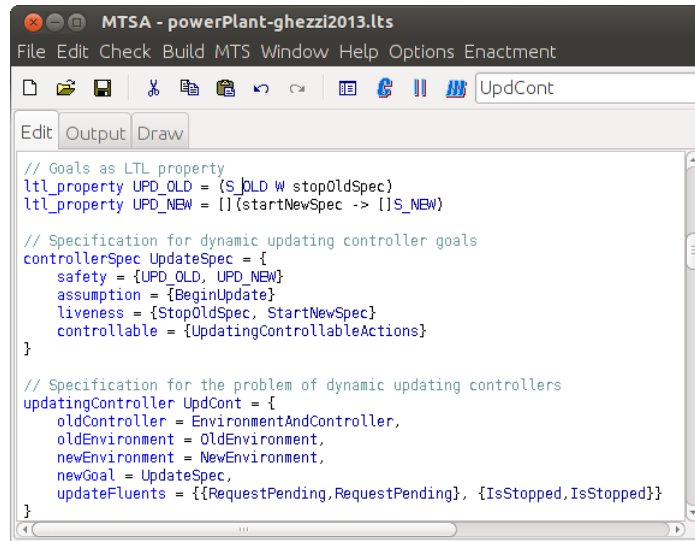


Figura 2.1: El Mundo y la Máquina

estar en modo *indicativo* u *optativo*. En otros trabajos, como en [vL09], las sentencias utilizadas son *descriptivas* y *prescriptivas*.

- ★ Sentencias descriptivas: representan propiedades que son independientes de cómo se comporta el sistema. Se usan en modo *indicativo*. No pueden ser cambiadas ni removidas.
- ★ Sentencia prescriptivas: afirman propiedades deseables que pueden estar presentes o no. Deben estar aplicadas por los componentes del sistema. Normalmente, pueden cambiar fortaleciéndose o debilitándose, o incluso pueden ser eliminadas.

Anteriormente, fue mencionado que los estados pueden variar en su alcance. Ambos tipos de sentencias pueden referirse a características de la máquina que no son compartidas por el mundo. En otras ocasiones, sentencias pueden referirse a fenómenos compartidos por el mundo y la máquina. Más precisamente, una *propiedad de dominio* es una sentencia descriptiva sobre el mundo. Durante todo este trabajo, vamos a llamar *modelo ambiente*, al conjunto de propiedades del dominio de un problema particular.

Por otro lado, un *supuesto de ambiente* es una sentencia que podría no suceder y debe ser satisfecha por el ambiente. Un requisito de software, o *requisito* de forma abreviada, es una sentencia prescriptiva que la máquina deberá satisfacer independientemente de cómo se comporta el problema detallado en el mundo y deben ser elaboradas en términos de



fenómenos compartidos entre el mundo y la máquina.

Para finalizar y siguiendo lo publicado en [VL01, VLL00] podremos determinar a una acción como supervisada/controlable si dicha acción es supervisada/controlable por la máquina. En este trabajo, llamaremos a las acciones supervisadas como acciones no controlables, ya que están controladas por el ambiente.

## 2.2. Sistema de Transición Etiquetados (Labelled Transition System)

En esta sección vamos a dar una notación para los sistemas de transiciones etiquetados o Labelled Transition System (LTS), la cual usaremos durante este trabajo. Dichos sistemas, son muy usados actualmente para modelar y analizar comportamiento en sistemas concurrentes y distribuidos. Un LTS es un sistema de transiciones de estados donde cada uno de ellos esta etiquetado con una acción. El conjunto de todas las acciones que posee un LTS es llamado alfabeto.

**Definición 2.2.1.** (*Sistema de Transición Etiquetado*) [Kel76] Sea *States* un conjunto universal de estados, *Act* un conjunto universal de etiquetas. Un Sistema de Transición Etiquetado (LTS) es una tupla  $E = (S_E, A_E, \Delta_E, s_{E_0})$ , donde  $S_E \subseteq \text{States}$  es un conjunto finito de estados,  $A_E \subseteq \text{Act}$  es un alfabeto finito,  $\Delta_E \subseteq (S_E \times A_E \times S_E)$  es una relación, y  $s_0 \in S_E$  es el estado inicial.

Si  $(s, \ell, s') \in \Delta_E$  diremos que  $\ell$  está activo desde  $s$  en  $E$ . Diremos también que un LTS  $E$  es *determinístico* si  $\forall_{(s, \ell, s'), (s, \ell, s'') \in \Delta_E}$  implica  $s' = s''$ . Para un estado  $s$  definiremos  $\Delta_E(s) = \{\ell \mid (s, \ell, s') \in \Delta_E\}$ . Dado un LTS  $E$ , vamos a referirnos a su alfabeto como  $\alpha E$ .

**Definición 2.2.2.** (*Composición en Paralelo*) Sean  $M = (S_M, A_M, \Delta_M, s_{M_0})$  y  $E = (S_E, A_E, \Delta_E, s_{E_0})$  LTSs. Una Composición en Paralelo ( $\parallel$ ) es un operador simétrico tal que  $E \parallel M$  es el LTS definido de la siguiente manera  $E \parallel M = (S_E \times S_M, A_E \cup A_M, \Delta, (s_{E_0}, s_{M_0}))$ , donde  $\Delta$  es la relación mas pequeña que satisface las siguientes reglas, donde  $\ell \in A_E \cup A_M$ :

$$\frac{(s,\ell,s') \in \Delta_E}{((s,t),\ell,(s',t)) \in \Delta} \ell \in A_E \setminus A_M \qquad \frac{(t,\ell,t') \in \Delta_M}{((s,t),\ell,(s',t')) \in \Delta} \ell \in A_M \setminus A_E$$

$$\frac{(s,\ell,s') \in \Delta_E, (t,\ell,t') \in \Delta_M}{((s,t),\ell,(s',t')) \in \Delta} \ell \in A_E \cap A_M$$

**Definición 2.2.3.** (*LTS Legal*) Dado  $E = (S_E, A_E, \Delta_E, s_{E_0})$ ,  $M = (S_M, A_M, \Delta_M, s_{M_0})$  LTSs, y  $A_{E_u} \in A_E$ . Decimos que  $M$  es un LTS Legal para  $E$  con respecto a  $A_{E_u}$  si para todos  $(s_E, s_M) \in E \parallel M$  sucede lo siguiente:  $\Delta_{E \parallel M}((s_E, s_M)) \cap A_{E_u} = \Delta_E(S_E) \cap A_{E_u}$ .

Intuitivamente, un LTS  $M$  es un LTS Legal para el LTS  $E$  con respecto a  $A_{E_u}$ , si para todos los estados en la composición  $(s_E, s_M) \in S_{E \parallel M}$  se cumple que, una acción  $\ell \in A_{E_u}$  es deshabilitada en  $(s_E, s_M)$  si y sólo si también esta deshabilitada en  $s_E \in E$ . En otras palabras,  $M$  no restringe a  $E$  con respecto a  $A_{E_u}$ .

**Definición 2.2.4.** (*Tranza*) Sea un LTS  $E = (S, A, \Delta, s_0)$ . Una secuencia  $\pi = \ell_0, \ell_1, \dots$  es una traza en  $E$  si existe una secuencia  $s_0, \ell_0, s_1, \ell_1, \dots$  donde para todo  $i$  tenemos  $(s_i, \ell_i, s_{i+1}) \in \Delta$ .

**Definición 2.2.5.** (*Estados Alcanzables*) Sea un LTS  $E = (S_E, A_E, \Delta_E, s_0)$ . Un estado  $s \in S_E$  es alcanzable (desde el estado inicial) en  $E$  si existe una secuencia  $s_0, \ell_0, s_1, \ell_1, \dots$  donde para cada  $i$  tenemos  $(s_i, \ell_i, s_{i+1}) \in \Delta$  y  $s = s_{i+1}$ . Nos referimos a el conjunto de todos los estados alcanzables en  $E$  como  $\text{Reach}(E)$ .

En el transcurso de esta tesis, vamos a estudiar sólo aquellos LTSs  $E$  donde todos sus estados  $s \in S_E$  son alcanzables.

### 2.3. Lógica Lineal Temporal de Flujos (Fluent Linear Temporal Logic)

La Lógica Lineal Temporal (LTL) esta siendo ampliamente usada en la ingeniería de los requerimientos [KPR04, GM03, VLL00, LvL02]. La motivación para escoger a las LTL de flujos es que estas proveen un framework uniforme para especificar propiedades basados en estados sobre modelos basados en eventos [GM03]. Fluent Linear Temporal Logic (FLTL) [GM03] es una lógica de tiempo lineal, temporal, para razonar acerca de flujos. Un *flujo*  $Fl$  es definido por un par de conjuntos y un valor booleano:  $Fl = \langle I_{Fl}, T_{Fl}, \text{Init}_{Fl} \rangle$ , donde

$I_{Fl} \subseteq Act$  es el conjunto de acciones iniciadoras,  $T_{Fl} \subseteq Act$  es el conjunto de acciones finalizadoras y  $I_{Fl} \cap T_{Fl} = \emptyset$ . Un flujo puede ser inicializado con valor *true* o *false* indicado por  $Init_{Fl}$ . Toda acción  $\ell \in Act$  induce un flujo, que notaremos  $\ell = \langle \ell, Act \setminus \{\ell\}, false \rangle$ . Por último, el alfabeto de un flujo es el que se obtiene mediante la unión del conjunto de acciones iniciadoras y el conjunto de acciones finalizadoras.

Sea  $\mathcal{F}$  el conjunto de todos los posibles flujos sobre  $Act$ . Una formula FLTL se define inductivamente utilizando los conectores booleanos estandar y operadores temporales como el **X** (próximo), **U** (antes fuerte) de la siguiente manera:

$$\varphi ::= Fl \mid \neg\varphi \mid \varphi \vee \psi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \psi \quad (2.1)$$

donde  $Fl \in \mathcal{F}$ . Para comodidad sintáctica, vamos a introducir las operaciones de  $\wedge$ ,  $\diamond$  (eventualmente) y  $\square$  (siempre). Sea  $\Pi$  el conjunto de trazas infinitas sobre  $Act$ , diremos que la traza  $\pi = \ell_0, \ell_1, \dots$  satisface un flujo  $Fl$  en la posición  $i$ , notado  $\pi, i \models Fl$ , si y sólo si una de las siguientes condiciones es válida:

$$\star \text{ } Init_{Fl} \wedge (\forall j \in \mathbb{N} : 0 \leq j \leq i \rightarrow \ell_j \notin T_{Fl})$$

$$\star \text{ } \exists j \in \mathbb{N} : (j \leq i \wedge \ell_j \in I_{Fl}) \wedge (\forall k \in \mathbb{N} : j < k \leq i \rightarrow \ell_k \notin T_{Fl})$$

Dada una traza infinita  $\pi$ , la fórmula que satisface  $\varphi$  en la posición  $i$ , denotada como  $\pi, i \models \varphi$ , es definida a continuación como se muestra en la semántica para el operador de satisfacción:

$$\begin{aligned} \pi, i \models Fl & \triangleq \pi, i \models Fl \\ \pi, i \models \neg\varphi & \triangleq \neg(\pi, i \models \varphi) \\ \pi, i \models \varphi \vee \psi & \triangleq (\pi, i \models \varphi) \vee (\pi, i \models \psi) \\ \pi, i \models \mathbf{X}\varphi & \triangleq \pi, 1 \models \varphi \\ \pi, i \models \varphi \mathbf{U} \psi & \triangleq \exists j \geq i : \pi, j \models \psi \wedge \forall i \leq k < j : \pi, k \models \varphi \end{aligned}$$

Diremos que  $\varphi$  se cumple en  $\pi$ , denotado como  $\pi \models \varphi$ , si  $\pi, 0 \models \varphi$ . Una fórmula  $\varphi \in \text{FLTL}$  es cierta si un LTS  $E$  (denotado como  $E \models \varphi$ ) si éste es cierto en toda traza infinita producida por  $E$ .

## 2.4. Problemas de síntesis de controladores

Los problemas de síntesis de controladores son aquellos que producen una máquina, la cual, restringe las ocurrencias de los eventos controlables, basado en las observaciones, de los eventos no controlables que han ocurrido. Dicha máquina, al ser desplegada con un ambiente adecuado logramos satisfacer el conjunto de objetivos del sistema. Cabe destacar, que estos objetivos se cumplirán si se satisfacen las asunciones que se hacen sobre el ambiente. Resumiendo, tendremos una especificación del ambiente, asunciones, objetivos, y un conjunto de acciones controlables. Resolver el *problema de síntesis de control* es hallar una máquina, que al trabajar concurrentemente con el ambiente, que satisface las asunciones del dominio, satisfacemos el conjunto de objetivos del sistema.

Hecha esta introducción definiremos el problema de síntesis de control para modelos basados en eventos de la siguiente manera. Dada una LTS que detalla el comportamiento del ambiente, un conjunto de eventos controlables, un conjunto de formulas FLTL que describen los objetivos del sistema, el problema de control LTS consiste en encontrar un LTS que restringe solamente la ocurrencia de acciones controlables y garantiza que la composición paralela del ambiente con el LTS recién descrito estará libre de deadlocks y que, si las presunciones del ambiente valen, satisfacerá también los objetivos del sistema.

**Definición 2.4.1.** (*Control LTS*) Dada una especificación de un entorno en forma de una LTS  $E$ , un conjunto de acciones controlables  $A_c \in Act$  y un conjunto  $H$  de pares  $(A_{s_i}, G_i)$  donde  $A_{s_i}$  y  $G_i$  son fórmulas FLTL especificando presunciones y objetivos del sistema respectivamente, la solución al problema de control LTS  $\mathcal{E} = \langle E, H, A_c \rangle$  consiste en encontrar una LTS  $M$  de forma que  $M$  es legal a  $E$  sobre el conjunto de acciones no controlables  $A_U = \overline{A_c}$ ,  $E || M$  se encuentra libre de deadlocks, y para cada par  $(A_{s_i}, G_i) \in H$  y para cada traza  $\pi$  en  $E || M$  se cumple que si  $\pi \models A_{s_i}$  entonces  $\pi \models G_i$ .

Ahora pasaremos a definir un subconjunto de problemas de control LTS que esta determinado por aquellos problemas de control que son computables en tiempo polinómico. Identificaremos estos problemas como problemas de control LTS SGR(1) (Safe Generalised Reactivity(1)). Estos se construyen a partir de GR(1) y problemas de seguridad pero en modelos basados en eventos. Dichos problemas, constan de un modelo del ambiente  $E$  que

será un LTS determinístico para asegurar que el controlador tenga una visión completa de los estados del ambiente. Requerimos que  $H$  sea  $\{(\emptyset, I), (A_s, G)\}$ , donde  $I$  es un invariante de seguridad de la pinta  $\Box \rho$ , las asunciones  $A_s$  son una conjunción de sub-fórmulas FLTL de la pinta  $\Box \Diamond \phi$ , y el objetivo  $G$  una conjunción de sub-fórmulas FLTL de la pinta  $\Box \Diamond \gamma$  donde  $\rho, \phi$  y  $\gamma$  son combinación booleana de flujos.

**Definición 2.4.2.** (*Control LTS SGR(1)*) un problema de control LTS  $\mathcal{E} = \langle E, H, A_C \rangle$  es SGR(1) si  $E$  es determinístico, y  $H = \{(\emptyset, I), (A_s, G)\}$ , donde  $I = \Box \rho$ ,  $A_s = \bigwedge_{i=1}^n \Box \Diamond \phi_i$ ,  $G = \bigwedge_{j=1}^m \Box \Diamond \gamma_j$ , y  $\phi_i, \rho$  y  $\gamma_j$  son combinación booleana de flujos.

## 2.5. Juegos de dos jugadores

Llamaremos juegos de dos jugadores a aquellos que consisten en dos jugadores, jugador 1 y jugador 2, donde el objetivo del jugador 1 es satisfacer una especificación independientemente de las acciones que el jugador 2 ejecute. Intuitivamente, el jugador 1 puede deshabilitar las acciones que él controla aunque no podrá deshabilitarlas todas ya que esto transformaría dicho estado a un estado de deadlock.

Durante el transcurso de esta tesis llevaremos los juegos de dos jugadores al marco de síntesis de controladores, donde el jugador 1 (el controlador) elige, del conjunto de acciones controlables, cual habilitar y el jugador 2 (el ambiente) elige que acciones tomar libremente. Formalmente podemos definir un juego de dos jugadores de la siguiente manera:

**Definición 2.5.1.** (*Juego de dos jugadores*) Un juego de dos jugadores es  $G = (S, \Gamma^-, \Gamma^+, s_{g_0}, \varphi)$ , donde  $S$  es un conjunto finito de estados,  $\Gamma^-, \Gamma^+ \subseteq S \times S$  son conjuntos de transiciones no controlables y controlables respectivamente,  $s_{g_0} \in S$  es el estado inicial, y  $\varphi \subseteq S^\omega$  es la condición de ganada. Definimos  $\Gamma^-(s) = \{s' \mid (s, s') \in \Gamma^-\}$  y análogamente para  $\Gamma^+$ . Un estado  $s$  es no controlable si  $\Gamma^-(s) \neq \emptyset$  y controlable en el resto de los casos. Una jugada en  $G$  es una secuencia  $p = s_{g_0}, s_{g_1}, \dots$ . Una jugada  $p$  terminada en  $s_{g_n}$  es extendida por el controlador eligiendo un subconjunto  $\gamma \subseteq \Gamma^+(s_{g_n})$ . Luego el ambiente elige un estado  $s_{g_{n+1}} \in \gamma \cup \Gamma^-(s_{g_n})$  y agrega  $s_{g_{n+1}}$  a  $p$ .

Un detalle importante es que si para un estado controlable  $\gamma$  el conjunto de opciones del controlador es vacía, esto puede llevar a un deadlock. Esto será considerado como

prohibido más adelante, ya que el controlador definirá este estado como un estado perdedor. Para un estado no controlable el controlador puede decidir deshabilitar todas las acciones controlables. Las elecciones del controlador son formalizadas como estrategias y estas reglas son las que el controlador aplicará. Por lo general, las estrategias son elegidas dependiendo de la historia. Esto puede verse en la estrategia utilizando un valor de memoria  $\Omega$  y actualizando este valor de acuerdo a la evolución del juego.

Es importante destacar, que este tipo de juegos, con memoria, es diferente al definido en [PPS06]. Piterman et al. definen un juego en el cual el ambiente elige su movimiento y recién luego de este, el controlador podrá elegir cual será el siguiente paso.

**Definición 2.5.2.** (*Estrategia con memoria*) Una estrategia con memoria  $\Omega$  para el controlador es un par de funciones  $(\sigma, u)$ , donde  $\Omega$  es una memoria que tiene designado como valor inicial  $\omega_0$ ,  $\sigma : \Omega \times S \rightarrow 2^S$  tal que  $\sigma(\omega, s) \subseteq \Gamma^+(s)$  y  $u : \Omega \times S \rightarrow \Omega$ .

Intuitivamente,  $\sigma$  le informa al controlador cuales estados debe habilitar como posibles sucesores y  $u$  define como actualizar la memoria en cada paso. Si  $\Omega$  es finita, diremos que la estrategia usa memoria finita.

**Definición 2.5.3.** (*Consistencia y estrategia ganadora*) una jugada finita o infinita  $p = s_0, s_1, \dots$  es consistente con  $(\omega, u)$  si para cada  $n$  tenemos que  $s_{n+1} \in \sigma(\omega_n, s_n)$  donde  $\omega_{i+1} = u(\omega_i, s_{i+1})$  para toda  $i \geq 0$ . Una estrategia  $(\sigma, u)$  para el controlador desde el estado  $s$  es ganadora si cada jugada maximal empezando de  $s$  y consistente con  $(\sigma, u)$  es infinita y en  $\varphi$ . Diremos que el controlador gana el juego  $G$  si tiene una estrategia ganadora desde el estado inicial.

Diremos que verificar si un controlador gana un juego  $G$  es resolver el juego  $G$ . Una vez definido un juego de dos jugadores, pasaremos a traducir un problema de síntesis de controladores a este tipo de juegos. La transformación se basa en generar una estrategia ganadora para el controlador. Si dicha estrategia existe, diremos que el problema de control es realizable [MPS95, RW89]. Resultados estudiados anteriormente [PR89], demuestran que si un controlador gana el juego  $G$  y  $\varphi$  es  $\omega$ -regular, el juego puede ganarse utilizando una estrategia con memoria finita.

## 2.6. Resolviendo el problema de control LTS SGR(1)

En esta sección explicaremos como una solución para un problema de control SGR(1) puede ser obtenida por construcción utilizando técnicas existentes de síntesis de controladores (basados en estados), llamados GR(1). [PPS06]

La construcción de la máquina para un problema de control LTS SGR(1) está dividido en dos pasos. Primero, se crea un juego GR(1)  $G$  en representación del ambiente  $E$ , las asunciones  $A_s$ , los objetivos  $O$  y el conjunto de acciones controlables  $A_C$ . Como segundo paso, se elabora una solución  $(\sigma, u)$  al juego GR(1) para construir una máquina  $M$  (i.e un controlador LTS) para  $\mathcal{E}$ . Esta solución al problema de control LTS SGR(1)  $\mathcal{E}$  existe, si y sólo si, existe una solución al juego GR(1)  $G$ . Luego, podremos afirmar que el controlador LTS  $M$  creada a partir de  $(\sigma, u)$  es una solución a  $\mathcal{E}$ .

### 2.6.1. Control LTS SGR(1) a juegos GR(1)

Convertiremos el problema de control LTS SGR(1) a un juego GR(1). Dado un problema de control LTS SGR(1)  $\mathcal{E} = \langle E, H, A_C \rangle$  construimos un juego GR(1)  $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$  tal que cada estado en  $S_g$  representa un estado en  $E$  y una valuación de todos los flujos que aparecen en  $A_s$  y en  $G$ .

Más precisamente, y por la definición de control LTS SGR(1) (definición 2.4.2) tendremos que  $H = \{(\emptyset, I), (A_s, G)\}$ ,  $E = (S_e, A, \Delta_e, s_{e_0})$ ,  $A_s = \bigwedge_{i=1}^n \square \diamond \phi_i$ ,  $I = \square \rho$  y  $G = \bigwedge_{j=1}^m \square \diamond \omega_j$ . Sea  $fl = \{\dot{1}, \dots, \dot{k}\}$  un conjunto de flujos usados en  $A_s$  y en  $G$  donde  $\dot{i} = \langle I_i, T_i, Init_i \rangle$ . Construimos al juego  $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$  de la siguiente manera:

Construimos  $S_g$  a partir de  $E$  de tal forma, que los estados en  $S_g$  corresponden a un estado en  $E$  y los valores de verdad de los flujos en  $\varphi$ . Formalmente, tenemos que  $S_g = S_e \times \prod_{i=1}^k \{true, false\}$ . Consideramos un estado  $s_g = (s_e, \alpha_1, \dots, \alpha_k)$ . Dado un flujo  $fl_i$ , diremos que  $s_g$  satisface  $fl_i$  si  $\alpha_i$  es *true* y  $s_g$  no satisface  $fl_i$  si no.

Además, definiremos las relaciones  $\Gamma^-$  y  $\Gamma^+$  aplicando las siguientes reglas. Sea  $s_g = (s_e, \alpha_1, \dots, \alpha_k)$ . Si  $s_g$  no satisface  $\rho$  (es decir,  $s_g$  es no seguro) no agregaremos los sucesores a  $s_g$ . Si  $s_g$  satisface  $\rho$ , por cada transición  $(s_e, l, s'_e) \in \Delta_e$  agregaremos  $(s_g, (s'_e, \alpha'_1, \dots, \alpha'_k))$

en  $\Gamma^\beta$ , donde  $\beta$  y  $\alpha'_i$  cumplen las siguiente condiciones:

$\beta$	$\alpha'_i$
es +: si $l \in A_C$ ,	es $\alpha_i$ : si $l \notin I_{fl_i} \cup T_{fl_i}$ ,
es -: si $l \notin A_C$ .	es <i>true</i> : si $l \in I_{fl_i}$ o
	es <i>false</i> : si $l \in T_{fl_i}$ .

El estado inicial  $s_{g_0}$  es  $(s_{e_0}, \text{initially}_1, \dots, \text{initially}_k)$ .

Por último, construiremos la *condición de ganada*  $\varphi_g$ , definida como un conjunto infinito de trazas, para  $A_S$  y  $G$  de la siguiente manera: abusando de la notación denotaremos  $\phi_i$  al conjunto de estados  $s_g$  tales que  $s_g$  satisface las asunciones  $\phi_i$  y a  $\gamma_i$  al conjunto de secuencias que satisfacen  $gr((\phi_1, \dots, \phi_n), (\gamma_1, \dots, \gamma_m))$ . De esta forma, obtendremos que  $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$  es un juego GR(1).

Cabe destacar que las propiedades de seguridad (*safety*) que son parte de la especificación no están contempladas en la *condición de ganada*  $\varphi_g$  del juego GR(1), pero si se traducen a un problema de *deadlock avoidance* a la hora de construir  $\Gamma^-$  y  $\Gamma^+$ . De esta manera, la *condicion de ganada* es  $\Box \rho \wedge (\bigwedge_{i=1}^n \Box \Diamond \phi_i \Rightarrow \bigwedge_{j=1}^m \Box \Diamond \omega_j)$ .

### 2.6.2. Traduciendo la estrategia a un Controlador LTS

Ahora pasaremos a explicar como conseguir un controlador LTS a partir de una estrategia ganadora para el juego en GR(1). Intuitivamente, la transformación es de la siguiente manera: dado un problema de control LTS SGR(1)  $\mathcal{E} = \langle E, H, A_C \rangle$ , el juego  $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$  obtenido a partir de  $\mathcal{E}$  y de la estrategia ganadora para  $G$ , construiremos  $M = (S_M, A, \Delta_M, s_{M_0})$  una solución para  $\mathcal{E}$  traduciendo a estados de  $S_M$  un estado de  $S_g$  y un estado de la memoria dada por la estrategia ganadora.

Más formalmente, sea  $E = (S_e, A, \Delta_e, s_{e_0})$ ,  $fl = \{fl_1, \dots, fl_k\}$  el conjunto de flujos que aparecen en  $\varphi$ ,  $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$  el juego GR(1) construido a partir de  $E$  como explicamos anteriormente, y sea  $\sigma : \Omega \times S_g \rightarrow 2^{S_g}$  y  $u : \Omega \times S_g \rightarrow \Omega$  la estrategia ganadora para  $G$ . Construiremos la máquina  $M = (S_M, A, \Delta_M, s_{M_0})$  de la siguiente manera.

Para construir  $S_M \subseteq \Omega \times S_g$ , consideremos dos estados  $s_g = (s_e, \alpha_1, \dots, \alpha_k)$  y  $s'_g = (s'_e, \alpha'_1, \dots, \alpha'_k)$ . Decimos que esa acción  $\ell$  es *posible* desde  $s_g$  hacia  $s'_g$  si:



1.  $(s_g, s'_g) \in \Gamma^- \cup \Gamma^+$ ,
2. existe una acción  $\ell$  tal que  $(s_e, \ell, s'_e) \in \Delta_e$  y
3. para cada *flujo*  $fl_i$  valga alguna de las siguiente condiciones:
  - ★  $\ell \notin I_{fl_i} \cup T_{fl_i}$  y  $\alpha'_i = \alpha_i$ ,
  - ★  $\ell \in I_{fl_i}$  y  $\alpha'_i = true$ , o
  - ★  $\ell \in T_{fl_i}$  u  $\alpha'_i = false$ .

Para construir  $\Delta_M \subset S_M \times A \times S_M$ , consideremos la transición  $(s_g, s'_g) \in \Gamma^-$ . Por definición de  $\Gamma^-$  existe una acción  $\ell \notin A_C$  tal que  $\ell$  es posible desde  $s_g$  hacia  $s'_g$ . Si  $s'_g \in \sigma(\omega, s_g)$  entonces para cada acción  $\ell$  tal que  $\ell$  es posible desde  $s_g$  hacia  $s'_g$  agregamos  $((\omega, s_g), \ell, (u(\omega, s_g), s'_g))$  hacia  $\Delta_M$ . De forma similar, consideramos una transición  $(s_g, s'_g) \in \Gamma^+$ . Por definición de  $\Gamma^+$  existe una acción  $\ell \in A_C$  tal que  $\ell$  es posible desde  $s_g$  hacia  $s'_g$ . Si  $s'_g \in \sigma(\omega, s_g)$  entonces para cada acción  $\ell$  tal que  $\ell$  es posible desde  $s_g$  hacia  $s'_g$  agregamos  $((\omega, s_g), \ell, (u(\omega, s_g), s'_g))$  hacia  $\Delta_M$ .

El estado inicial de  $M$  esta definido como  $s_{M_0} = (\omega_0, s_{g_0})$  donde  $\omega_0$  es el valor inicial de la memoria  $\Omega$ . De esta forma completamos la definición de  $M$ .

### 2.6.3. Algoritmo

En esta sección, presentaremos el algoritmo implementado en la herramienta MTSA [DFCU08] el cual está basado en las ideas de Juvekar y Piterman [JP06a].

Este algoritmo realiza una búsqueda de ciclos de estados que satisfacen todas las asunciones pero no todos los objetivos restringiendo acciones controlables. De haber ciclos como estos podrían permitir trazas en las que el controlador pierde el juego GR(1). Para lograr evitar estos ciclos, el algoritmo busca para cada estado, una estrategia que garantice la satisfacción de todos los objetivos. Para esto, se configura un orden en el cual satisfacer los objetivos. El algoritmo, mediante la técnica de punto fijo computa la mejor forma en que cada estado puede satisfacer el siguiente objetivo. A su vez, mide la “calidad” de cada uno de los diferentes sucesores para satisfacer un objetivo mediante un sistema de rankings [Jur00]. El ranking de un sucesor particular mide la distancia (cantidad de transiciones utilizadas) al siguiente objetivo en términos de número de veces que las asunciones son satisfechas antes de alcanzar el objetivo. Si este número tiende a infinito, deduciremos que desde el estado actual existe una traza infinita en la cual las asunciones del ambiente

valen infinitamente, pero los objetivos no se satisfacen. Es así, como el algoritmo reconoce estados que deben ser evitados para la construcción de la estrategia para el controlador.

**Definición 2.6.1.** (*Función de Ranking*) Sea  $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi)$  donde  $\varphi = gr((\phi_1, \dots, \phi_n), (\gamma_1, \dots, \gamma_m))$ . Una función de ranking para un objetivo  $\gamma_j$  es una función  $R_j : S_g \rightarrow (\mathbb{N} \times \{1, \dots, n\} \cup \{\infty\})$ . Intuitivamente,  $R_j(s_g) = (k, \ell)$  significa que para alcanzar desde  $s_g$  a un estado en el cual  $\gamma_j$  vale, todos los caminos satisfacen la asunción  $\phi_\ell$  a lo sumo  $k$  veces.  $R(s) = \infty$  significa que  $s$  es un estado perdedor, es decir, desde  $s$  no hay una estrategia para el controlador que pueda evitar una traza en la cual se satisfice infinitamente las asunciones, pero no satisface infinitamente a todos los objetivos.

El algoritmo 1 computa un ranking estable en cada estado  $s_g \in T$  si  $s_g$  es ganador para el controlador (es decir,  $R_1(t) < \infty$ ). Conceptualmente, podemos separar el algoritmo en dos grandes instancias, inicialización y estabilización. El valor inicial del ranking para cada estado en el juego, junto a la cola de estados *pending* para ser procesados, se crean en la etapa de inicialización. Agregaremos un estado a *pending* si no satisface ningún objetivo y satisface las asunciones. Todos los estados en cada función de ranking son inicializados con el valor  $(0, 1)$ . Este valor indica el menor ranking posible. Los estados que cumplen que  $\Gamma^- \cup \Gamma^+ = \emptyset$  serán inicializados con el valor  $\infty$ . De esta manera, los estados cuyos rankings son  $\infty$  son aquellos donde no se satisface  $\rho$  o son estados de *deadlock* en  $E$ .

La sección de estabilización es un iteración de punto fijo sobre la cola *pending* hasta que se vacía. La función `is_stable(state, g)` devuelve true si la  $g$ -ésima función de ranking es estable para `state`.

La función `unstable_pred(state, g)` devuelve un conjunto de pares de predecesores de `state` y un ranking  $g$  para el cual el ranking es inestable.

La función `best(state, g)` devuelve el mejor ranking basado en sus sucesores. Para eso utiliza la siguiente función  $sr : S_g \rightarrow (\mathbb{N} \times \{1, \dots, n\} \cup \{\infty\})$ . Esta función también codifica el hecho de que los estados de *deadlock* tienen ranking  $\infty$ . Además, notemos que usa un orden lexicográfico para los objetivos. Dado un estado  $s_g$  y un objetivo  $\gamma_j$ ,  $sr(s_g, j)$  está definida de la siguiente manera:

- \* Si  $\Gamma^+(s_g) \cup \Gamma^-(s_g) = \emptyset$ , entonces  $sr(s_g, j) = \infty$ , caso contrario,
- \* si  $s_g$  es controlable y  $s_g \in \gamma_j$ , entonces  $sr(s_g, j) = \min_{s'_g \in \Gamma^+(s_g)} R_{j \oplus 1}(S'_g)$ .

---

**Algorithm 1** para resolver juegos SGR(1)

---

```

1: procedure SOLVEGAME( $\text{GAME}=(\text{STATES},\text{TRANSITIONS}),\text{SAFE},\text{GUARANTEES},\text{ASSUMPTIONS}$ )
2:   // Inicialización:
3:   for state : states do
4:     for g : guarantees do
5:        $\text{rank}_g(\text{state}) \leftarrow (0,1)$ 
6:   // Encolar pendientes
7:   for state : states do
8:     if  $\exists g : \text{guarantees} / \text{state} \notin g \wedge \text{state} \in \text{assume}_1$  then
9:       pending.push(pair(state,g))
10:    if  $\Gamma^-(\text{state}) = \emptyset \wedge \Gamma^+(\text{state}) = \emptyset$  then
11:      for g : guarantees do
12:         $\text{rank}_g(\text{state}) \leftarrow \infty$ 
13:        pending.push(unstable_pred(state,g))
14:   // Estabilización:
15:   while !pending.empty() do
16:     (state,g)  $\leftarrow$  pending.pop()
17:     if  $\text{rank}_g(\text{state}) = \infty$  then
18:       continue
19:     if isStable( $\text{rank}_g(\text{state})$ ) then
20:       continue
21:      $\text{rank}_g(\text{state}) \leftarrow \text{inc}(\text{best}(\text{state},g),\text{state},g)$ 
22:     pending.push(unstable_pred(state,g))

```

---

- ★ si  $s_g$  es controlable y  $s_g \notin \gamma_j$ , entonces  $sr(s_g, j) = \min_{s'_g \in \Gamma^+(s_g)} R_j(s'_g)$ .
- ★ si  $s_g$  es no controlable y  $s_g \in \gamma_j$ , entonces  $sr(s_g, j) = \max_{s'_g \in \Gamma^-(s_g)} R_{j \oplus 1}(s'_g)$ .
- ★ si  $s_g$  es no controlable y  $s_g \notin \gamma_j$ , entonces  $sr(s_g, j) = \max_{s'_g \in \Gamma^-(s_g)} R_j(s'_g)$ .

Por último,  $\text{inc}((k, \ell), \text{state}, g)$  devuelve  $(0, 1)$  si  $\text{state}$  está en  $\gamma_g$ , devuelve  $(k, \ell)$  si  $\text{state}$  no está en  $\text{assumption}_\ell$ , y devuelve el mínimo valor mayor que  $(k, \ell)$  en el resto de los casos. Notemos que  $\text{inc}(\infty, \text{state}, g)$  es  $\infty$ , y si  $n = \max_\ell(|\phi_\ell - (\gamma_g)|)$  y  $\text{state}$  está en  $\phi_m - \gamma_g$  entonces  $\text{inc}((n, m), \text{state}, g)$  es  $\infty$ . Este algoritmo calcula el mínimo ranking estable. Basados en ideas del mundo de autómatas de Büchi [EWS05, JP06b], este algoritmo puede ser implementado en  $O(m \cdot n \cdot |S|^2)$ .

## 2.7. Procesos de estados Finitos (Finite State Process)

A esta altura, ya hemos definido las LTSs definiendo sus componentes, como lo son, sus estados, sus acciones, sus transiciones y su estado inicial. Esta representación es adecuada para LTSs con pocos estados, pero se vuelve muy poco práctica a la hora de trabajar con LTSs de gran tamaño. Por esta razón, usamos una simple notación de álgebra de procesos llamada procesos de estados finitos (FSP: Finite State Process) para especificar LTSs. [MKG97, MK99]

El FSP es un lenguaje de especificación de semántica bien definida en términos de LTSs que provee describirlos de manera concisa. Cada expresión FSP  $E$  puede ser relacionada a un LTS finito. Notaremos  $\text{lts}(E)$  al LTS que corresponde a dicho FSP. A continuación discutiremos detalladamente la sintaxis del FSP.

A modo de ejemplo, en la Figura 2.2, mostramos un código FSP que representa el funcionamiento de una planta nuclear.

En FSP, los nombres de los procesos empiezan con letras mayúsculas y las acciones con minúsculas. El código de la planta nuclear consta de dos procesos FPS, el primero, llamado **MAINTENANCE** modela el proceso de enviar un mensaje para que se realice el mantenimiento de la bomba refrigeradora y recibe la respuesta de dicho mensaje. Estas acciones se representan con la acción **request** y **ok** respectivamente. Por otro lado, tenemos el proceso **COOLER** que posee como procesos auxiliares a los subprocesos **STARTED**

```

MAINTENANCE = (request->ok->MAINTENANCE) .

COOLER = STARTED,
STARTED = (stopPump->STOPPED | procedure->STARTED |
           ok->STARTED) ,
STOPPED = (startPump->STARTED | procedure->STOPPED |
           ok->STOPPED) .

||COOLING_TOWER = (MAINTENANCE||COOLER) .

```

Figura 2.2: Ejemplo FSP

y **STOPPED** que son locales al proceso FSP en donde están definidas. **COOLER** está definida para que inicialmente se comporte como **STARTED** puesto que queremos modelar que la bomba en estado inicial esta prendida. Luego, podemos ejecutar diferentes acciones, **stopPump**, **procedure** y **ok**. **STARTED** está definido usando el operador de acción **->** y recursión. Por ejemplo, dicho proceso está definido para empezar ejecutando, o bien **procedure** o **ok**, acciones que nos llevan a seguir ejecutando como el proceso **STARTED** indica, o **stopPump** que nos llevará a ejecutar el proceso **STOPPED**.

A su vez, los FSP soportan distintos operadores de composición como la composición en paralelo. Dicha operación, denotada como **||**, esta definida para preservar la semántica de la composición en paralelo de los LTS definidos en la definición 2.2.2. Por lo tanto, dados dos procesos FSP  $P$  y  $Q$ , tenemos:  $\text{lhs}(P||Q) = \text{lhs}(P)||\text{lhs}(Q)$ .

En procesos FSP que están definidos mediante una composición de dos procesos no auxiliares, son llamados procesos compuestos y sus nombres poseen el prefijo **||**. En nuestro ejemplo, la composición en paralelo entre los procesos FSP **MAINTENANCE** y **COOLER** se escribe como **||COOLING\_TOWER = (MAINTENANCE||COOLER)**.

Además, FSP posee palabras reservadas que se colocan antes de la definición de un proceso que fuerzan a la herramienta MTSA a realizar una operación mas compleja al proceso. Un caso de estos, es la palabra reservada **minimal**, la cual, hace que MTSA construya un LTS minimal que respeta la semántica equivalente o la palabra reservada **deterministic**, que construye un LTS minimal con respecto a las trazas.

FSP también permite definir propiedades FLTL. Un flujo que marca aquellos estados donde la bomba esta apagada puede ser expresada en lenguaje FSP mediante el siguiente código: **fluent IsStopped = <stopPump,startPump> initially 0**. Como di-

jimos anteriormente, la bomba empieza encendida, por lo tanto `IsStopped` es inicialmente falso, pasa a ser verdadero cuando sucede la acción **`stopPump`** y falso nuevamente cuando la acción **`startPump`** sucede.

Finalizando, FSP nos otorga facilidad para especificar LTSs y FLTL formulas. Este lenguaje es el que utilizaremos en los siguiente capítulos para definir modelos que representan ambientes y objetivos.

### 3. MTSA COMO HERRAMIENTA DE MODELADO Y SÍNTESIS

En el marco de este trabajo se hace uso de la herramienta MTSA tanto para el modelado de los procesos del sistema como para la síntesis de controladores que, al ser compuesto con el entorno, permite satisfacer los objetivos. En el siguiente capítulo se describe la extensión de la herramienta para permitir ejecutar el controlador sintetizado en el entorno físico. Utilizaremos sistemas LTS para modelar las distintas entidades.

#### 3.1. Construcción

En MTSA, los modelos son definidos mediante una extensión del lenguaje de Procesos de Estados Finitos (FSP). Dicho lenguaje es un lenguaje textual centrado en la construcción composicional de modelos complejos que originalmente fue usado para describir LTSs.

Los FSP incluyen varios operadores tradicionales para describir modelos de comportamiento, de los cuales destacaremos el prefijo de acción ( $\rightarrow$ ), elección ( $|$ ), composición secuencial ( $;$ ), composición paralela ( $||$ ) y mezcla. La semántica de la mezcla es tal que dada dos descripciones parciales del mismo componente, el operador de mezcla devuelve una LTS que combina la información provista por las descripciones parciales originales.

Es necesario destacar que construir los modelos que son compuestos sigue siendo una tarea difícil que requiere de un intenso trabajo y un grado considerable de experiencia. Para mitigar este problema, MTSA también provee la funcionalidad que permite sintetizar modelos de comportamiento de forma automática, a partir de especificaciones declarativas de los requerimientos, escenarios y casos de uso.

La palabra clave *constraint* de MTSA se usa de manera conjunta con las propiedades de seguridad (*safety*) formalizadas haciendo uso de Lógica Lineal Temporal de Flujos (FLTL). Para una declaración de tipo *constraint*, MTSA construye automáticamente el modelo de LTS que caracteriza a todos los modelos LTS libres de *deadlock* que satisfacen la fórmula

FLTL. Al sintetizar y mezclar modelos LTS obtenidos con definiciones FLTL, se puede construir de forma iterativa una LTS que caracteriza a la cota superior de los sistemas de comportamiento esperados.

La palabra clave *abstract* de MTSA puede aplicarse a procesos FSP. Su semántica es tal que el modelo resultante es la LTS de menor refinamiento que garantiza el comportamiento requerido por los procesos FSP. Esta palabra clave, utilizada en conjunción con los procesos FSP que modelan el comportamiento descrito en la especificación del escenario, provee una LTS que caracteriza a todas las implementaciones que satisfacen dicha especificación.

### 3.2. Análisis

Habiendo construido una aproximación inicial del comportamiento esperado del sistema, el análisis pasa a ser una tarea crucial que puede brindar información del dominio, tanto del problema como de la solución, aumentando la confianza que se tiene de la adecuación y correctitud del software y llama a proseguir la elaboración del modelo parcial.

La herramienta MTSA soporta varios tipos de análisis, el más básico involucra la inspección de modelos LTS y está soportado a través de la construcción automática de representaciones visuales de los modelos LTS escritos usando FSP. Esta inspección queda sujeta al tamaño del modelo, limitación tal que puede mitigarse haciendo uso de los operadores de minimización y ocultamiento.

Aunque la inspección y animación no permiten una exploración exhaustiva de los modelos LTS, MTSA implementa un número de técnicas de análisis automáticas para éste propósito. En particular, MTSA permite verificar si un modelo LTS satisface una propiedad expresada en FLTL. Un modelo LTS caracteriza un conjunto de implementaciones, de las cuales algunas pueden satisfacer la propiedad siendo verificada y algunas pueden violarla. Por este motivo MTSA automáticamente verifica una relación de satisfactibilidad trivaluada entre el modelo LTS y una fórmula FLTL. Mientras que un Modelo LTS  $M$  puede caracterizar a un conjunto extremadamente grande, potencialmente infinito, de implementaciones, verificar una propiedad en  $M$  con *model checking* se reduce a dos verificaciones tradicionales de FLTL. Finalmente, MTSA permite verificar si un modelo es



libre de *deadlocks*. Al igual que en el caso de *model checking* para propiedades FLTL, el resultado de esta verificación tiene uno de tres valores: o bien todas las implementaciones exhiben *deadlocks*, o bien todas son libres de *deadlocks* o bien hay una combinación de implementaciones que exhiben *deadlocks* y otras que no.

### 3.3. Modelando problemas de controladores dinámicamente actualizables

Agregamos un conjunto de palabras reservadas a FSP para poder soportar objetivos actualizables. En la figura 3.1 mostramos el código FSP necesario para síntesis de controladores actualizables para el ejemplo del reactor nuclear presentado en la sección 5.1.1.

El operador **updatingController** es utilizado para definir el problema de controladores actualizables. Al ejecutar la composición en paralelo de este elemento obtendremos, si existe, un controlador que satisface los objetivos del problema de control sin violar la especificación del ambiente de actualización. Ambos definidos en el capítulo 4. Necesitamos en esta declaración, además de los nuevos requerimientos, suministrar el controlador actual, el modelo del ambiente actual, el modelo del ambiente nuevo y un conjunto de pares de flujos donde cada elemento indicará correspondencia de propiedades del ambiente actual al ambiente nuevo como dijimos en la definición 4.2.2. Por ejemplo, en la figura 3.1 podemos observar que al usar la palabra reservada **updatingController** configuramos `ENVIRONMENTANDCONTROLLER` como el controlador actual, `OLDENVIRONMENT` como modelo del ambiente actual, `NEWENVIRONMENT` como modelo del ambiente nuevo, `UPDATESPEC` como el conjunto de objetivos para controladores actualizables según la definición 4.2.1 y  $\{\{\text{REQUESTPENDING}, \text{REQUESTPENDING}\}, \{\text{ISSTOPPED}, \text{ISSTOPPED}\}\}$  es el conjunto de pares de flujos donde se indica que por cada estado del ambiente actual, existe una correspondencia con estados del ambiente nuevo si la valuación de los flujos de los primeros en cada par, iguala a la valuación de los segundos en cada par. Además, en este ejemplo, `OLDENVIRONMENT` y `NEWENVIRONMENT` poseen distintos nombres, pero son el mismo ambiente puesto a que este ejemplo es una actualización que no requiere hacer un cambio en el ambiente.

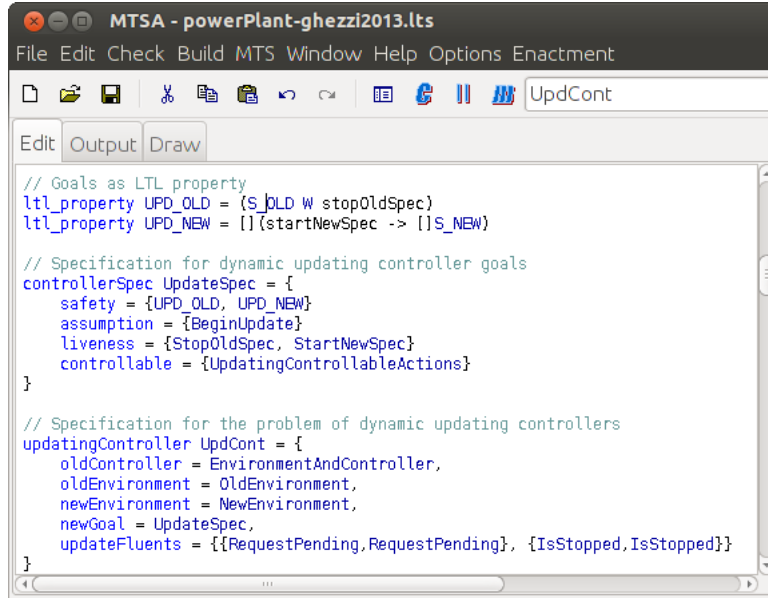


Figura 3.1: MTSA – Definición de problema de controladores dinámicamente actualizables

Los objetivos nuevos están definidos mediante el operador **controllerSpec**. La palabra reservada **safety** permite definir requerimientos de seguridad (*safety*). Aquí incluiremos fórmulas FLTL definidas en la sección 2.3 y, en particular para controladores actualizables dinámicamente, escribiremos las fórmulas FLTL 1, 2 y 3 de acuerdo a la definición 4.2.1. Como se puede ver en la figura 3.1, UPD\_OLD representa la fórmula 1, UPD\_NEW corresponden a la fórmula 2 y la fórmula 3 no está especificada debido a que en este ejemplo configuramos a la fórmula 3 como *true*. S\_OLD y S\_NEW son aserciones FLTL que representan a los objetivos viejos ( $G$ ) y a los objetivos nuevos ( $G'$ ) respectivamente.

La sección de *liveness* esta definida usando los operadores **assumption** y **liveness** que especifican las asunciones del ambiente y objetivos de *liveness* respectivamente. Ambas deberán definirse mediante FLTLs. En la figura 3.1, BEGINUPDATE, STOPOLDSPEC y STARTNEWSPEC son los flujos que se inicializan apagados y se prenden cuando la acción del nombre del flujo sucede. Luego, el controlador sintetizado nos garantizará que si infinitamente la actualización es requerida (i.e sucede la acción *beginUpdate*), las acciones *stopOldSpec* y *startNewSpec* sucederán infinitamente. Por lo tanto, con estos dos operadores definimos las fórmula 4 y 5 de la definición 4.2.1.

---

Por último, la sección de **controllableActions** detalla el conjunto de acciones consideradas controlables a la hora de resolver el problema de control. En nuestro ejemplo, utilizamos a `UPDATINGCONTROLLABLEACTIONS` que posee las acciones `OK`, `STARTNEWSPEC`, `STOPOLDSPEC`, `ADFASDFAS`.



## 4. PROBLEMA DE ACTUALIZACIÓN DE CONTROLADORES DINÁMICAMENTE

En este capítulo explicaremos primero como la actualización dinámica de controladores está formalmente especificada, luego como dicha especificación es convertida a un problema de síntesis de controladores. Por último, detallaremos como este problema de síntesis de controladores se resuelve.

### 4.1. Especificación

En capítulos anteriores mostramos ejemplos en los cuales nos referíamos a la actual y nueva especificación como entidades monolíticas (las llamaremos  $S$  y  $S'$  respectivamente). Sin embargo, para presentar la actualización dinámica de controladores como un problema de control asumiremos que la especificación  $S$  esta dividida en tres partes. La primera es la descripción de la interfaz del controlador dada como un conjunto de etiquetas  $A$  que representa las acciones controlables del controlador. La segunda es una descripción operacional  $E$ , en forma de LTS, que describe el comportamiento del ambiente en cuanto a acciones controlables y monitoriables. La tercera son los objetivos del controlador  $G$ , expresado como una formula FLTL.

Tenga en cuenta que como el estado inicial de  $E'$  depende el estado actual de  $E$ , el ingeniero debe proveer un mapeo de estados  $M$  de  $E$  a  $E'$ . Omitiremos referirnos a  $M$  hasta que lleguemos a la sección donde formalizaremos todos estos conceptos para mantener la presentación simple.

En consecuencia, el problema general que apuntamos a resolver puede ser planteado de la siguiente manera: Ejecutar un sistema adaptable mediante un controlador  $C$  que controla un conjunto de acciones  $A$  y a su vez satisface los objetivos  $G$  para un ambiente  $E$ . El usuario desea cambiar dinámicamente el controlador que se esta ejecutando para empezar a satisfacer el nuevo objetivo  $G'$  en el nuevo ambiente  $E'$  controlando un conjunto

de acciones  $A'$  y a su vez satisfacer los requerimientos de transición  $T$ .

Por ejemplo, en el caso de estudio del buscador UAV de vida salvaje 5.1.3, el conjunto de acciones controlables para la especificación original ( $A$ ) incluye *start*, *return2base*, *scan*, *picture* y *goto*. Las acciones controlables para la nueva especificación ( $A'$ ) posee más extensa ya que provee nuevas capacidades para el controlador como por ejemplo *follow*.

Los modelos de los ambientes  $E$  y  $E'$  pueden ser construidos mediante la composición paralela de varios modelos LTSs que describen diferentes aspectos del comportamiento del ambiente. Por ejemplo podríamos definir una LTS que describa el comportamiento de un tren que va atravesando distintas secciones de una vía. Por otro lado, tenemos otra LTS que describe la comunicación que el tren realiza con la barrera electrónica para identificar si está habilitado para cruzar. Para definir por completo el ambiente, necesitamos componer en paralelo ambas LTS.

## 4.2. La actualización dinámica de controladores como un problema de síntesis de controladores

Una primera aproximación intuitiva para implementar la actualización de controladores dinámicamente puede ser construyendo dos controladores adicionales al controlador actual  $C$ . El primero es un controlador  $C'$  que puede satisfacer los objetivos  $G'$  para el nuevo ambiente  $E'$  controlando acciones  $A'$ . El segundo es un “controlador de transición”  $C^*$  que controla el traspaso del controlador actual  $C$  al nuevo controlador  $C'$  satisfaciendo los requerimientos de transición  $T$ .

Si bien es conceptualmente elegante, el enfoque de los tres controladores es ingenuo ya que estos controladores están intrínsecamente relacionados. La nueva especificación sólo puede ser alcanzada por un controlador  $C'$  desde estados específicos ( $I_{C'}$ ). Estos estados iniciales de  $C'$  necesitan ser calculados y considerados como estados finales del “controlador de transición”  $C^*$ . Luego, los estados desde donde el “controlador de transición” puede alcanzar  $I_{C'}$  necesitan ser computados ( $I_{C^*}$ ). Finalmente, nos queda analizar si  $C$  puede ser extendido para garantizar que alcance algún estado en  $I_{C^*}$  sin violar sus objetivos ( $G$ ).

La interacción entre estos tres controladores y la necesidad de generar una técnica para computar  $C'$  y  $C^*$  puede ser obtenida mediante la resolución de un problema de control que produce un sólo controlador (el cual vamos a llamar  $C_u$ ) que ejecuta acciones simulando las tres fases, primero simulando a  $C$ , luego a  $C^*$  y finalmente a  $C'$ . Ahora pasaremos a explicar como la actualización de controladores dinámica puede ser expresada como un problema de control que abarca estas tres fases.

El problema de control para la actualización de controladores dinámicos, como cualquier otro problema de control, necesita de un modelo del ambiente, el cual llamaremos  $E_u$ , un objetivo, el cual llamaremos  $G_u$ , y un conjunto de acciones,  $A_u$ . El objetivo  $G_u$ , lo definiremos en términos de  $G$ ,  $G'$  y  $T$  más los eventos *stopOldSpec*, *startNewSpec* y *beginUpdate*. El ambiente  $E_u$  estará definido en base a  $E$ ,  $E'$  y  $C$ . El conjunto de acciones  $A_u$  estará definido en base a  $A$  y  $A'$ .

#### 4.2.1. El objetivo del problema de control

La formalización del objetivo para la actualización de controladores dinámicamente,  $G_u$ , puede ser formalizado como una conjunción de las siguientes formulas FLTL.

**Definición 4.2.1.** (*Objetivo para el problema de control de la actualización de controladores dinámicamente*) Sean  $\Box G$  y  $\Box G'$  los objetivos actuales y los nuevos para un escenario de actualización de controladores dinámicamente, donde  $G$  y  $G'$  son una combinación Booleana de flujos y  $T$  es una propiedad de safety. Definimos a  $G_u$ , el objetivo para el problema de control de la actualización de controladores dinámicamente como la conjunción de las siguientes fórmulas FLTL:

1.  $\Box(G \text{ } W \text{ } stopOldSpec)$
2.  $\Box(startNewSpec \implies \Box G')$
3.  $\Box T$
4.  $\Box(beginUpdate \implies \Diamond startNewSpec)$
5.  $\Box(beginUpdate \implies \Diamond stopOldSpec)$

La primer fórmula requiere que los objetivos viejo  $G$  valgan hasta que el controlador active la señal *stopOldSpec*. Tenga en cuenta que, esta propiedad de manera aislada significa que la especificación vieja, cualesquiera sean sus estados, puede dejar de valer en

cualquier momento. Esto no es lo deseado para una actualización dinámica, por lo tanto debemos restringirlo en la especificación de transición ( $T$ ).

La segunda fórmula simplemente requiere que la nueva especificación empiece a valer desde el momento en que el controlador active la señal *startNewSpec*. Esto forzará al controlador a que sólo produzca esta señal cuando puede asegurar  $G'$ .

La tercera fórmula indica que los requerimientos de transición deben valer siempre. Restringimos  $T$  a que sea una propiedad de seguridad (*safety*).  $T$  es esperado que predique sobre eventos *stopOldSpec* y *startNewSpec* para poder restringir el comportamiento del sistema cuando ni  $G$  ni  $G'$  valen. Un enfoque más preciso sería requerir que  $T$  sólo valga entre ambos eventos, esto sería  $\Box inTransition \implies T$  donde  $inTransition = \langle \{stopOldSpec\}, \{startNewSpec\}, \perp \rangle$ . Esta idea es muy restrictiva ya que los requerimientos de transición pueden necesitar referirse a situaciones que suceden antes que la especificación vieja deje de valer. Por ejemplo, en el caso del buscador UAV de vida salvaje 5.1.3, una foto tomada antes de *stopOldSpec* que no fue procesada antes *stopOldSpec*. O puede ser necesario referirse a situaciones que suceden luego de *startNewSpec*. En nuestro ejemplo no empezar a satisfacer la nueva especificación hasta que todas las fotos sin procesar sean procesadas.

Finalmente, las ultimas dos fórmulas requiere que el controlador luego de ejecutar la acción *beginUpdate*, continúe con el procedimiento y garantice que los eventos *stopOldSpec* y *startNewSpec* sucedan.

#### 4.2.2. Modelo del ambiente del problema de control

El modelo del ambiente  $E_u$  para la actualización de controladores dinámicamente debe ser construido para cubrir las tres fases del controlador a ser sintetizado: el ambiente para la especificación vieja, el ambiente para la nueva especificación, y el ambiente para la transición. A grandes rasgos, esto significa definir a  $E_u$  para que sea una combinación de  $E$  y  $E'$  más el agregado de transiciones para los eventos *beginUpdate*, *stopOldSpec* y *startNewSpec*. Por otro lado, hay dos problemas que debemos tener en cuenta cuidadosamente a la hora de definir formalmente  $E_u$ .

El primer concepto clave que debemos considerar en la construcción de  $E_u$  esta re-



lacionado en buscar un ambiente que permita un hot-swap transparente del controlador actual con el obtenido por la síntesis. Necesitamos que la actualización del controlador sea estructuralmente idéntico al controlador actual hasta que la acción *beginUpdate* suceda. Este requerimiento es crucial para lograr establecer un mapeo de estados desde el controlador actual a los estados de controlador actualizable. Esto hace que sea trivial que debemos establecer el estado inicial del nuevo controlador basándonos en el estado actual del controlador viejo, y luego intercambiar controladores durante la ejecución sin perder información del estado. De esta manera, el nuevo controlador va a continuar ejecutando exactamente de la misma manera que como lo hacía el viejo hasta el momento que se inicie el proceso de actualización.

Para lograr esta propiedad en el controlador de actualización definiremos su ambiente en su parte inicial como la composición paralela del controlador viejo y su ambiente (es decir  $E||C$ ). Esto apunta a construir un controlador actualizable que inicialmente ejecutará en un ambiente que ya esta siendo controlado por  $C$ . Tenga en cuenta que como no queremos que la actualización se interponga con  $C$  antes que *beginUpdate* suceda, vamos a asegurar que en esta fase el controlador actualizable no controle nada, sólo monitorea lo que sucede.

Es cuando *beginUpdate* sucede, que el controlador actualizable debe empezar a tomar medidas e intentar garantizar la transición correcta para satisfacer la nueva especificación. Es decir que luego de que la actualización es solicitada necesitamos deshabilitar  $C$  y por lo tanto, cambiar de fase del ambiente.

Entonces, el ambiente  $E_u$  es construido para ser como  $E||C$  y luego, cuando *beginUpdate* sucede, empezará a comportarse como  $E$  hasta que la nueva especificación se pueda cumplir, momento en el cual  $E_u$  se comportará como  $E'$ .

El segundo problema que debemos manejar en la construcción de  $E_u$  está relacionado con preservar el estado del ambiente cuando cambia desde  $E$  a  $E'$ . Esto es, que el estado inicial del modelo del nuevo ambiente al momento de hacer la actualización debería estar configurado de acuerdo al estado actual del modelo del ambiente viejo. Por ejemplo en nuestro caso de estudio del buscador UAV de vida salvaje de la sección 5.1.3, si el sistema está en un estado de batería baja, entonces el estado del modelo del nuevo ambiente debería reflejar esto. Por lo tanto, lo que es necesario es un mapeo de estados

que relacione estados del modelo del ambiente viejo  $E$  al nuevo  $E'$ .

El mapeo desde estados de  $E$  a  $E'$  puede ser definido de varias maneras, sólo necesitamos que todos los estados de  $E$  tengan al menos un estado correspondiente en  $E'$ . Tenga en cuenta que, permitimos subespecificar la correspondencia de estados de  $E'$  permitiendo, por ejemplo, una evolución no determinística desde un estado cuyo valor de batería es  $\neg LowBattery$  a un estado donde el valor es  $MidBattery$  o  $\neg HighBattery$ . Durante este capítulo vamos a asumir una relación de mapping de estados  $M \subset S_E \times S_{E'}$  que cubre a todos  $S_E$ . En la herramienta que desarrollamos, por conveniencia, definimos el mapeo naturalmente sobre definiciones de flujos: un estado  $E$  esta mapeado a todos los estados de  $E'$  que preservan el valor de flujos.

Ahora definiremos formalmente el ambiente  $E_u$  para el problema de control de la actualización de controladores dinámicamente:

**Definición 4.2.2.** (*Ambiente para el problema de control de la actualización de controladores dinámicamente*) Sean  $C$  el controlador actual,  $A$ ,  $E$  y  $G$  la especificación actual y  $A'$ ,  $E'$  y  $G'$  la especificación nueva para el problema de control de actualización, donde  $E = (S_E, A_E, \Delta_E, s_{E_0})$  y  $E' = (S_{E'}, A_{E'}, \Delta_{E'}, s_{E'_0})$ . Además, sea  $M \subset S_E \times S_{E'}$  un mapeo de estados tal que para todo  $s \in S_E$  existe un estado  $s' \in S_{E'}$ . El ambiente para el problema de control de la actualización de controladores dinámicamente ( $E_u$ ) es un LTS  $(S_u, A_u, \Delta_u, s_u)$  tal que  $E_u$  es una unión disjunta de estados en  $E \parallel C$ ,  $E$  y  $E'$  (i.e.  $E_u = S_{E \parallel C} \uplus S_E \uplus S_{E'}$ ),  $s_u = s_{E \parallel C}$ ,  $A_u = A_E \cup A_{E'}$   $\bar{\ell} | \ell \in A$  y  $\Delta_u$  es la relación más pequeña que satisface las reglas que siguen a continuación donde  $\ell \in A_u$ :

- [1] si  $(s, \ell, s') \in \Delta_{E \parallel C} \wedge \ell \notin A$  entonces  $(s, \ell, s') \in \Delta_u$
- [2] si  $(s, \ell, s') \in \Delta_{E \parallel C} \wedge \ell \in A$  entonces  $(s, \bar{\ell}, s') \in \Delta_u$
- [3] si  $(s, t) \in S_{E \parallel C}$  entonces  $((s, t), beginUpdate, s) \in \Delta_u$
- [4] si  $(s, \ell, s') \in \Delta_E$  entonces  $(s, \ell, s') \in \Delta_u$
- [5] si  $s \in \Delta_E$  entonces  $(s, stopOldSpec, s) \in \Delta_u$
- [6] si  $s \in \Delta_{E'}$  entonces  $(s, stopOldSpec, s) \in \Delta_u$
- [7] si  $(s, s') \in M$  entonces  $(s, startNewSpec, s') \in \Delta_u$
- [8] si  $(s, \ell, s') \in \Delta_{E'}$  entonces  $(s, \ell, s') \in \Delta_u$

Una representación gráfica informal de  $E_u$  esta representado en la figura 4.1

Las definiciones anteriores construyen a  $E_u$  para comportarse exactamente igual a  $E \parallel C$

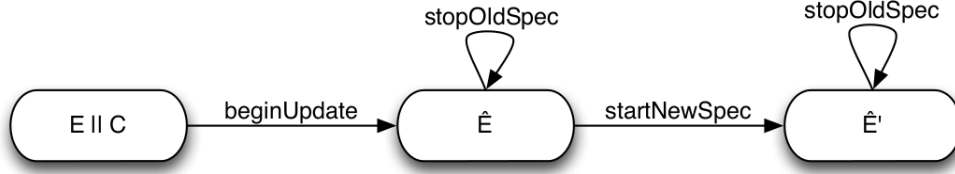


Figura 4.1: Representación gráfica del ambiente  $E_u$  para el problema de control de la actualización de controladores dinámicamente -  $\hat{E}$  es  $E$  con la diferencia de que su estado inicial es el estado actual de  $E$  en  $E||C$ .  $\hat{E}'$  es  $E'$  con la diferencia de que su estado inicial es alguno de los estados relacionados en  $M$  con el estado actual de  $E$ .

(ver reglas 1 y 2) hasta que sucede *beginUpdate*. La regla 2 simplemente renombra acciones controladas de  $C$  para impedir que el controlador de actualización las controle. Juntas, ambas reglas permiten intercambiar  $C$  por  $C_u$  mientras aseguramos que  $C_u$  continua ejecutando exactamente de la misma manera que  $C$ . Una vez que *beginUpdate* se efectúa (regla 3), el ambiente se comporta como  $E$  (regla 4). Es justo en este momento que el nuevo controlador tendrá que mantener la vieja especificación pero forzándolo a llegar a un estado desde el cual los requerimientos de transición puedan ser satisfechos y luego alcanzar la nueva especificación. En cualquier momento el controlador puede efectuar la acción *stopOldSpec* (regla 5) e incluso *startNewSpec* (regla 7). En caso de que ya haya ocurrido la acción *startNewSpec*, el ambiente de actualización  $E_u$  se comporta como el nuevo ambiente  $E'$  (regla 8). Aunque como no forzamos que la vieja especificación deje de valer antes de que valga la nueva especificación, *stopOldSpec* puede ocurrir durante esta última fase (regla 6).

#### 4.2.3. El problema de control de la actualización de controladores dinámicamente

Ahora podemos definir formalmente el problema de control que nos soluciona el escenario de la actualización de controladores dinámicamente.

El problema de control de la actualización de controladores dinámicamente puede ser visto como un problema de control de LTS usando un ambiente  $E_u$  como definimos en la definición 4.2.2, los objetivos  $G_u$  como definimos en la definición 4.2.1 y el conjunto de acciones controlables  $A_u$  definido como la unión de las acciones controlables entre  $A$

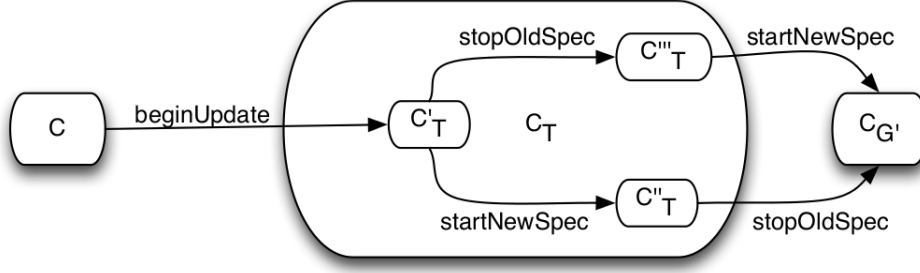


Figura 4.2: Abstracción informal de un controlador  $C_u$  para el problema de control de la actualización de controladores dinámicamente -  $C_T$  representa el comportamiento que satisface  $T$  mientras que garantiza eventualmente tanto  $stopOldSpec$  como  $startNewSpec$ , mientras que  $C_{G'}$  representa el comportamiento del controlador que satisface  $G'$ .

y  $A'$ . La única sutileza es que como  $E_u$  introduce cambios de nombres de acciones que son controladas por  $C$  en la primera fase ( $E||C$ ), estos cambios deben ser revertidos una vez computado el controlador de actualización (reemplazar las acciones  $\bar{\ell}$  por  $\ell$ ). Recordar que el cambio de nombre es realizado para asegurar que el controlador de actualización, cuando se esta generando, no considere a las acciones de  $E||C$  como controlables hasta que suceda  $beginUpdate$ . Una vez computado, mientras el controlador de actualización es ejecutado las acciones deben ser revertidas a su nombre original.

**Definición 4.2.3.** (El problema de control de la actualización de controladores dinámicamente) Sean  $G$  y  $G'$  expresiones de flujos sin expresiones temporales. Sea  $E$  un LTS y sea la formula FLTL  $\Box G$  la especificación actual que el sistema esta satisfaciendo tras ejecutar el LTS  $C$ , controlador que controla las acciones en  $A$ . Sea  $E'$  un LTS, la formula FLTL  $\Box G'$  y un conjunto de acciones controlables  $A'$  la nueva especificación que es esperada satisfacer con la actualización dinámica del controlador del sistema.

$C_u$  es una solución al problema de control de actualización de controladores dinámicamente si  $C_u$  es el resultado de renombrar toda acción  $\bar{\ell}$  por  $\ell$  en el controlador  $\bar{C}$  que es una solución al problema de control LTS  $\langle E_u, G_u, A_u \rangle$  donde  $E_u$  esta definido en la definición 4.2.2,  $G_u$  esta definido en la definición 4.2.1 y  $A_u = A \cup A'$ .

Por definición, cualquier solución del problema de control de la actualización de controladores dinámicamente satisfacerá a  $G_u$  asumiendo que el ambiente se comporta como

$E_u$ . La validez de las asunciones de  $E_u$  dependen en la validez de las especificaciones actuales y nuevas,  $E$  y  $E'$ , que son responsabilidad del ingeniero, y que la infraestructura de ejecución del controlador va a dinámicamente cargar cualquier nueva habilidad descrita en  $E'$  cuando la señal *startNewSpec* suceda.

Un controlador que es solución del problema definido recientemente tomará, informalmente, la estructura descrita en la Figura 4.2. Primero va a comportarse como  $C$  excepto que aceptará en cualquier momento que la acción no controlable *beginUpdate* suceda. Luego, se comportará como un controlador que esta intentando realizar las acciones *stopOldSpec* y *startNewSpec* mientras satisface la propiedad  $T$ . Finalmente, una vez que la acción *startNewSpec* ha ocurrido se comportará como un controlador que satisface  $G'$ .

### 4.3. Resolviendo el problema de actualización dinámica de controladores

Resolver un problema de control LTS (como el definido en la definición 2.4.1 en el capítulo 2) por cada propiedad FLTL es 2EXPTIME complete [PR89]. El problema de actualización dinámica de controladores es una instancia específica del problema de control LTS general. En efecto, dada la estructura de  $G_u$ , su resolución puede ser computada bajo límites de menor complejidad.

Bajo la asunción de que  $G$ ,  $G'$  y  $T$  son propiedades de seguridad,  $G_u$  puede ser codificado como propiedades de obligación (por ejemplo disyunciones de afirmaciones de seguridad y alcanzabilidad,  $\bigwedge_{i=1}^n (\Box I_i \vee \Diamond R_i)$ ). Los problemas de control LTS con objetivos de obligación pueden ser resueltos en tiempo lineal para modelos de ambientes determinísticos. Para ambientes no determinísticos, un especializado subconjunto de construcciones pueden usarse para producir versiones determinísticas, sin embargo, podría crecer exponencialmente dependiendo del grado de no determinismo que exista.

Es simple de ver que el objetivo del problema de control de actualización dinámica de controladores,  $G_u$ , puede ser reformulado como afirmaciones de obligación. Las formulas 1) a 3) en la definición 4.2.1 son propiedades de seguridad (tenga en cuenta que  $W$  es un “weak until” y por lo tanto es una propiedad de seguridad). Una pro-

propiedad de seguridad  $I_j$  puede simplemente ser codificada como  $\Box I_j \vee \Diamond \perp$ . Las otras dos formulas de la definición 4.2.1 son de la forma  $\Box(p \implies \Diamond q)$  y pueden ser codificadas como  $\Box \neg \text{UpdateBegan} \vee \Diamond(\text{UpdateBegan} \wedge \text{NewSpecStarted})$  y  $\Box \neg \text{UpdateBegan} \vee \Diamond(\text{UpdateBegan} \wedge \text{OldSpecStopped})$  donde los flujos  $\text{NewSpecStarted}$ ,  $\text{OldSpecStopped}$  y  $\text{UpdateBegan}$  modelan que sus respectivos eventos han sucedido alguna vez en el pasado. Formalmente estarían definidos como  $\langle \text{startNewSpec}, \emptyset, \perp \rangle$ ,  $\langle \text{stopOldSpec}, \emptyset, \perp \rangle$  y  $\langle \text{beginUpdate}, \emptyset, \perp \rangle$ .

El problema de control de la actualización dinámica de controladores puede tener o no una solución. La existencia de esta solución significa que el controlador resultante  $C_u$  garantiza que: se comporte inicialmente como el controlador actual  $C$  bajo la especificación del ambiente actual  $E$  y acepta en cualquier punto el comando  $\text{beginUpdate}$ ; asegura una transición correcta, satisfaciendo  $T$ , preservando el objetivo viejo  $G$  y conduciendo al sistema a un punto en el cual ( $\text{startNewSpec}$ ) el nuevo ambiente  $E'$  puede ser reemplazado y el nuevo objetivo  $G'$  puede ser garantizado.

La no existencia de una solución a este problema de control puede significar dos posibles escenarios. El primero es que el nuevo objetivo bajo las asunciones del nuevo ambiente no puede ser alcanzados por el controlador (esto es, el problema de control definido por  $E'$  y  $\Box G'$  no tiene solución para ningún estado inicial de  $E'$ ). Esta es una situación extrema donde el controlador tiene poco por hacer con la actualización que se le pide. El segundo, asumiendo que el problema de control definido por  $E'$  y  $\Box G'$  tiene solución para algunos estados de  $E'$ , puede suceder que la transición de una especificación a la otra no pueda satisfacer la propiedad  $T$ , o sino puede suceder que sea imposible alcanzar un estado de  $E'$  desde el cual  $G'$  valga. Este segundo escenario puede producirse, por ejemplo, porque el requerimiento de transición  $T$  es excesivamente restrictiva o porque el controlador tiene capacidades insuficientes, para alcanzar un estado desde el cual el nuevo ambiente puede garantizar la validez del nuevo objetivo. Más generalmente, tendremos el problema cuando tenemos una combinación de especificaciones muy estrictas ( $E$ ,  $G$ ,  $E'$ ,  $G'$  y  $T$ ) y falta de control del controlador sobre los eventos del ambiente.

## 5. VALIDACIÓN

En este capítulo reportaremos los casos de estudio que corrimos para validar nuestro enfoque. El propósito de los casos de estudio es mostrar la aplicación del enfoque mediante la resolución de casos de estudios tomados de trabajos previos y a su vez, seguir analizando las virtudes y defectos de los trabajos previos existentes de actualización dinámica de controladores.

Según nuestro conocimiento, el primer y único trabajo que investiga sobre actualización dinámica de controladores donde hay un cambio de especificación explícito es en Ghezzi et al. [GGM12]. Ellos adoptan un criterio general, natural y correcto. Una actualización dinámica es correcta si el comportamiento exhibido por el sistema es equivalente al obtenido luego de una actualización apagando la máquina. Esto relaja el esfuerzo del ingeniero al no tener que especificar requerimientos de transición (como en nuestro trabajo) pero con el costo de limitar posibles actualizaciones que pueden ser soportadas. Como en [GGM12], permitimos especificaciones solapadas (ver Figura 5.1); pero también permitimos periodos en los que ninguna especificación vale a diferencia de [GGM12] (ver Figura 5.2).

Es posible obtener el comportamiento de actualización obtenido en [GGM12] especificando como parte del requerimiento de transición  $T$  que *startNewSpec* pueda ocurrir si la nueva especificación vale desde el último estado inicial antes de *beginUpdate* (ver la imagen de abajo de la Figura 5.3) o tan pronto como el estado inicial es alcanzado nuevamente (ver la imagen de arriba de la Figura 5.3). Esto, podemos formalizarlo de la

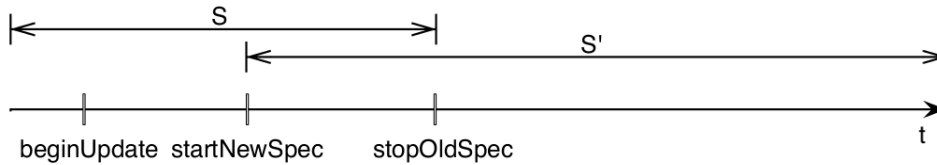


Figura 5.1: Abstracción de la línea de tiempo de la actualización dinámica de controladores. Escenario en el cual la nueva especificación está garantizada antes de que la vieja especificación deje de valer.

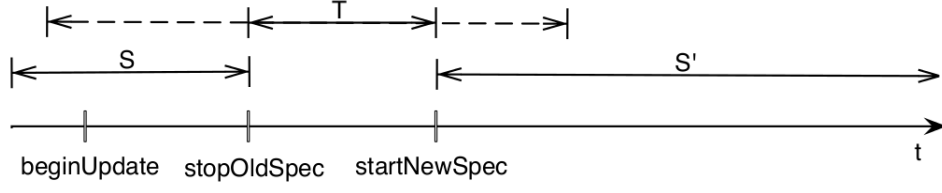


Figura 5.2: Abstracción de la línea de tiempo de la actualización dinámica de controladores. Escenario en el cual hay un periodo donde ni la vieja, ni la nueva especificación vale.

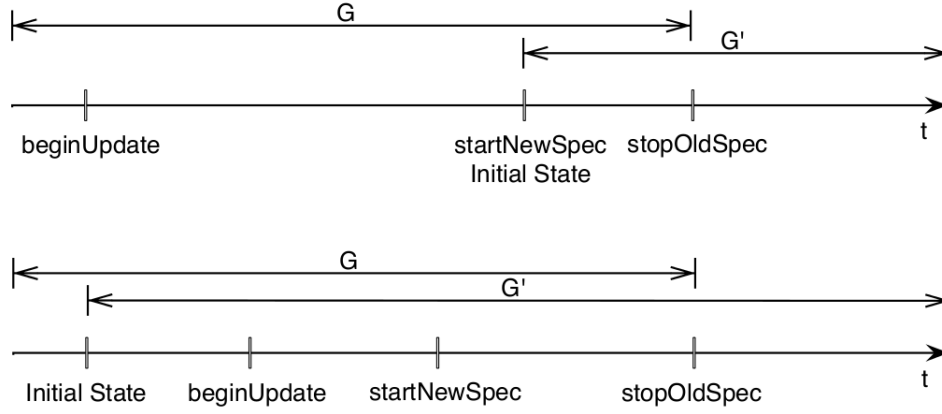


Figura 5.3: Relación de eventos relevantes en la actualización de controladores dinámicamente para estados actualizables según [GGM12]

siguiente manera:

$$\Box[(LastInitBeforeUpdate \wedge G' W startNewSpec) \vee (startNewSpec \implies Init)] \quad (5.1)$$

donde  $LastInitBeforeUpdate = Init \wedge \bigcirc(\neg Init W beginUpdate)$  y  $Init$  representa estar en un estado inicial de  $C$ .

En [MGGB13], tres criterios de actualización mas débiles son introducidos para permitir actualizaciones en sistemas donde el estado inicial no es visitado nuevamente. Por ejemplo, la noción de estados co-inicial (estados que son similares al estado inicial), expande las situaciones en las cuales la actualización es permitida. De todos modos, no hay garantías, para ninguno de los nuevos criterios, que el criterio correcto original si tiene. La falta de garantías requiere de un ingeniero que valide el controlador resultante. En nuestro trabajo, involucramos a un ingeniero capacitado y habilitado para proveer una especificación de un criterio correcto para la actualización ( $T$ ).



## 5.1. Casos de Estudio

### Configuración experimental

Todos los casos de estudios se ejecutan usando una extensión de la herramienta MTSA [DFCU08] que nos da soporte para especificar LTSs y propiedades, usando una notación textual, álgebra de procesos y FLTL. La herramienta también nos da soporte para síntesis de controladores para problemas de control SGR(1), los cuales son estrictamente más caros de lo que es requerido para problemas de control en la actualización de controladores dinámicamente. La herramienta fue extendida para automáticamente computar  $A_u$ ,  $G_u$  y  $E_u$ , y resuelve el problema de control de la actualización dinámica de controladores. La versión extendida de la herramienta y la especificación completa de todos los casos de estudio pueden verse en [ ].

#### 5.1.1. Planta de energía nuclear

En [MGGB13] se analiza un controlador para un sistema de refrigeración de una planta de energía nuclear. El controlador actual debe ejecutar servicios de mantenimiento primero apagando la bomba que refrigera la planta y luego reiniciarla. El nuevo controlador no necesita parar la bomba y reiniciarla para efectuar el mantenimiento. Hay también un invariante del sistema que indica que la bomba de refrigeración no puede estar apagada indefinidamente (esto puede llevar a un accidente devastador).

Los autores muestran que si una actualización se lleva a cabo en un estado en donde el controlador actual tiene la bomba apagada, entonces el nuevo controlador puede no reiniciar la bomba, generando un accidente. Esto nos muestra dos problemas. Primero, que la forma segura de preservar el invariante del sistema es requerir que la actualización preserve el comportamiento como si el controlador actual haya sido reemplazado desde su estado inicial (cuando la bomba es reiniciada). Esto es lo que los autores aseguran en [GGM12]. Segundo, que las restricciones de cuándo una actualización de [GGM12] es suavizada como en [MGGB13] puede generar fallas y consecutivamente ellos recomiendan que “el diseño de controladores actualizables dinámicamente basados en el criterio débil

de actualización incluye una subsiguiente validación de los controladores”.

### 5.1.2. RailCab

El sistema de RailCab es un sistema mecatrónico que fue desarrollado en la Universidad de Paderborn [oP14] donde los vehículos autónomos deben coordinar de que manera transportar pasajeros o bienes cada vez que le es solicitado. El sistema que se discute en [GGM12] como caso de estudio base es controlar RailCabs cuando está acercándose a un cruce de vías. El RailCab puede monitorear eventos como *endOfSection* que determinan que esta entrando a un cruce de vía, y otros eventos que indican si esta habilitado para usar el freno o si tiene que utilizar el freno de emergencia, que los llamaremos *lastBrake* y *lastEmergencyBrake*. El RailCab controla el freno (*brake*) y el freno de emergencia (*emergencyBrake*) e incluso puede recibir respuestas a pedidos que controla como *requestPermissionToEnter*.

La especificación del controlador actual requiere que el RailCab entre al cruce de vías sólo si se le ha garantizado el permiso para hacerlo. La especificación también describe varias asunciones de que la respuesta al pedido de permiso es contestada antes de que el RailCab llegue a la sección de *lastEmergencyBrake*, siempre y cuando, el pedido haya sido realizado antes de llegar a la zona de *lastBrake*. Como ejemplo de una nueva especificación, en [GGM12] discuten hacer un objetivo más fuerte requiriendo que el RailCab envíe otro pedido preguntando si las barreras están funcionando antes de entrar al cruce de vía. También refuerzan la hipótesis del ambiente en relación a cuando la respuesta a *checkCrossingStatus* puede ser recibida.

### 5.1.3. Buscador UAV de vida salvaje

Este caso de estudio consta de un robot UAV (unmanned aerial vehicle), un típico caso de sistema adaptable, que posee un controlador para buscar vida salvaje en una área de preservación. El UAV cuenta con un algoritmo de reconocimiento de vida salvaje que requiere que el UAV vuele en el rango de 40 a 50 metros y trasmita una foto por cada posible objetivo y debe volver a la base cuando la señal de batería baja se dispara. Durante la misión, se decide que el UAV debe ahora inmediatamente seguir al primer

posible objetivo que encuentre en vez de continuar su búsqueda. Además, deberá transmitir una foto de este animal cada 5 minutos. La nueva misión, requiere que el UAV vuele en un rango entre 20 y 30 metros de altura y use un algoritmo de reconocimiento mejorado. Los módulos del software nuevo van a estar cargados en el UAV y el controlador debe ser actualizado para lograr sus nuevos objetivos usando los nuevos módulos.

Es importante destacar que los rangos de alturas no solapados implican que un período de transición serán requeridos entre las dos misiones donde el UAV deberá volar entre 30 a 40 metros de altura y ninguna especificación se satisface. A su vez, existen otros requerimientos que pueden ser relevantes para el período de transición. Por ejemplo, la nueva especificación no tiene en cuenta lo que sucederá con las imágenes que están siendo transmitidas por cada animal encontrado. Consecuentemente, una actualización que sucede entre que un animal es encontrado y la transmisión de la foto puede llevar a perder la foto. Por lo tanto, un requerimiento para el período de transición puede incluir que a la hora de dejar de satisfacer la especificación vieja, no haya imágenes pendientes para transmitir.

#### 5.1.4. Production Cell

Consideramos varios escenarios de actualización para el caso de estudio de Production Cell presentado en [LL95]. El caso de estudio es sobre una fábrica que manufactura diferentes tipos de productos, donde cada uno de ellos, requiere un proceso de producción que involucra aplicar distintas herramientas en diferentes ordenes. El sistema de producción de la fábrica debería adaptar su proceso de producción al número de factores como la cantidad de herramientas disponibles, la especificación de como procesar cada tipo de producto, y otras restricciones como por ejemplo, un requerimiento de consumo de energía que restringe el uso concurrente de ciertas herramientas.

Además de las herramientas, la fábrica tiene una bandeja de entrada, una bandeja de salida y un brazo robótico. El brazo robótico es el encargado de mover los productos entre las herramientas y las bandejas. Los productos en crudo llegan a la bandeja de entrada, para que el brazo robótico controle cada producto respetando la especificación y colocando el producto final en la bandeja de salida.

### 5.1.5. Cuadrcóptero

El propósito de este caso de estudio fue experimentar la técnica de actualización dinámica más allá de la especificación y tareas de síntesis discutidos en los casos de estudio previos. El enfoque, entonces, de este caso es poder analizar un controlador resultante ejecutando dicha solución en una infraestructura de adaptación (enactment), tomando un dominio de aplicación específico. Usamos una infraestructura reportada en [BDP<sup>+</sup>13], para poder dinámicamente cargar un controlador en un ARDrone [Par14]. La especificación, tanto vieja como nueva, son simplificaciones del caso de estudio del Buscador UAV de vida salvaje (sección 5.1.3).

El caso de estudio incluye las acciones de despegar (*takeOff*), aterrizar (*land*), detectar marcas mediante la cámara onboard (*read*) y objetivos relacionados con las acciones que el drone ejecuta, obteniendo feedback de los leds onboard, cuando las marcas son leídas (*blink*). La diferencia entre especificación vieja y nueva es que al principio el robot debe parpadear al leer una marca  $x$  y luego de la actualización debe parpadear al leer una marca  $x'$  distinta a la original. Además, luego de la actualización tendremos un contador de batería que irá decrementando su valor a medida que el ARDrone ejecute acciones. El nivel de batería no debe agotarse nunca para satisfacer los objetivos, y para esto el robot puede ejecutar la acción *charge* la que pondrá dicho contador en su nivel más alto.

Por cada una de estas acciones, que son parte de la especificación, definimos una clase *Action* dentro de nuestro enactment framework para ejecutar implementaciones del framework YaDrone [Ham14].

## 5.2. Resultados

En esta sección detallaremos los resultados obtenidos para cada caso de estudio planteado en la sección 5.1. Para comparar diferentes resultados obtenidos por el problema de control notaremos  $Tr(C)$  como el conjunto de trazas del LTS  $C$ . Informalmente, una traza es una secuencia de acciones que un controlador puede realizar. Por lo tanto, dados controladores  $C_1$ ,  $C_2$ , si toda traza de  $C_1$  está en  $Tr(C_2)$  (i.e.  $Tr(C_1) \subset Tr(C_2)$ ) podremos decir que el controlador  $C_2$  tiene más comportamiento que  $C_1$  debido a que en

el controlador  $C_2$  puedo ejecutar todas las secuencias de acciones de  $C_1$  y algunas más.

### 5.2.1. Planta de energía nuclear

Nosotros resolvemos tres diferentes problemas de control de actualización dinámica de controladores para este caso de estudio. El primero no tiene requerimientos para el período de transición entre la especificación vieja y la nueva (esto es,  $T$  configurado como *true*). El controlador obtenido ( $C_u^1$ ) exhibe el comportamiento invalido descrito en [MGGB13], permitiendo una actualización mientras la bomba esta apagada, prendiendo la bomba durante la transición y permitiendo al nuevo controlador a nunca reiniciarla.

El segundo problema de control usa a  $T$  con el requerimiento genérico de [GGM12] (ver fórmula 5.1). El controlador resultante ( $C_u^2$ ) evita la actualización mientras la bomba esta apagada, y por lo tanto, evita que el nuevo controlador deje la bomba apagada sin intención. De hecho, como se espera, este controlador exhibe estrictamente menos comportamiento que el controlador previo ( $Tr(C_u^2) \subset Tr(C_u^1)$ ).

Finalmente, el tercer problema de control modela explícitamente el requerimiento de que la bomba no debería estar apagada continuamente (esto es, en realidad, un requerimiento de transición de aceptación Büchi que puede ser manejado por nuestro enfoque sin problemas. Ver capítulo 6). El controlador resultante ( $C_u^3$ ) no sólo evita el escenario descrito en [MGGB13] y exhibe menos comportamiento que el primer controlador ( $Tr(C_u^2) \subset Tr(C_u^1)$ ) sino que también provee más oportunidades de actualización que el segundo controlador exhibiendo estrictamente más comportamiento ( $Tr(C_u^2) \subset Tr(C_u^3)$ ).

En otras palabras, nuestra técnica provee un criterio más relajado para controladores actualizables con respecto a [GGM12] y que a su vez, es correcto por construcción ya que satisface los requerimientos de transición y además no necesita de una validación manual posterior como en [MGGB13].

### 5.2.2. RailCab

Como en el caso de estudio previo, resolvimos tres problemas de control de actualización dinámica distintos. El primero sin requerimientos para la transición entre la vieja

especificación y la nueva (i.e.  $T = true$ ). El controlador resultante  $C_u^1$  ....

### 5.2.3. Buscador UAV de vida salvaje

### 5.2.4. Production Cell

Para analizar este caso, exploramos varios escenarios de adaptación definiendo una especificación inicial y nueva. Para cada escenario identificamos posibles especificaciones alternativas. Por ejemplo, una decisión común para todo escenario de actualización es como son los productos procesados. ¿Debería la cadena de montaje estar vacía antes de que se produzca la actualización? En este caso, los productos que están siendo procesados ¿deberían ser descartados? ¿o terminarlos? Si decidimos terminarlos, ¿los deberíamos terminar con la vieja especificación o con la nueva? O tal vez, durante el proceso de transición, la cadena de montaje no necesita ser vaciada y no importa que receta se lleva a cabo para construir un producto, siempre y cuando, estén contruidos de una manera consistente.

Estas alternativas fueron modeladas cambiando el valor del requerimiento de transición  $T$ ...

### 5.2.5. Cuadróptero

Un aspecto interesante del controlador resultante, en este caso de estudio, es que en la vieja especificación el nivel de batería nunca es tenido en cuenta, generando en el ambiente de actualización  $E_u$  mucho no determinismo a la hora de efectuar la acción *startNewSpec* desde cualquier estado (i.e. una transición por cada nivel de batería distinto que puede tener el nuevo ambiente). A pesar del no determinismo que este problema de control tiene al cambiar la especificación, el controlador de actualización simplemente carga la batería inmediatamente después de que se efectúe la actualización (i.e. cuando *startNewSpec* ocurre). Esto sucede debido a que el controlador de actualización obtenido asume una actualización en el peor caso, que es cuando no hay batería disponible en el robot.

Por otra parte, Pudimos validar el comportamiento del drone tras observar mediante la herramienta MTSA como los estados del controlador de actualización eran recorridos, y a su vez, observar las acciones reales que el robot efectuaba. La única acción controlada por el

---

operador del sistema vía MTSA fue *beginUpdate*. Un video con uno de estos experimentos puede ser visto en [1].





## 6. CONCLUSIONES Y TRABAJOS FUTUROS

### 6.1. Discusión y trabajo futuro

El problema de actualización dinámica ha sido estudiado extensivamente y hay una gran cantidad de problemas distintos que deben ser abordados en función del dominio de aplicación, tecnología y la forma en que se efectúa la actualización (vea [SAM13] para más información). El grueso del esfuerzo en actualización dinámica asume que no hay cambios de especificación y por lo tanto el mismo comportamiento es esperado (por ejemplo en [PH13]), o que la especificación es genérica (como en [SSH<sup>+</sup>05], [CYH<sup>+</sup>11], [ORH02], [KM07], [GJB96], [KM90]) y no proporcionada por el usuario. Ejemplos de este último, aparte de asegurar que la actualización no desemboca un fallo, asegura type safety (ej. [SHM08]) y data isolation entre versiones [SHB<sup>+</sup>07]. Quiescence [KM90] y nociones relacionadas (ej. [VEBD07], [AR09], [GJB96]) no requieren una representación explícita de las propiedades a ser preservadas, pero fue utilizado en conjunto con técnicas que garantizan consistencia genérica de semántica (ej. [BMPT10]).

La necesidad de una actualización por medio de la definición de propiedades por parte del usuario han sido reconocidas en [BG10]. En [HMH<sup>+</sup>12], se considera la especificación de propiedades de actualización, pero posee un enfoque en verificar si el programa satisface estas propiedades, en vez de automáticamente sintetizar la actualización para satisfacer estas propiedades.

Síntesis, estrategia operacional para construir un autómata que satisface una especificación dada, ha sido usado extensivamente para garantizar código que es correcto por construcción (por ejemplo en [GBC<sup>+</sup>13]). La naturalidad automática de la síntesis logra poder aplicar esta técnica no sólo en el momento en el que el diseño se lleva a cabo sino que también en tiempo de ejecución, generando un sistema adaptable. Dicha evolución no está limitada exclusivamente para sistemas adaptables como describimos en la etapa de introducción y motivación (ver sección 1.1). Por ejemplo, en [PTBP08] el problema de evolucionar componentes conjuntos es resuelto sintetizando “glue code” (por ejemplo,

controladores). Si bien la síntesis se ejecuta sin frenar el sistema, el nuevo controlador sólo puede reemplazar al viejo cuando el sistema entra en “quiescence”.

La síntesis requiere algún tipo de especificación desde la que, a través de diferentes técnicas de razonamiento, producir una solución. El resultado de sintetizar es correcto sólo en la medida en que la especificación es válida. Por lo tanto, técnicas de síntesis no son, en principio, resistentes a errores en la especificación o a ambientes que evolucionan y divergen de la especificación. El trabajo descrito en esta tesis es también susceptible a especificaciones invalidas. En el dominio de sistemas adaptables se han estudiado distintos enfoques que pueden detectar y resolver ciertas situaciones (por ejemplo en [DBK<sup>+</sup>14] y [VWMA11]) e incluso, aprender nuevas especificaciones en tiempo de ejecución. El enfoque descrito en este documento puede ser combinado con dichas técnicas.

Una suposición de este trabajo es que es posible controlar cambios del ambiente. En otras palabras, si la nueva especificación introduce diferentes características en el ambiente, como serían nuevos componentes, cambios en el protocolo de llamada de un componente, o deshabilitar un componente, es el controlador quien puede decidir cuando estos cambios ocurren. El controlador controla una infraestructura intermedia que puede cargar, descargar o cambiar componentes en tiempo de ejecución. Esto permite al controlador a planificar un cambio y darle más libertad para encontrar una estrategia que puede satisfacer el cambio requerido. Por otro lado, en muchas situaciones, puede suceder un cambio no anunciado en el ambiente y actualizar el controlador para acomodar este cambio es deseado.

Por ejemplo, si la conexión entre el UAV y la base puede perderse, y al suceder esto, la interfaz que el controlador de UAV maneja queda deshabilitada. En este caso, la actualización de este controlador debe producirse inmediatamente y puede ser imposible continuar garantizando los objetivos actuales o los nuevos. En [DBK<sup>+</sup>14] presentan un enfoque donde las garantías funcionales del controlador se degradan elegantemente para estos casos. Por otro lado, la técnica requiere que el controlador y la especificación del nivel de degradación preserve una relación de refinación entre el controlador actual y la especificación. Dicho requerimiento puede ser restrictivo.

El trabajo realizado en esta tesis, puede ser visto como una generalización de una de-

gradación progresiva presentada en [DBK<sup>+</sup>14]. Decimos esto, ya que la no controlabilidad del cambio de ambiente, puede en nuestro enfoque manejarse teniendo que la especificación de transición  $T$  requiera que el evento *startNewSpec* ocurra inmediatamente después de *beginUpdate*, sin embargo, el ambiente viejo  $E$  no es requerido para obtener refinamientos de  $E$  como en [DBK<sup>+</sup>14]

En este trabajo, limitamos la expresividad de nuestro enfoque a objetivos de *safety* (ver Definición 4.2.1). Esto hace a esta presentación simple e incluso permite una resolución de complejidad lineal al problema de control de actualización de controladores dinámicamente siempre y cuando el ambiente es determinístico. Sin embargo, podríamos permitir mayor expresividad en  $G$ ,  $G'$  y  $T$  sin incurrir del todo en la penalidad de resolver un problema de control (2EXPTIME-COMPLETE). Es posible, reformular la definición 4.2.1 para permitir que las especificaciones  $G$ ,  $G'$  y  $T$  puedan ser de la forma  $\Box\Diamond G$ ,  $\Box\Diamond G'$  y  $\Box\Diamond T$ . Este criterio de aceptación de Büchi extiende la expresividad manteniendo la complejidad en orden polinomial (esto lo conseguimos redefiniendo  $G_u$  a  $\Box\Diamond[(G \vee OldSpecDropped) \wedge (G' \vee \neg NewSpecEnsured) \wedge (\neg BeginUpdate \vee NewSpecEnsured) \wedge (\neg BeginUpdate \vee OldSpecDropped) \wedge T]$ . Donde *OldSpecDropped* es el flujo definido como  $\langle \{stopOldSpec\}, \emptyset, \perp \rangle$ , *NewSpecEnsured* es el flujo definido como  $\langle \{startNewSpec\}, \emptyset, \perp \rangle$  y *BeginUpdate* es el flujo definido como  $\langle \{beginUpdate\}, \emptyset, \perp \rangle$ . Se necesitan futuras investigaciones para analizar si, por ejemplo, las especificaciones SGR(1) [DBPU13] en el problema de control de actualización de controladores dinámicos, sólo puede ser resuelto en tiempo polinómico.

La complejidad lineal del problema de control de actualización de controladores dinámicamente cuando se restringe a propiedades de *safety* provee un argumento analítico para la escalabilidad. Sin embargo, queda por hacer una validación experimental, y en particular evaluar la necesidad práctica de introducir el no-determinismo, lo que puede producir que la complejidad se vaya a exponencial en peor caso. Los problemas de control en los casos de estudio descritos en la sección 5.1 fueron resueltos usando MTSA que provee algoritmos más complejos que los necesarios (polinomial para SGR(1) en vez de linear para formulas de obligación). Además, la estrategia para resolver el problema de actualización definida en esta tesis, que construye un controlador que controla las tres fases de actualización (ver Figura 4.1), prioriza el entendimiento de dicha técnica. Para reducir la complejidad (parti-

cularmente para el caso no lineal de fórmulas de Büchi), es posible producir un controlador de actualización composicionalmente, computando primero estados en el nuevo ambiente desde los cuales los nuevos objetivos pueden ser garantizados, y luego, computando un controlador que puede satisfacer alcanzar uno de esos estados luego de que *beginUpdate* ocurra, mientras se satisfacen los requerimientos de transición  $T$ .

Tenga en cuenta, que no discutimos hasta ahora que pasa si otra actualización dinámica es deseada luego de que se efectúe la primera. Una visión *naive* para concretar esto sería considerar a  $C_u$ , computado en la primera actualización, como controlador para la segunda. Esta idea sería ineficiente ya que  $C_u$  contiene, como lo muestra la Figura 4.2, el comportamiento entero del controlador original  $C$  más el control de la fase de transición y el control para la nueva especificación. Usar a  $C_u$  como  $C$  para la segunda actualización dinámica y aplicar la misma técnica tantas veces sea necesario, llevaría a obtener un controlador que posee el historial de todos los controladores y las actualizaciones realizadas en el sistema. Esto contrae problemas de escalabilidad. Esta situación puede ser evitada ya que una vez que la actualización se ha “estabilizado” con la nueva especificación (esto es, el controlador de actualización efectuó la acción *startNewSpec* y *stopOldSpec*), el controlador  $C_u$  garantiza no visitar estados que correspondan a la fase inicial (esto puede verse en el super-estado  $C_{G'}$  de la Figura 4.2). Por lo tanto, para la nueva actualización, en vez de considerar por completo todos los estados de  $C_u$ , es posible considerar la componente conexa de  $C_u$  que incluye al estado actual de  $C_u$  (lo que es a lo sumo, todos los estados que pertenecen al super-estado de  $C_{G'}$  de la Figura 4.2).

En la sección 5.1 comparamos nuestro enfoque con [GGM12] y [MGGB13] mediante casos de estudio. Como discutimos, una ventaja del trabajo de [GGM12] es que no le fue necesario especificar requerimientos de transición, requiriendo que la actualización sea equivalente a una actualización offline. La desventaja, es que si el sistema no puede retornar a un estado inicial y no exhibe comportamiento compatible con la nueva especificación desde el último estado inicial, entonces la actualización no podrá ser efectuada. El caso de estudio del UAV de la sección 5.1.3 ejemplificamos esto. También generalizamos la relación entre historias entre estados de la especificación vieja y de la nueva usadas en [GGM12] y [MGGB13], permitiendo al usuario especificar un mapeo de estados.

Finalizando, como discutimos en el capítulo 5 el criterio de actualización correcta de-

finido en [GGM12] puede ser expresado como un requerimiento de transición, pero hay una pequeña diferencia en los resultados obtenidos. Nuestro controlador de actualización forzará al sistema a que llegue al estado inicial, *garantizando* que la actualización va a ocurrir. Mientras que en [GGM12] hay una *asunción* de que el controlador actual y su ambiente van a ser cooperar a que el estado inicial eventualmente sea visitado, es decir, no garantiza que la actualización ocurra.

## 6.2. Conclusiones

Mostramos como resolver el problema de controladores actualizados dinámicamente para satisfacer una nueva especificación mediante el uso de síntesis de controladores. La solución propuesta garantiza satisfacer la nueva especificación y cualquier requerimiento de transición que sea dada por el usuario, pero también asegura que la actualización va a ocurrir tomando el control del sistema bajo la vieja especificación, y forzándolo a un estado seguro desde el cual la transición puede empezar para garantizar la nueva especificación eventualmente.

Se pueden realizar trabajos futuros cuyo enfoque sería mejorar la expresividad sin pagar el precio de una síntesis general. También, podríamos hacer la integración con otros enfoques que tienen como objetivo facilitar la capacidad de adaptación de alto nivel a los sistemas de software complejos. Esto lo lograríamos incluyendo técnicas para aprender el comportamiento del ambiente en tiempo de ejecución y adaptación cuantitativas para propiedades no-funcionales.



## Bibliografía

- [AR09] Austin Anderson and Julian Rathke. Migrating protocols in multi-threaded message-passing systems. In *Proceedings of the 2Nd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pages 8:1–8:5, New York, NY, USA, 2009. ACM.
- [BDP<sup>+</sup>13] Víctor Braberman, Nicolas D'Ippolito, Nir Piterman, Daniel Sykes, and Sebastian Uchitel. Controller synthesis: From modelling to enactment. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1347–1350, Piscataway, NJ, USA, 2013. IEEE Press.
- [BG10] Luciano Baresi and Carlo Ghezzi. The disappearing boundary between development-time and run-time. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 17–22, New York, NY, USA, 2010. ACM.
- [BMPT10] F. Banno, D. Marletta, G. Pappalardo, and E. Tramontana. Handling consistent dynamic updates on distributed systems. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*, pages 471–476, June 2010.
- [CYH<sup>+</sup>11] Haibo Chen, Jie Yu, Chengqun Hang, Binyu Zang, and Pen-Chung Yew. Dynamic software updating using a relaxed consistency model. *Software Engineering, IEEE Transactions on*, 37(5):679–694, Sept 2011.
- [DBK<sup>+</sup>14] Nicolas D'Ippolito, Víctor Braberman, Jeff Kramer, Jeff Magee, Daniel Sykes, and Sebastian Uchitel. Hope for the best, prepare for the worst: Multi-tier control for adaptive systems. In *ICSE 2014, Proceedings of the 36th International Conference on Software Engineering*, pages 688–699. ACM New York, 2014. ISBN: 978-1-4503-2756-5.
- [DBPU13] Nicolás D'Ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. Synthesizing nonanomalous event-based controllers for liveness goals. *ACM Trans. Softw. Eng. Methodol.*, 22(1):9:1–9:36, March 2013.

- [DFCU08] N. D’Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. Mtsa: The modal transition system analyser. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 475–476, Sept 2008.
- [EWS05] K. Etessami, T. Wilke, and R. Schuller. Fair simulation relations, parity games, and state space reduction for büchi automata. *SIAM Journal on Computing*, 34(5):1159–1175, 2005.
- [GBC<sup>+</sup>13] Joel Greenyer, Christian Brenner, Maxime Cordy, Patrick Heymans, and Erika Gressi. Incrementally synthesizing controllers from scenario-based product line specifications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, pages 433–443, New York, NY, USA, 2013. ACM.
- [GGM12] C. Ghezzi, J. Greenyer, and V.P.L. Manna. Synthesizing dynamically updating controllers from changes in scenario-based specifications. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 145–154, June 2012.
- [GJB96] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *Software Engineering, IEEE Transactions on*, 22(2):120–131, Feb 1996.
- [GM03] Dimitra Giannakopoulou and Jeff Magee. Fluent model checking for event-based systems. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-11*, pages 257–266, New York, NY, USA, 2003. ACM.
- [Ham14] Universität Hamburg. Yet another ardrone 2.0 framework. <http://vsiis-www.informatik.uni-hamburg.de/projects/yadrone>, Aug. 2014. [Online; accessed 2-Oct-2014].
- [HMH<sup>+</sup>12] Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. Specifying and verifying the correctness of dynamic soft-



- ware updates. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'12, pages 278–293, Berlin, Heidelberg, 2012. Springer-Verlag.
- [Jac95a] Michael Jackson. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.
- [Jac95b] Michael Jackson. The world and the machine. In *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, pages 283–283, April 1995.
- [JP06a] Sudeep Juvekar and Nir Piterman. Minimizing generalized büchi automata. In Thomas Ball and RobertB. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 45–58. Springer Berlin Heidelberg, 2006.
- [JP06b] Sudeep Juvekar and Nir Piterman. Minimizing generalized büchi automata. In Thomas Ball and RobertB. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 45–58. Springer Berlin Heidelberg, 2006.
- [Jur00] Marcin Jurdziński. Small progress measures for solving parity games. In Horst Reichel and Sophie Tison, editors, *STACS 2000*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer Berlin Heidelberg, 2000.
- [Kel76] Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, July 1976.
- [KM90] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *Software Engineering, IEEE Transactions on*, 16(11):1293–1306, Nov 1990.
- [KM07] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering, 2007. FOSE '07*, pages 259–268, May 2007.
- [KPR04] R. Kazhamiakin, M. Pistore, and M. Roveri. Formal verification of requirements using spin: a case study on web services. In *Software Engineering and*

- Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, pages 406–415, Sept 2004.
- [LL95] Claus Lewerentz and Thomas Lindner, editors. *Formal Development of Reactive Systems - Case Study Production Cell*, London, UK, UK, 1995. Springer-Verlag.
- [LvL02] Emmanuel Letier and Axel van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 83–93, New York, NY, USA, 2002. ACM.
- [MGGB13] V.P.L. Manna, Joel Greenyer, Carlo Ghezzi, and Christian Brenner. Formalizing correctness criteria of dynamic updates derived from specification changes. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '13, pages 63–72, Piscataway, NJ, USA, 2013. IEEE Press.
- [MK99] Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.
- [MKG97] J. Magee, J. Kramer, and D. Giannakopoulou. Analysing the behaviour of distributed software architectures: a case study. In *Distributed Computing Systems, 1997., Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of*, pages 240–245, Oct 1997.
- [MPS95] Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *STACS*, pages 229–242, 1995.
- [oP14] University of Paderborn. New rail technology paderborn. <http://www.railcab.de/>, Aug. 2014. [Online; accessed 1-Oct-2014].
- [ORH02] A Orso, A Rao, and M.J. Harrold. A technique for dynamic updating of java software. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 649–658, 2002.

- 
- [Par14] Parrot. Ardrone 2.0. <http://ardrone2.parrot.com/>, Aug. 2014. [Online; accessed 2-Oct-2014].
- [PH13] Cristian Cadar Petr Hosek. Safe software updates via multi-version execution. In *International Conference on Software Engineering (ICSE 2013)*, pages 612–621, 5 2013.
- [PM95] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25:41–61, 1995.
- [PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa’ar. Synthesis of reactive(1) designs. volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer Berlin Heidelberg, 2006.
- [PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’89, pages 179–190, New York, NY, USA, 1989. ACM.
- [PTBP08] Patrizio Pelliccione, Massimo Tivoli, Antonio Bucchiarone, and Andrea Polini. An architectural approach to the correct and automatic assembly of evolving component-based systems. *Journal of Systems and Software*, 81(12):2237 – 2251, 2008. Best papers from the 2007 Australian Software Engineering Conference (ASWEC 2007), Melbourne, Australia, April 10-13, 2007 Australian Software Engineering Conference 2007.
- [RW89] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, Jan 1989.
- [SAM13] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. A survey of dynamic software updating. *Journal of Software: Evolution and Process*, 25(5):535–568, 2013.
- [SEA14] *ICSE Symposium on Software Engineering for Adaptive and self-Managing Systems, SEAMS. ACM/IEEE*, 2006-2014.
- [SHB<sup>+</sup>07] Gareth Stoyale, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtii. Mutatis mutandis: Safe and predictable dynamic software updating, 2007.

- [SHM08] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates for java: A vm-centric approach, 2008.
- [SSH<sup>+</sup>05] Junrong Shen, Xi Sun, Gang Huang, Wenpin Jiao, Yanchun Sun, and Hong Mei. Towards a unified formal model for supporting mechanisms of dynamic component update. *SIGSOFT Softw. Eng. Notes*, 30(5):80–89, September 2005.
- [VEBD07] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D’Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *Software Engineering, IEEE Transactions on*, 33(12):856–868, Dec 2007.
- [VL01] Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE ’01, pages 249–, Washington, DC, USA, 2001. IEEE Computer Society.
- [vL09] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.
- [VLL00] A Van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *Software Engineering, IEEE Transactions on*, 26(10):978–1005, Oct 2000.
- [VWMA11] Pieter Vromant, Danny Weyns, Sam Malek, and Jesper Andersson. On interacting control loops in self-adaptive systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS ’11, pages 202–207, New York, NY, USA, 2011. ACM.
- [ZJ97] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6:1–30, 1997.