Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Hot-control. Una técnica para la generación y actualización automática de controladores discretos

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-CONTROL: UNA TÉCNICA PARA LA GENERACIÓN Y ACTUALIZACIÓN AUTOMÁTICA DE CONTROLADORES DISCRETOS

Es esperado que muchos sistemas corran 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. 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. Mediante la síntesis de controladores podremos obtener modelos de forma correcta debido a que son obtenidos mediante construcciones.

El enfoque de esta tesis es definir y plantear formalmente mediante síntesis de controladores el problema de la actualización dinámica, detallando un conjunto de inputs necesarios para la solución del mismo. A su vez, desarrollaremos varios casos de test utilizando la herramienta MTSA (Modal Transition System Analyser) dejando constancia de que los inputs definidos son suficientes para obtener el controlador buscado.

Palabras claves: *Ingeniería de requerimientos; Síntesis de controladores; Actualización dianámica; Especificación basada en eventos; MTSA framework; Concurrencia, LTS; Fluent; LTL.*

AGRADECIMIENTOS

Quiero aprovechar este espacio para agradecer

A mi familia con todo mi amor.

Índice general

1. INTRODUCCIÓN

1.1. Motivación

La operación continua, sistemas donde cada una de sus componentes se mantienen operativos, 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 [16] y más recientemente con el diseño de software adaptable. [1]

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** (*quiescense*) [16] y luego **tranquilidad** (*tranquility*) [29]. Muchas otras técnicas han sido desarrolladas (como en [2] y [8]) aunque estas nunca explican los requerimientos de actualización [3], ni indican cuando es correcto realizar el cambio a la nueva especificación. Para tal fin Ghezzi et al. [6, 21] estudió 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 [16], 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 [6] 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 ultimo estado inicial puede controlarse para satisfacer la nueva especificación. Por otra parte, en [21], 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 que podrían suceder con estos.

1.2. Resumen de la contribución

1.3. Esquema de tesis

2. FUNDAMENTOS TEÓRICOS

2.1. El Mundo y la Máquina

Los primeros conceptos que detallaré estarán involucrados con la ingeniería de los requerimientos. Los puntos de vista mas relevantes son los de Zave y Jackson ([30, 9, 10]) por un lado, y los de Letier y Van Lamsweerde ([28, 26]) 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 soluciona 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 (véase la Fig. 2.1).

Así mismo, es esperado que la máquina proponga una solución al problema. Por ejemplo, en la Fig. 2.1 podemos ver que la célula de producción debe procesar cada produc-

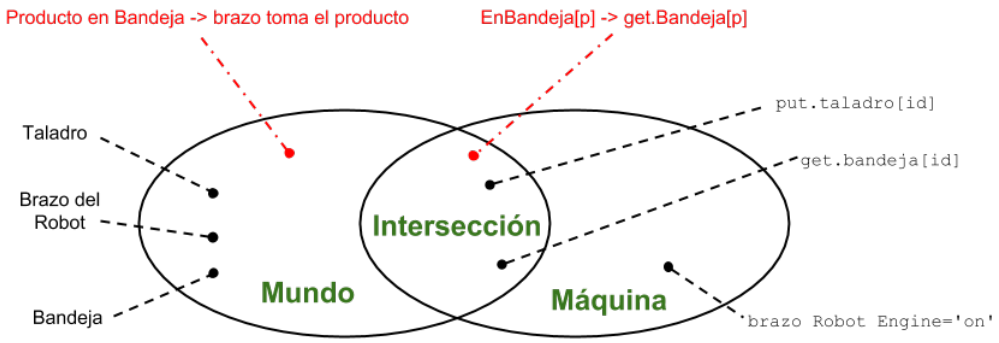

Fig. 2.1: El Mundo y la Máquina

to, solo si están disponible en la bandeja de entrada en ese momento. Con la sentencia $EnBandeja[p] \rightarrow get.Bandeja[p]$ muestro que se espera que el brazo del robot solo 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* [22, 9]. Además, estas pueden estar en modo *indicativo* u *optativo*. En otros trabajos, como en [27], 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 ocaciones, 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 [26, 28] 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 una de ellas esta etiquetada con una acción. El conjunto de todas las acciones que posee un LTS es llamado alfabeto.

Definición 2.2.1. *(Sistema de Transición Etiquetado)[15] 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* $S_E \subseteq States$ *es un conjunto finito de estados,* $A_E \subseteq Act$ *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 ℓ 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 αE.

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(||) es un operador simétrico tal que* $E||M$ *es el LTS definido de la siguiente manera* $E||M = (S_E \times S_M, A_E \cup A_M, \Delta, (s_{E_0}, s_{M_0}))$*, donde* Δ *es la relación mas pequeña que satisface las siguientes reglas, donde* $\ell \in A_E \cup A_M$*:*

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

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

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 $(s_E, s_M) \in E||M$ *sucede lo siguiente:* $\Delta_{E||M}((s_E, s_M)) \cap A_{E_u} = \Delta_E(S_E) \cap A_{E_u}$.

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 $(s_E, s_M) \in S_{S||M}$ se cumple que, una acción $\ell \in A_{E_u}$ es deshabilitada en (s_E, s_M) si y solo 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. *(Tranzas) Sea un LTS* $E = (S, A, \Delta, s_0)$. *Una secuencia* $\pi = \ell_0, \ell_1, ...$ *es una traza en* E *si existe una secuencia* $s_0, \ell_0, s_1, \ell_1, ...$ *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* $s_0, \ell_0, s_1, \ell_1, ...$ *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* $Reach(E)$.

En el transcurso de esta tesis, vamos a estudiar solo 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 [14, 7, 28, 17]. 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 [7]. Fluent Linear Temporal Logic (FLTL) [7] 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: $Fl =< I_{Fl}, T_{Fl}, Init_{Fl} >$, 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 $\ell =< \ell, Act\ \{\ell\}, false >$. 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 todas las posibles flujos sobre Act. Una formula FLTL esta se define inductivamente utilizando los conectores booleanos estandar y operadores temporales como el **X** (próximo), **U** (antes fuerte) de la siguiente manera:

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

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

- $Init_{Fl} \wedge (\forall j \in \mathbb{N} : 0 \leq j \leq i \rightarrow \ell_k \notin T_{Fl})$
- $\exists j \in \mathbb{N} : (j \leq i \wedge \ell_j \in I_{Fl}) \wedge (\forall k \in \mathbb{N} : j < k \leq i \rightarrow \ell_k \notin T_{Fl})$

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

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

Diremos que φ se cumple en π, denotado como $\pi \vDash \varphi$, si $\pi, 0 \vDash \varphi$. Una fórmula $\varphi \in$ FLTL es cierta si un LTS E (denotado como $E \vDash \varphi$) 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 adecuando 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 una LTS que restringe solamente la ocurrencia de acciones controlables y garantiza que la composición paralela del ambiente con la LTS recién descripta 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 $\mathcal{E} =< E, H, A_c >$ 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 π en $E||M$ se cumple que si $\pi \vDash A_{s_i}$ entonces $\pi \vDash G_i$.*

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 $\square\ \rho$, las asunciones A_s son una conjunción de sub-fórmulas FLTL de la pinta $\square \Diamond\ \phi$, y el objetivo G una conjunción de sub-fórmulas FLTL de la pinta $\square \Diamond\ \gamma$ donde ρ, ϕ y γ son combinación booleana de flujos.

Definición 2.4.2. *(Control LTS SGR(1)) un problema de control LTS $\mathcal{E} =< E, H, A_C >$*

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* ϕ_i, ρ *y* γ_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) elije, del conjunto de acciones controlables, cual habilitar y el jugador 2 (el ambiente) elije que acciones tomar libremente. Formalmente podemos definir lo siguiente.

Definición 2.5.1. *(Juego de dos jugadores) Un juego de dos jugadores es* $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi)$, *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* Γ^+. *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* $p = s_{g_0}, s_{g_1}, ...$ *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 γ el conjunto de opciones del controlador es vacía, esto puede llevar a un deadlock. Esto será considerado como prohibido mas 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 Ω 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 [23]. Piterman et al. define 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 Ω para el controlador es un par de funciones (σ, u), donde Ω es una memoria que tiene designado como valor inicial $\omega_0, \sigma : \Omega \times S \to 2^S$ tal que $\sigma(\omega, s) \subseteq \Gamma^+(s)$ y $u : \Omega \times S \to \Omega$.*

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

Definición 2.5.3. *(Consistencia y estrategia ganadora) una jugada finita o infinita $p = s_0, s_1, ...$ es consistente con (ω, 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 (σ, u) para el controlador desde el estado s es ganadora si cada jugada maximal empezando de s y consistente con (σ, u) es infinita y en φ. Diremos que el controlador gana el juego G si tiene una estrategia ganadora desde el estado inicial.*

Diremos que chequear 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 [20, 25]. Resultados estudiados anteriormente [24], demuestran que si un controlador gana el juego G y φ es ω-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). [23]

La construcción de la máquina para un problema de control LTS SGR(1) esta divido 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 (σ, 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 solo si, existe una solución al juego GR(1) G. Luego, podremos afirmar que el controlador LTS M creada a partir de (σ, 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) $\mathcal{E} =< E, H, A_C >$ 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.

Mas 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 $fl = \{\dot{1}, ..., \}$ un conjunto de flujos usados en A_s y en G donde $=< I_i, T_i, Init_i >$. 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 φ. Formalmente, tenemos que $S_g = S_e \times \prod_{i=1}^{k} \{true, false\}$. Consideramos un estado $s_g = (s_e, \alpha_1, ..., \alpha_k)$. Dado un flujo fl_i, diremos que s_g satisface fl_i si α_i es $true$ y s_g no satisface fl_i si no.

Además, definiremos las relaciones Γ^- y Γ^+ aplicando las siguientes reglas. Sea $s_g = (s_e, \alpha_1, ..., \alpha_k)$. Si s_g no satisface ρ (es decir, s_g es no seguro) no agregaremos los sucesores a s_g. Si s_g satisface ρ, por cada transición $(s_e, l, s'_e) \in \Delta_e$ agregaremos $(s_g, (s'_e, \alpha'_1, ..., \alpha'_k))$ en Γ^β, donde β y α'_i cumplen las siguiente condiciones:

β	α'_i
es +: si $l \in A_C$,	es α_i: si $l \notin I_{fl_i} \cup T_{fl_i}$,
es -: si $l \notin A_C$.	es $true$: si $l \in I_{fl_i}$ o
	es $false$: si $l \in T_{fl_i}$.

El estado inicial s_{g_0} es $(s_{e_0}, initially_1, ..., initially_k)$.

Por último, construiremos la *condición de ganada* φ_g, definida como un conjunto infinito

de trazas, para A_S y G de la siguiente manera: abusando de la notación denotaremos ϕ_i al conjunto de estados s_g tales que s_g satisface las asunciones ϕ_i y a γ_i al conjunto de secuencias que satisfacen $gr((\phi_1, .., \phi_n), (\gamma_1, ..., \gamma_m))$. 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* φ_g del juego GR(1), pero si se traducen a un problema de *deadlock avoidance* a la hora de construir Γ^- y Γ^+. De esta manera, la *condicion de ganada* es $\square\ \rho \wedge (\bigwedge_{i=1}^{n} \square\Diamond\phi_i \Rightarrow \bigwedge_{j=1}^{m} \square\Diamond\omega_j)$.

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) $\mathcal{E} =< E, H, A_C >$, 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, construimos $M = (S_M, A, \Delta_M, s_{M_0})$ 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.

Mas formalmente, sea $E = (S_e, A, \Delta_e, s_{e_0})$, $fl = \{fl_1, ..., fl_k\}$ el conjunto de flujos que aparecen en φ, $G = (S_g, \Gamma^-, \Gamma^+, s_{g_0}, \varphi_g)$ el juego GR(1) construido a partir de E como explicamos anteriormente, y sea $\sigma : \Omega \times S_g \to 2^{S_g}$ y $u : \Omega \times S_g \to \Omega$ 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 $s_g = (s_e, \alpha_1, ..., \alpha_k)$ y $s'_g = (s'_e, \alpha'_1, ..., \alpha'_k)$. Decimos que esa acción ℓ es *posible* desde s_g hacia s'_g si:

1. $(s_g, s'_g) \in \Gamma^- \cup \Gamma^+$,
2. existe una acción ℓ 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 Γ^- existe una acción $\ell \notin A_C$ tal que ℓ es posible desde s_g hacia s'_g. Si

$s'_g \in \sigma(\omega, s_g)$ entonces para cada acción ℓ tal que ℓ es posible desde s_g hacia s'_g agregamos $((\omega, s_g), \ell, (u(\omega, s_g), s'_g))$ hacia Δ_M. De forma similar, consideramos una transición $(s_g, s'_g) \in \Gamma^+$. Por definición de Γ^+ existe una acción $\ell \in A_C$ tal que ℓ es posible desde s_g hacia s'_g. Si $s'_g \in \sigma(\omega, s_g)$ entonces para cada acción ℓ tal que ℓ es posible desde s_g hacia s'_g agregamos $((\omega, s_g), \ell, (u(\omega, s_g), s'_g))$ hacia Δ_M.

El estado inicial de M esta definido como $s_{M_0} = (\omega_0, s_{g_0})$ donde ω_0 es el valor inicial de la memoria Ω. De esta forma completamos la definición de M.

2.6.3. Algoritmo

En esta sección, presentaremos el algoritmo implementado en la herramienta MTSA [4] el cual está basado en las ideas de Juvekar y Piterman [12].

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 tranzas 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 [11]. 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* $\varphi = gr((\phi_1, ..., \phi_n), (\gamma_1, ..., \gamma_m))$. *Una función de ranking para un objetivo* γ_j *es una función* $R_j : S_g \to (\mathbb{N} \times \{1, ..., n\} \cup \{\infty\})$. *Intuitivamente,* $R_j(s_g) = (k, \ell)$ *significa que para alcanzar desde* s_g *a un estado en el cual* γ_j *vale, todos los caminos satisfacen la asunción* ϕ_ℓ *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 satisface infinitamente las asunciones, pero no satisface infinitamente a todos los objetivos.

Algorithm 1 para resolver juegos SGR(1)

```
1:  procedure SOLVEGAME(GAME=(STATES,TRANSITIONS),SAFE,GUARANTEES,ASSUMPTIONS)
2:      Inicialización:
3:      for state : states do
4:          for g : guarantees do
5:              rank_g(state)← (0,1)
6:      Queue pending
7:      for state : states do
8:          if ∃ g : guarantees / state ∉ g ∧ state ∈ assume_1 then
9:              pending.push(pair(state,g))
10:         if Γ⁻(state) = ∅ ∧ Γ⁺(state) = ∅ then
11:             for g : guarantees do
12:                 rank_g(state) ← ∞
13:                 pending.push(unstable_pred(state,g))
14:     Estabilización:
15:     while !pending.empty() do
16:         (state,g) ← pending.pop()
17:         if rank_g(state) = ∞ then
18:             continue
19:         if isStable(rank_g(state)) then
20:             continue
21:         rank_g(state)← inc(best(state,g),state,g)
22:         pending.push(unstable_pred(state,g))
```

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 ∞. De esta manera, los estados cuyos rankings son ∞ son aquellos donde no se satisface ρ 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-esima 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 $sr : S_g \to (\mathbb{N} \times \{1, ..., n\} \cup \{\infty\})$. Esta función también codifica el hecho de que los estados de deadlock tienen ranking ∞. Además, notemos que usa un orden lexicográfico para los objetivos. Dado un estado s_g y un objetivo γ_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 $sr(s_g, j) = min_{s'_g \in \Gamma^+(s_g)} R_{j\oplus 1}(S'_g)$.
- 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, `inc((k,`ℓ`),state,g)` devuelve $(0, 1)$ si `state` está en γ_g, devuelve (k, ℓ) si `state` no está en $assumption_\ell$, y devuelve el mínimo valor mayor que (k, ℓ) en el resto de los casos. Notemos que `inc(`∞`,state,g)` es ∞, y si $n = max_\ell(|\phi_\ell - (\gamma_g)|)$ y `state` está en $\phi_m - \gamma_g$ entonces `inc((n,m),state,g)` es ∞. Este algoritmo calcula el mínimo ranking estable. Basados en ideas del mundo de autómatas de büchi [5, 13], 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. [19, 18]

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 $lts(E)$ 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`** 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,

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

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

||COOLING_TOWER = (MAINTENANCE||COOLER).
```

Fig. 2.2: Ejemplo FSP

dados dos procesos FSP `P` y `Q`, tenemos: `lts(P||Q) = lts(P)||lts(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 dijimos 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

4. PROBLEMA DE ACTUALIZACIÓN DE CONTROLADORES DINÁMICAMENTE

5. VALIDANDO LOS ALGORITMOS

5.1. Casos de Estudio

5.2. Resultados

6. CONCLUSIONES Y TRABAJOS FUTUROS

6.1. Conclusiones

6.2. Trabajo futuro

Bibliografía

[1] *ICSE Symposium on Software Engineering for Adaptive and self-Managing Systems, SEAMS. ACM/IEEE*, 2006-2014.

[2] A. Anderson and J. 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.

[3] L. Baresi and C. 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.

[4] 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.

[5] 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.

[6] C. Ghezzi, J. Greenyer, and V. 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.

[7] D. Giannakopoulou and J. 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.

[8] 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.

[9] M. Jackson. *Software Requirements &Amp; Specifications: A Lexicon of Practice,*

Principles and Prejudices. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[10] M. Jackson. The world and the machine. In *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, pages 283–283, April 1995.

[11] M. Jurdziński. Small progress measures for solving parity games. In H. Reichel and S. Tison, editors, *STACS 2000*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer Berlin Heidelberg, 2000.

[12] S. Juvekar and N. Piterman. Minimizing generalized büchi automata. In T. Ball and R. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 45–58. Springer Berlin Heidelberg, 2006.

[13] S. Juvekar and N. Piterman. Minimizing generalized büchi automata. In T. Ball and R. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 45–58. Springer Berlin Heidelberg, 2006.

[14] 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.

[15] R. M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, July 1976.

[16] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *Software Engineering, IEEE Transactions on*, 16(11):1293–1306, Nov 1990.

[17] E. Letier and A. 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.

[18] J. Magee and J. Kramer. *Concurrency: State Models &Amp; Java Programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[19] J. Magee, J. Kramer, and D. Giannakopoulou. Analysing the behaviour of distributed software architectures: a case study. In *Distributed Computing Systems, 1997., Pro-*

ceedings of the Sixth IEEE Computer Society Workshop on Future Trends of, pages 240–245, Oct 1997.

[20] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *STACS*, pages 229–242, 1995.

[21] V. Panzica La Manna, J. Greenyer, C. Ghezzi, and C. 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.

[22] D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25:41–61, 1995.

[23] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer Berlin Heidelberg, 2006.

[24] 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.

[25] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, Jan 1989.

[26] A. 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.

[27] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[28] A. Van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *Software Engineering, IEEE Transactions on*, 26(10):978–1005, Oct 2000.

[29] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disrupti-

ve alternative to quiescence for ensuring safe dynamic updates. *Software Engineering, IEEE Transactions on*, 33(12):856–868, Dec 2007.

[30] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6:1–30, 1997.