Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Ejecución de controladores discretos sintetizados a partir de una especificación de alto nivel para un robot modelo N6

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

Mariano J. Cerrutti

Director: Nicolás D'Ippolito

Buenos Aires, 2014

EJECUCIÓN DE CONTROLADORES DISCRETOS SINTETIZADOS A PARTIR DE UNA ESPECIFICACIÓN DE ALTO NIVEL PARA UN ROBOT MODELO N6

El objetivo de esta tesis es evaluar la adecuación de las técnicas de control instrumentadas en la herramienta de modelado, verificación y síntesis MTSA para un entorno industrial que hace uso de un robot N6 satisfaciendo el suministro de bienes entre los puestos de trabajo de una planta.
El problema se expresa con máquinas de transiciones etiquetadas (LTS) para cada uno de los componentes involucrados y las propiedades son expresadas en un subconjunto de las lógicas temporales lineales que caracterizan juegos de reactividad generalizada con propiedades de seguridad (SGR(1)) sobre dominios falibles.
El robot utilizado es un modelo N6 producido por la empresa RobotGroup en Argentina con fines didácticos.
Durante las fases de implementación, adaptación y ejecución en el entorno de MTSA se analiza la calidad de los controladores sintetizados, la respuesta ante cambios de configuración en el sistema y los patrones emergentes que podrían motivar guías de trabajo para problemas similares.

Palabras claves: Síntesis, Control, Cambio de Configuración, Robot planar, LTS, FLTL.

ENACTMENT OF A DISCRETE CONTROLLER SYNTHETIZED FROM A HIGH LEVEL SPECIFICATION TO BE USED WITH A N6 ROBOT

The objective of this thesis is to evalute de adequacy of the control techniques employed by the MTSA, the modelling, synthesis and verification tool. The case study is done on an industrial environment where a N6 robot is supposed to be taking items from one workstation to the other inside the facility.
The problem is expressed through labeled transition systems (LTS) describing each of the components comprising the environment. System properties are expressed with FLTL formulas.
The robot is a N6 model made by RobotGroup in Argentina for didactic purposes.
During implementation, adaptation and enactment within MTSA the quality of the synthetized controllers is evaluated, as well as the response to configuration changes. Emerging patterns are identified as they can provide guidelines for future work o similar domains.

Palabras claves: Controller Synthesis, Change of Configuration, Planar Robot, LTS, FLTL.

AGRADECIMIENTOS

Quisiera agradecer principalmente al grupo de gente que forma el laboratorio LaFHIS por darme la oportunidad de acceder al mundo académico de una forma que no hubiese esperado. Cabe nombrar a los directores del laboratorio Víctor Braberman y Sebastián Uchitel por el trabajo que realizan sin pausa para edificar este espacio del que he tenido el privilegio de participar. Reconocer especialmente el esfuerzo y el apoyo de Nicolás D'Ippolito como director de esta tesis, colega de trabajo y amigo. Al enorme grupo de personas que trabajan para permitir el acceso público a la educación, sin las cuales este trabajo y mi recorrido por la universidad hubiera sido imposible.
Agradezco también a mis padres y a mis hermanos la ayuda económica, logística y emocional que ha sido necesaria en este último tramo, a mis amigos por ayudarme a encontrar el placer de vivir y a aquellas personas que fueron mi soporte afectivo durante el curso de mi educación de grado.

A los dragones de la probabilidad.

Índice general

1. INTRODUCCIÓN

La producción automática de programas a partir de especificaciones declarativas es una prometedora alternativa a la escritura manual, la técnica empleada en esta tesis goza de al menos dos propiedades deseables: la conformidad con propiedades de progreso y seguridad por construcción y la adecuación a cambios de configuración en el sistema. Las propiedades se expresan como fórmulas de lógica temporal lineal (LTL). Este subconjunto de las lógicas temporales es un acompañante natural de los modelos basados en eventos. Se ha utilizado con éxito en repetidas ocasiones y resulta eficiente para expresar gran parte de las propiedades que se espera sean satisfechas por un sistema. En nuestro caso expresamos dos tipos de propiedades, las de seguridad (safety) en las que se espera que una propiedad valga siempre durante la ejecución del sistema, y las de progreso (liveness) en las que, sí se cumplen un conjunto de presunciones con infinita frecuencia, deberán cumplirse un conjunto de objetivos con infinita frecuencia. Un ejemplo de ésto puede ser un par de propiedades que expresen que una prensa no debe liberarse si otra herramienta está trabajando sobre la pieza prensada (safety) y que el sistema permita procesar piezas de forma contínua, por ejemplo tomando de una bandeja de entrada, prensando la pieza, perforándola y colocándola en una bandeja de salida (liveness). Uno esperaría que a partir de la presunción de que en la bandeja se colocarán piezas crudas con infinita frecuencia, piezas procesadas serán colocadas en la bandeja de salida con infinita frecuencia.

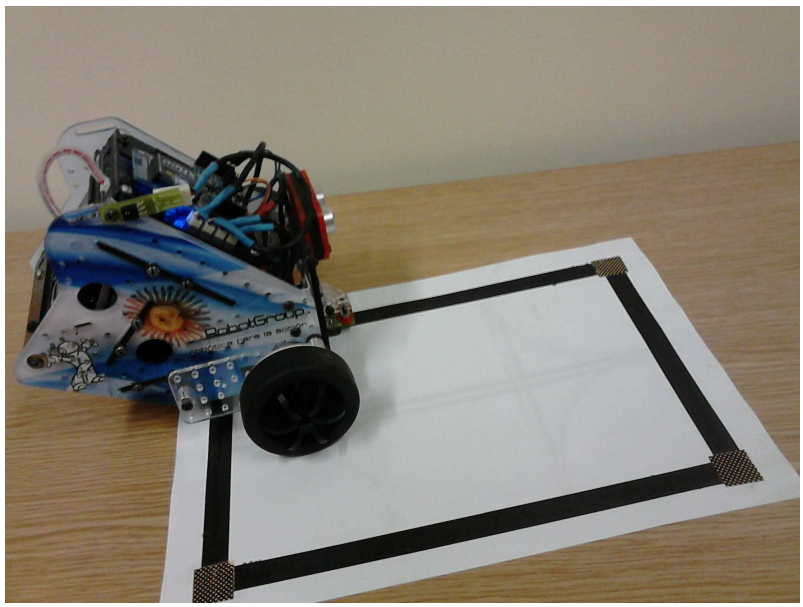


Fig. 1.1: El robot N6 en un mapa rectangular

Los programas conseguidos con ésta técnica podrían ser adecuados para procesos industriales cuya criticidad puede expresarse en el dominio de las lógicas temporales lineales, que es descripto con juegos de reactividad generalizada con propiedades de seguridad (SGR(1)) sobre dominios falibles.

En lo que respecta al cambio entre una configuración y otra, como parte natural de un proceso en evolución o que se encuentra en pleno refinamiento, al mantenerse en el plano declarativo la técnica de producción automática de programas permite adaptarse rápidamente al cambio sin la necesidad de realizar un trabajo manual por debajo de la especificación, manteniendo las propiedades expresadas inicialmente. Incluso si se refinan las propiedades de seguridad o progreso la técnica produce un programa que satisface el comportamiento esperado, en el caso de que tal programa exista.
En la fabricación de productos o provisión de servicios a través de procesos industriales, la calidad asociada al producto o al servicio se relaciona con la calidad de cada una de las operaciones realizadas y a la calidad del proceso per se.
Se podría pensar en una segunda función que evalúe la calidad del producto a partir de la calidad de las operaciones involucradas en el proceso, la calidad del proceso en sí y la confiabilidad de las técnicas de medición de las operaciones y el proceso. Como ejemplo se supone que al fijar dos piezas, un ajuste realizado con una herramienta manual y validado por una llave de corte es menos preciso que una herramienta automática con la capacidad de sensado suficiente para reportar valores de torque y ángulo con alguna cota de error. El caso es que desde las operaciones más básicas hasta los procesos más generales que pueden incluso exceder a la ingeniería de procesos industriales (e.g. la ingeniería de producto que cruza disciplinas como marketing, producción, distribución y venta), las medidas de calidad se propagan a través de un mecanismo de presunciones y garantías.
En este trabajo se van a utilizar técnicas de control relacionadas con juegos de reactividad generalizada en los que se pueden expresar propiedades de seguridad y progreso. Las propiedades de progreso están vinculadas con la práctica de la ingeniería de procesos en las que el ingeniero es responsable de diseñar una estrategia de producción que permita transformar los bienes entrantes en un bien compuesto con cierto valor agregado. Uno podría expresar una propiedad de progreso en la que exige que un producto sea entregado con infinita frecuencia como resultado de la aplicacion de operaciones atómicas. Las propiedades de seguridad describen situaciones que el proceso debe evitar en todo momento. Permiten verificar que un proceso cumple con ciertas garantías de seguridad. En nuestro caso las estrategias obtenidas con la técnica de síntesis satisfacen ambos tipos de propiedades por construcción.
Con las técnicas utilizadas en este trabajo no se declaran propiedades cuantitativas relacionadas, por ejemplo, con el tiempo y el costo.
En los casos de estudio presentados se emplean robots autónomos en un entorno industrial para el que permiten cumplir los requerimientos funcionales satisfaciendo alguna medida de calidad. Su comportamiento en planta puede expresarse como un sistema que involucra a otros agentes, debe satisfacer las propiedades mencionadas y puede ser sintetizado automáticamente. Estas estrategias o comportamientos son los responsables de coordinar la interacción de las partes para asegurar que el sistema una vez puesto en ejecución y en su totalidad (entendida como composición de partes) cumpla con el comportamiento esperado.

El cambio de requerimientos es una actividad aceptada y practicada en el entorno de aplicaciones industriales, inmanente a los procesos de producción bajo demanda, provisión y servicios. En el caso de la producción es común introducir un nuevo modelo de producto, o una variación de un producto existente e intentar satisfacer los nuevos requerimientos con las herramientas instaladas en la planta, o introducir una nueva ruta en el caso de la provisión y replanificar el suministro satisfaciendo las necesidades existentes sin modificar las

instalaciones. En un ambiente de éstas características (o incluso en un caso de producción bajo demanda, donde las especificaciones se modifican en base a una orden de producción emergente) la utilización de técnicas de síntesis automática cuentan con al menos dos propiedades deseables; permite reconstruir un controlador que es correcto por definición en conformidad con las propiedades expresadas para el sistema en su totalidad, y permite a la vez reconstruir un controlador que se adapte a los cambios en la descripción de los procesos o de las propiedades.
El objetivo principal de este trabajo es demostrar la aplicabilidad de las técnicas mencionadas en un entorno que hace uso de un robot autónomo modelo N6 para satisfacer las necesidades de distribución de suministros en una línea de producción industrial. Se mostrará así también la forma en que un cambio en los requerimientos o la descripción del entorno son satisfechos sin necesidad de escribir otro controlador en forma manual.
Se presentan los fundamentos teóricos, los detalles de implementación y la extensión de las herramientas existentes para permitir la puesta en marcha del controlador y los escenarios de prueba.
El modelo presentado goza de características interesantes porque permite desplazar el problema sobre varios ejes de cambio. Estos ejes pueden ser el tamaño del mapa, la cantidad de bandejas, la capacidad de carga, la cantidad de ítems a desplazar, las especificaciones de comportamiento del robot y la aparición de obstáculos en las vías que comunican los puestos.

1.1. Motivación

Puede entenderse que el resultado de la síntesis a la que se ha hecho referencia al hablar de los juegos SGR(1) es un programa, pero en verdad vale utilizar el término controlador por resultar más abarcativo y acertado. En particular uno cuyo comportamiento se expresa como un sistema de transiciones etiquetadas, donde el conjunto de transiciones se particiona entre aquellas controlables (operaciones o, en el contexto de este trabajo, mensajes de control) y aquellas no controlables (observaciones o, en el contexto de este trabajo, mensajes de estado). Esto puede traducirse en un programa, en un nodo de control que se comunica con otros componentes de naturaleza diversa, un diseño de lógica digital, una estrategia para realizar operaciones manuales o cualquier proceso basado en eventos que distinga entre acciones controlables y no controlables.
La interacción de diversos agentes en una planta industrial puede llevar fácilmente a situaciones indeseadas. Esto puede deberse a que el sistema permitió llegar a una situación riesgosa para las personas o los equipos en planta, o porque no se consigue satisfacer un objetivo de progreso. Estas situaciones demandan un análisis de las causas de error, una corrección y una puesta a prueba de un sistema que podría no estar cumpliendo (aún) los objetivos deseados.
Se presentan situaciones similares cuando se produce un cambio en alguna dimensión del sistema, ya sea en la especificación de los procesos participantes o de los objetivos a cumplir. Se repite el ciclo de análisis, corrección y puesta a prueba.
Se analiza como subdominio de lo anterior un sistema en el que un robot planar móvil transporta carga a través de una planta industrial, ya sea una planta de montaje donde se transportan piezas entre una línea y la otra, o un depósito en el que ha de reubicarse stock. En ambos casos se buscan satisfacer objetivos expresados en términos de seguridad y progreso (sobre la presunción de que los componentes se comportan conforme a una repre-

sentación en la forma de un proceso de estados finitos).
Estos requerimientos permiten probar la integración de las técnicas de modelado y síntesis provistas dentro de la herramienta MTSA, como también validar el funcionamiento de las extensiones necesarias para la ejecución de un controlador (`enactment`) en el caso en que el mismo opera sobre un conjunto de dispositivos físicos.
El comportamiento esperado del sistema debe ser expresado en términos de suposiciones, objetivos y un modelo operacional de las máquinas involucradas, de forma que al conseguir el controlador y componerlo con éstas satisfaga el comportamiento esperado. Para cada máquina que se encuentra representando a un sistema externo se debe proveer un mecanismo de interacción que permita relacionarlo con la representación interna en tiempo de ejecución (que en el contexto de este trabajo será referido como componente de ejecución o enactor).
En este trabajo se valida este enfoque construyendo un entorno apropiado a partir de las herramientas existentes y las extensiones relacionadas con ejecución (enactment).

1.2. Presentación del caso de estudio

Como se ha dicho anteriormente el caso de estudio trata de un robot que transporta carga entre puestos de trabajo en una planta industrial. El robot se desplaza entre un puesto y otro siguiendo una línea negra pintada en el piso, en cada intersección o punto de detención se reemplaza la línea negra por un área cuadrada pintada con un patrón que en promedio puede leerse como un valor intermedio o gris.

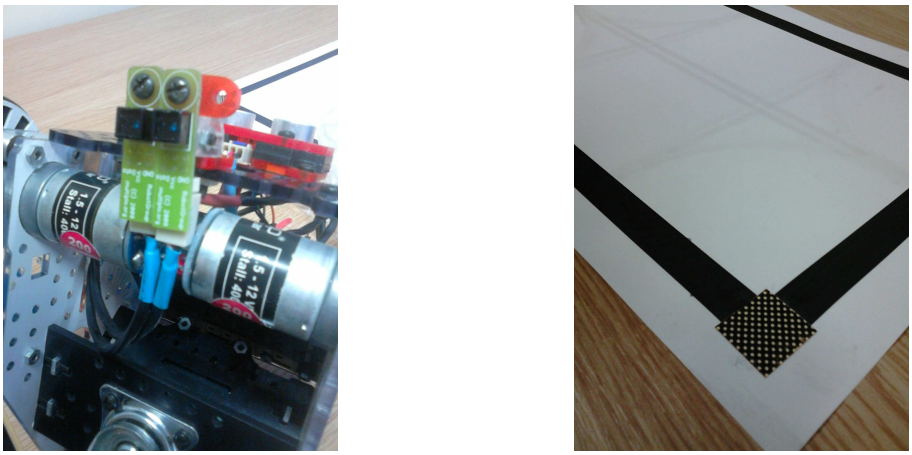


Fig. 1.2: Sensores infrarrojos y microrreductores del robot modelo N6 (izq.) y nodo de intersección o detención en un mapa rectangular (der.)

Los puntos de partida y llegada de la carga son representados como bandejas, pueden ser de salida o de entrada. Para este trabajo se supone que la carga o descarga puede realizarse en un punto particular de la línea cuando el robot se encuentra en una de cuatro posiciones (con el frente hacia el norte, este, sur u oeste). Las bandejas de entrada se representan como varillas rojas en la superficie de desplazamiento y las de salida como varillas negras en la superficie. La carga se representa con tachuelas.

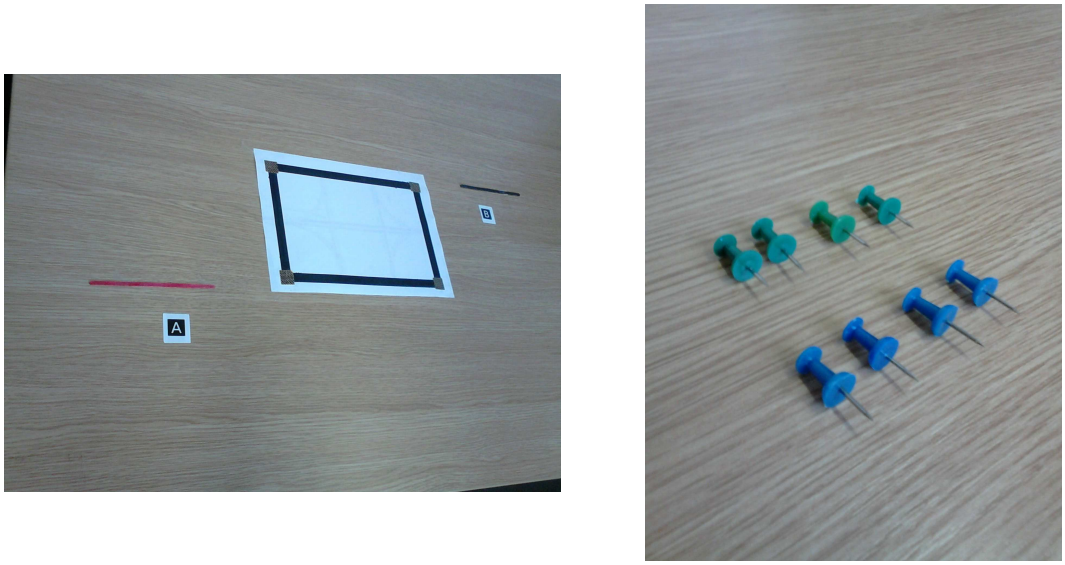


Fig. 1.3: Mapa rectangular con bandejas de entrada roja y salida negra (izq.) y carga representada con tachuelas (der.)

En las líneas de montaje de una planta industrial el robot suele estar compuesto por un pequeño mecanismo de control, actuadores (generalmente neumáticos) y un carro para transportar la carga. Para representar el carro de carga se ha adherido al revés del robot un anillo de goma donde se pueden fijar las tachuelas que representan los ítems a transportar.

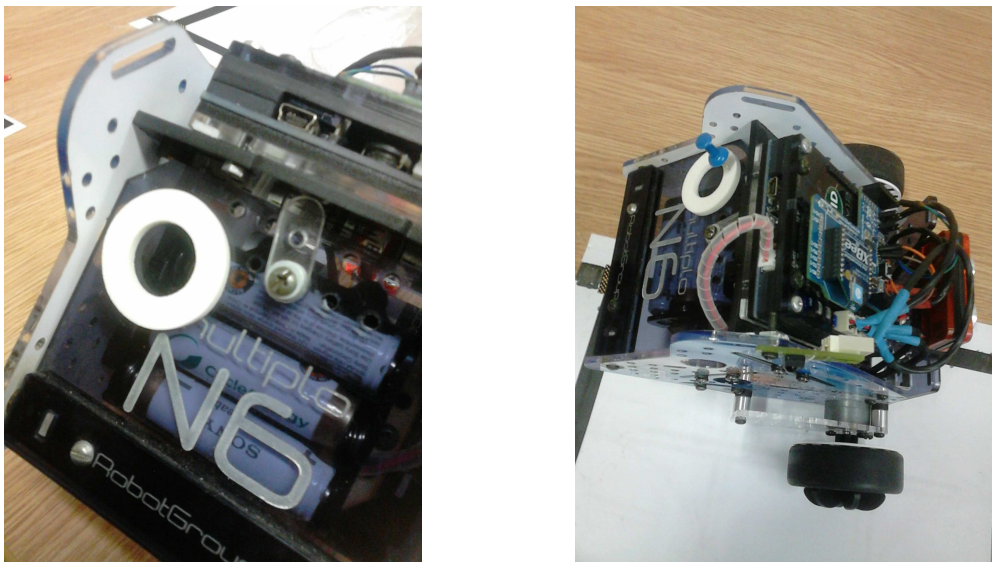


Fig. 1.4: Anillo de goma representando el carro de carga (izq.) y el robot modelo N6 llevando carga (der.)

Para las pruebas que involucran cambios de configuración se representará un obstáculo en una de las vías que comunican un puesto con otro utilizando una pequeña pieza de goma. Por otra parte, dado que se trabaja sobre un dominio falible, donde se expresa la posibilidad que el robot pierda rastro de la línea negra por errores de lectura, desgaste de la pista, cambios de iluminación o errores emergentes, se introduce un componente que representa a

un mecanismo de reubicación. El robot tiene capacidad suficiente para informar cuando ha perdido el rastro de la línea, en cuyo caso se ejecuta este mecanismo de reubicación (en este escenario manual), que lleva al robot a una posición y orientación conocida, permitiendo continuar la ejecución de la estrategia desde ese punto.
Se espera que el controlador pueda describir un proceso que obligue al robot a desplazarse a una bandeja de entrada en tanto haya carga, la tome, la lleve a destino y la descargue con infinita frecuencia, aún sobre un dominio que tolera faltas no sistemáticas.

1.3. Resumen de contribuciones

La mayor contribución de esta tesis sea probablemente la de explorar un caso de estudio horizontal que involucra no sólo el consumo de la teoría subyacente a la herramienta de modelado y síntesis, sino a la implementación de controles de bajo nivel y demás componentes involucrados en la puesta en marcha de un escenario de estas características. Se co-desarrolla y pone en prueba el entorno de ejecución como extensión de la herramienta, y se escriben los adaptadores necesarios para ésta.
Por otra parte la puesta en ejecución del controlador y la prueba de distintos esquemas de ejecución permiten razonar sobre el efecto que tiene el algoritmo de resolución y quizás en particular el sistema de ranking en los controladores resultantes. Un único caso de estudio es insuficiente para realizar generalización y motivar con fundamento un análisis de éstos, pero es un comienzo y precedente para el razonamiento guiado por ejecución.
También se plantea el cambio de configuración como característica principal y deseable de las técnicas empleadas, para esto se definen las dimensiones de cambio y las pruebas vinculadas. Si bien dicha característica es cubierta se podría decir por definición al utilizar técnicas de modelado y síntesis de naturaleza declarativa, esta tesis intenta hacer evidente esta capacidad como uno de los valores fundamentales del conjunto de teoría y herramienta que forman el entorno de trabajo de MTSA.

1.4. Estructura de la tesis

La estructura de esta tesis es la siguiente: en el capítulo 2 se presentan los fundamentos teóricos que dan soporte a las técnicas de síntesis y modelado. El capítulo 3 hace una presentación de la herramienta MTSA y de los modelos del sistema en la sintaxis extendida de FSP. El capítulo 4 se explica la motivación para extender la herramienta y dar soporte a la ejecución de controladores. El capítulo 5 describe el trabajo realizado para dar soporte funcional al sistema a través del robot N6 y el trabajo de adaptación al entorno de MTSA. En 6 se detalla la extensión de la herramienta y la forma de utilizar la nueva funcionalidad para ejecutar controladores sintetizados. En el capítulo 7 se describen los casos explorados, se presentan resultados, una breve discusión sobre el trabajo realizado y conclusiones de cierre.

2. FUNDAMENTOS

2.1. El Mundo y la Máquina

Comenzamos por ofrecer nociones acerca de la visión general de la ingeniería de requerimientos. En particular, presento el punto de vista de ingeniería de requerimientos de Zave y Jackson [3, 4, 14] y de Letier y Van Lamsweerde [13, 7]. Ambos puntos de vista están de acuerdo en que la distinción entre los problemas del *Mundo* y la solución de *Máquina* es fundamental para comprender si la máquina resuelve correctamente el problema en cuestión. 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. El problema define una parte del mundo real que queremos mejorar mediante la construcción de una máquina. Por lo general, incluye algunos de los componentes que interactúan con el mundo siguiendo normas y procesos conocidos. Por ejemplo, una herramienta de perforación, un brazo de robot o las reglas para el procesamiento de productos que entran en una línea de producción (véase la Figura 2.1).

Por otra parte, se espera que la solución de la máquina pueda resolver el problema. Por ejemplo, el ejemplo de la Figura 2.1 muestra que la célula de producción debe iniciar el procesamiento de los productos, solo si están disponibles en la bandeja de entrada. De hecho, la sentencia $inTray[p] \rightarrow get.InTray[p]$ muestra que se espera que el brazo del robot debe recoger los productos de la bandeja en el caso de que estén listos para ser procesados. Por último, los fenómenos compartidos son una parte común entre el problema mundo y la solución máquina. Por lo tanto, define la interfaz donde la máquina interactúa con el mundo, representada como la intersección de los dos conjuntos en la Figura 2.1. La máquina se hace referencia en el contexto de síntesis como el controlador, se utilizará uno u otro término en función del contexto. Podemos referirnos al problema mundo como el *modelo de entorno*.

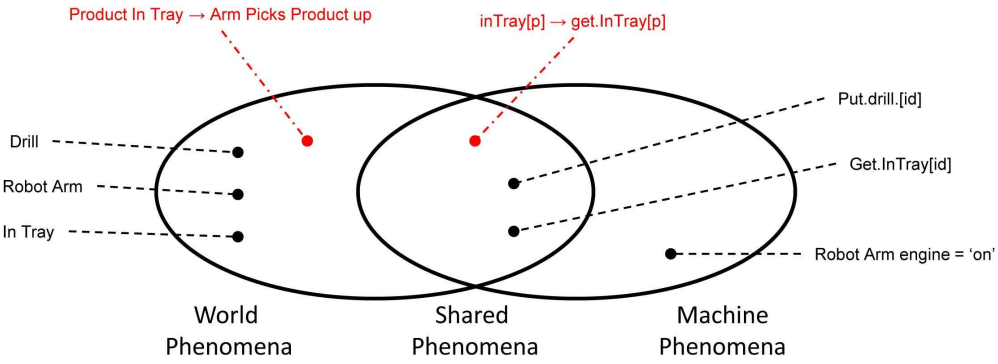


Fig. 2.1: World and Machine Phenomena

Las sentencias que describen los fenómenos, tanto del mundo como la máquina pueden variar en alcance y forma [3, 11]. Las sentencias pueden ser en modo *indicativo* u *optativo* [optative mood]. En [12], las sentencias que describen el sistema se caracterizan por ser *descriptivas* y *prescriptivas*.

Sentencias *descriptivas* representan propiedades sobre el sistema que son independientes de cómo se comporta el sistema. Las sentencias *descriptivas* están en modo *indicativo*.

Las sentencias *prescriptivas* afirman propiedades deseables que pueden estar presentes o no. De hecho, las sentencias prescriptivas deben ser aplicadas por los componentes del sistema. Naturalmente, las declaraciones *prescriptivas* pueden cambiar, fortaleciéndose / debilitándose, o incluso ser eliminadas, mientras que no se pueden tornar *descriptivas*.

Como se mencionó anteriormente, los estados pueden variar en su alcance. Tanto las sentencias prescriptivas como también descriptivas pueden referirse a características de la máquina que no son compartidas con el mundo. Otras sentencias pueden referirse a fenómenos compartidos por la máquina y el mundo. Más precisamente, una *propiedad de dominio* es una sentencia descriptiva sobre el problema mundo. Estas se deben tener independientemente del comportamiento del sistema. En este trabajo llamamos *Modelo Ambiente*, al conjunto de propiedades de dominio para un problema particular.

Un *supuesto de ambiente* es una sentencia que podría no suceder [may not hold] y debe ser satisfecha por el ambiente. Un requisito de software, o *Requisito* para abreviar, es una sentencia prescriptiva a ser aplicada por la máquina, independientemente de cómo se comporta el problema mundo y debe ser formulada en términos de los fenómenos compartidos entre la máquina y problema mundo.

Siguiendo [13, 7] decimos que una acción es supervisada / controlable si tal acción es supervisada / controlable por la máquina. Podemos referirnos a acciones supervisadas como acciones no controlables, ya que están controladas por el ambiente.

2.2. Sistemas de Transición Etiquetados

Vamos a describir y fijar una notación para los sistemas de transición etiquetados o Labeled Transition Systems (LTS) [6], que son ampliamente utilizados para el modelado y análisis del comportamiento de los sistemas concurrentes y distribuidos. LTS es un sistema de transición de estados donde las transiciones se etiquetan con acciones. El conjunto de las acciones de un LTS se llama su alfabeto para la comunicación y las interacciones que el sistema modelado puede tener con su entorno.

Definición *(Sistema de Transición Etiquetado) [6]* 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.

Dado $(s, \ell, s') \in \Delta_E$ decimos que ℓ está activo desde s en E. Decimos que un LTS E es *determinístico* si (s, ℓ, s') y (s, ℓ, s'') están en Δ_E implica que $s' = s''$. Para un estado s denotamos $\Delta_E(s) = \{\ell \mid (s, \ell, s') \in \Delta_E\}$. Dado un LTS E, podríamos referirnos a su alfabeto como αE.

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

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

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

Definición *(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 una *LTS Legal* para y LTS E con respecto a A_{E_u}, si para todos los estados en la composicón $(s_E, s_M) \in S_{E\|M}$ los contiene, una acción $\ell \in A_{E_u}$ es deshabilitada en (s_E, s_M) si y solo si ésta está también deshabilitada en $s_E \in E$. En otras palabras, M no restringe E con respecto a A_{E_u}.

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

Definición *(Estados Alcanzables)* Considérese 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 $s_0, \ell_0, s_1, \ell_1, \ldots$, 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 $Reach(E)$.

A traves de esta tesis vamos a limitar la atención a aquellos LTSs E donde todos sus estados $s \in S_E$, s son alcanzables.

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

Lógica lineal temporal (LTL) es ampliamente usada para describir el comportamiento de requerimientos [2, 13, 8, 5]. La motivación para escoger un LTL de flujos es que este provee un framework uniforme para especificar propiedades basada en estados en modelos basados en eventos [2]. Fluent Linear Temporal Logic (FLTL) [2] es una lógica linear-time temporal para razonar acerca de flujos. Un *flujo Fl* es definido por un par de conjuntos y un valor Booleano: $Fl = \langle I_{Fl}, T_{Fl}, Init_{Fl} \rangle$, 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 $true$o $false$ indicado por $Init_{Fl}$. Toda acción $\ell \in Act$ induce un flujo, a saber $\dot{\ell} = \langle \ell, Act \setminus \{\ell\}, \text{false} \rangle$. Finalmente, el alfabeto de un flujo es la unión de sus acciones iniciadoras y finalizadoras.

Sea $\mathscr{F}$ el conjunto de todas los posibles flujos sobre Act. Una fórmula FLTL es definida inductivamente usando los conectores Booleanos estándards y operadores temporales X (next), U (strong until) de la siguiente manera:

$$\varphi ::= Fl \mid \neg\varphi \mid \varphi \vee \psi \mid \mathrm{X}\varphi \mid \varphi \mathrm{U} \psi,$$

donde $Fl \in \mathscr{F}$. Introduciremos $\wedge$, F ($\Diamond$ eventually), y G ($\Box$ always) como una comodidad sintáctica. Sea Π el conjunto de trazas infinitas sobre Act. La traza $\pi = \ell_0, \ell_1, \ldots$ satisface un flujo Fl en la posición i, denotado como $\pi, i \models Fl$, si y sólo si una de las siguientes condiciones es válida:

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

Dada una traza infinita π, la fórmula que satisface φ en la posición i, denotada como $\pi, i \models \varphi$, es definida como se muestra en la Figura 2.2. Decimos que φ se cumple en π, denotado como $\pi \models \varphi$, si $\pi, 0 \models \varphi$. Una fórmula $\varphi \in$ FLTL es cierta si un LTS E (denotado como $E \models \varphi$) si éste es cierto en toda traza infinita producida por E.

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

Fig. 2.2: Semántica para el operador de satisfacción

2.4. Problemas de control

El problema de síntesis de control consiste en producir automáticamente una máquina que restringe la ocurrencia de eventos controlados basada en la observación de eventos que han ocurrido. Cuando se despliega en un entorno apropiado esta máquina asegura la satisfacción del conjunto de objetivos de sistema provistos. La satisfacción de estos objetivos depende de la satisfacción de las presunciones por parte del entorno. En otras palabras, damos una descripción del entorno, presunciones, objetivos de sistema y un conjunto de acciones controlables.
Una solución para el *problema de control basado en eventos* es encontrar una máquina tal que si comportamiento concurrente al entorno satisface las presunciones satisface también los objetivos.
Definimos el problema de síntesis de control para modelos basados en eventos de la siguiente forma: Dada una LTS que describe el comportamiento del entorno, un conjunto de acciones controlables, un conjunto de fórmulas FLTL para las presunciones del ambiente y un conjunto de fórmulas FLTL para los objetivos de sistema, el problema de control LTS consiste en encontrar una LTS que restringe sólo la ocurrencia de acciones controlables y garantiza que la composición paralela del ambiente con dicha LTS estará libre de deadlocks y que, si las presunciones de ambiente son satisfechas, satisfacerá también los objetivos de sistema.

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

Ahora definimos el problema de control SGR(1) que es computable en tiempo polinomial. Tiene base en los problemas GR(1) y de seguridad pero en un contexto de modelado basado en eventos. Requerimos que el modelo del entorno E sea una LTS determinística para asegurar que el controlador tenga observación completa del estado del entorno. Requerimos que H sea $(\emptyset, I), (A_s, G)$ donde I es un invariante de seguridad de la forma $\Box\rho$ las presunciones As son una conjunción de subfórmulas FLTL de la forma $\Box\Diamond\phi$, el objetivo G es una conjunción de subfórmulas FLTL de la forma $\Box\Diamond\gamma$ y ρ, ϕ y γ son una combinación booleana de Flujos.

Definición *(Control LTS SGR(1))* Un problema de control LTS $\varepsilon =< E, H, A_c >$ 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 ρ, ϕ y γ son una combinación booleana de Flujos.

2.5. Dominios falibles

Consideramos una técnica que permite la síntesis de controladores aún en entornos que exhiben fallas. Esta técnica toma el nombre de *síntesis con fallas*. Por lo general no puede controlarse a un entorno malicioso para conseguir los objetivos de sistema. De todas formas, proponemos nociones realistas de equidad que permiten a los controladores comportarse de la forma esperada por ejemplo para el caso en el que deberían intentar repetir una acción hasta conseguir una respuesta exitosa.
Distinguimos a las fallas de otras acciones de la siguiente manera: para cada problema de control definimos un conjunto de triplas $try - response$ (prueba, respuesta), una tripla de ésta característica captura la relación entre acciones controlables y sus reacciones de éxito o fracaso. Se precisa que 1) la acción "try"sea controlable, 2) todas las acciones pertenecientes a una tripla $try - response$ sean únicas con respecto a otras triplas del conjunto, 3) un reintento no ocurra antes de una respuesta 4)las respuestas ocurran solamente como resultado de un intento, 5)existe como máximo una respuesta por cada intento y 6) la decisión de fallar o tener éxito no sea forzada por otras acciones, con lo que la falla está habilitada sí y sólo sí el éxito está habilitado.
Se define el operador W como $(\phi U \psi) \vee G\phi$.

Definición *(Try-Response)* Dada una LTS $M = (S_M, L_M, \Delta_M, s_{M_0})$ donde $L_C \subseteq L_M$, decimos que un conjunto $T = (try_i, suc_i, fail_i)$ es un conjunto $try - response$ para M si se cumplen las siguientes condiciones para todo i:

1. $try_i \subseteq L_C, suc_i, fail_i \in L \setminus L_C$ y $suc_i \neq fail_i$
2. Para todo $j \neq i$, $try_i, suc_i, fail_i \cap try_i, suc_i, fail_i = \emptyset$
3. $\neg(\dot{fail}_i \vee \dot{suc}_i) W \dot{try}_i$
4. $\Box(\dot{try}_i \Rightarrow \bigcirc(\neg \dot{try}_i W(\dot{fail}_i \vee \dot{suc}_i)))$
5. $\Box((\dot{fail}_i) \vee \dot{suc}_i) \Rightarrow \bigcirc(\neg(\dot{fail}_i \vee \dot{suc}_i W \dot{try}_i))$
6. Para todo $s \in S_M$, $fail_i$ está habilitado para s sí y sólo sí suc_i está habilitado para s

La técnica presentada demanda una noción de equidad más fuerte, que describa la presunción de que si se ejecuta un intento con infinita frecuencia se debe conseguir un éxito con infinita frecuencia. Esta noción de equidad reforzada está, de hecho, fuertemente vinculada con la estructura de los modelos del entorno y el controlador. Lo que se precisa es que, para cada estado global (un estado de $E \| M$), un try_i sucede con infinita frecuencia, $fail_i$ no sucede con infinita frecuencia. Una forma más intuitiva de expresarlo sería pensar que la decisión de fallar es equitativa e independiente del estado del controlador o el entorno.
La siguiente definición captura esta noción de equidad reforzada. Requiere que para cada transición etiquetada con un try, si es tomada con infinita frecuencia luego $success$ sucede con infinita frecuencia antes del próximo try.

Definición *(t-strong fairness)* Dada una LTS M y un try-response T para M, una traza $\pi \in tr(M)$ es *t-strong fair* (fuertemente t-equitativa) con respecto a M y T si para todo $(try_i, suc_i, fail_i) \in T$ y para todas las transiciones $t = (s, try_i, s')$ vale que: $\pi' \models \Box\Diamond try'_i \Rightarrow \Box\Diamond(\neg try_i U suc_i)$, donde $\pi' = \varepsilon' \mid_{L_M \cup \{try_i^b\}}$, $\varepsilon' = \varepsilon \mid_{[s.try_i.s^b/s.try_i.try_i^b,s^b]}$, y ε es una ejecución de M tal que $\varepsilon' \mid_{L_M} = \pi$.

Cabe notarse que $w \mid_A$ es la proyección de la palabra w sobre el alfabeto A y que $w_{[v/v^b]}$ es el resultado de reemplazar en w todas las ocurrencias de v con v^b.

Hace falta extender la noción de equidad aún más para cumplir la noción intuitiva de que las presunciones del entorno deben ser independientes de las fallas, particularmente porque la elección de una falla o de un éxito puede entenderse como no-determinística dado que abstrae la verdadera causa del éxito o fracaso.
Formalizamos esta noción de esta manera: restringimos las trazas de interés a aquellas que satisfacen que las presunciones deben poder cumplirse con infinita frecuencia sin observar fallas, o, más precisamente, que si el controlador intenta con suficiente frecuencia, entonces no sólo tendrá éxito, sino que tendrá éxito a la vez que satisface todas las presunciones. Esto es, que si las presunciones y las fallas son verdaderamente independientes, intentar la acción con suficiente frecuencia garantiza que en algún punto luego de un intento, ninguna falla sucederá en tanto no se hayan satisfecho todas las presunciones.

Definición *(Strong Independent Fairness)* Dada una LTS M y un try-response T para M y A un conjunto de fórmulas FLTL, una traza $\pi \in tr(M)$ se dice *Strong Independent Fair* (de equidad reforzada independiente) respecto de A si para todo $(try_i, suc_i, fail_i) \in T$ y para toda transición $t = (s, try_i, s')$ vale que $\pi \models \Box\Diamond try'_i \Rightarrow \Box\Diamond((\neg try_i U suc_i) \wedge (\bigwedge_{i=1}^{n}(\neg(\bigvee_{j=1}^{n} fail_j) W A_i)))$, donde $\pi' = \varepsilon' \mid_{L_M \cup try_i^b}, \varepsilon' = \varepsilon \mid_{[s.try_i.s^b/s.try_i.try_i^b,s^b]}$, y ε es una ejecución de M tal que $\varepsilon' \mid_{L_M} = \pi$.

Formalizamos a continuación el problema de control bajo las condiciones de equidad arriba presentadas. Toma el nombre de *problema de control con éxito recurrente* (recurrent success control problem). Para todas las trazas que sean de equidad reforzada independiente garantiza propiedades generales de seguridad y progreso que son del tipo GR(1). Extendemos el problema de control SGR(1) definido en 2.4 introduciendo fallas y expectativas sobre la equidad del ambiente.

Definición *(Recurrent Success)* Dado un problema de control SGR(1) LTS $\mathscr{L} =< E, H, L_c >$ y un try-response T para $\mathscr{L}$, la solución del problema de control de éxito recurrente $\mathscr{R} =< \mathscr{L}, T >$ consiste en encontrar un LTS M tal que M con acciones controlables L_c y no controlables $\overline{L_c}$ es un entorno legal para E, $E \| M$ no contiene deadlocks y para cada par $(As_i, G_i) \in H$, para cada $(try_i, suc_i, fail_i)$ y para cada traza de equidad reforzada independiente π en $M \| E$ vale que si $\pi \models As_i$ entonces $\pi \models G_i$.

2.6. Procesos de Estados Finitos (FSP)

Hasta ahora, hemos descripto LTSs (MTSs) definiendo sus componentes, ej. estados, acciones, relaciones de transición (requeridos y posibles), y el estado inicial. Esta representación es válida para LTSs (MTSs) con pocos estados. Sin embargo, esta representación se torna impráctica cuando trabajamos con LTSs (MTSs) extensos. Por esta razón, usamos

un proceso simple llamado algebra de *Procesos de Estados Finitos* (FSP) que especifican textualmente un LTSs [10, 9].

FSP es una especificación de lenguaje con una bien definida semántica en términos de LTSs (MTSs), que provee de manera concisa una forma de describir un LTSs. Cada expresión FSP E puede ser mapeada en un LTS (MTS) finito, usaremos $lts(E)$ para denotar el LTS (MTS) que le corresponde. A continuación discutiremos brevemente la sintaxis FSP.

Como un ejemplo, en la Figura 2.3, mostramos el código FSP de un proceso de cocción de cerámica.

En FSP, los nombres de procesos comienzan con mayúsculas y las acciones comienzan con minúscula. El código para la cocción de cerámica define 2 procesos FSP, uno modela el proceso que simplemente está en espera, llamado $fsp(IDLE)$, y otro proceso llamado $fsp(DOMAIN)$. Adicionalmente, $fsp(DOMAIN)$ define los procesos auxiliares $fsp(COOKING)$, $fsp(COOKED)$, y $fsp(OH)$. Los procesos auxiliares son locales para el proceso FSP de la forma que han sido definidos. $fsp(DOMAIN)$ es definido usando la acción y el operador $fsp(->)$ junto a la recursión. Por ejemplo, el proceso se encuentra definido para comenzar ejecutando indistintamente $fsp(idle)$ quedándose en este o $fsp(cook)$ que comienza con el proceso de $fsp(COOKING)$.

```
IDLE = idle -> IDLE.
DOMAIN = (idle->DOMAIN | cook->COOKING),
COOKING = (cooking->COOKING | cook->OH
            | finishedCooking->COOKED),
COOKED = (moveToBelt->DOMAIN | cook->COOKING),
OH = (overHeated->OH).
||IDLE_DOMAIN = (IDLE||DOMAIN).
```

Fig. 2.3: FSP Example

FSP soporta numerosos operadores para la composición como la composición en paralelo de LTS y MTS, o la fusión de MTSs [1]. El operador para composición en paralelo, denotado como $fsp(||)$ es definido para preservar la semántica de la composición en paralelo de LTS presentado en 2.2. Así, dados dos procesos FSP $fsp(P)$ y $fsp(Q)$, tenemos que:fsp(lts(P||Q)=lts(P)||lts(Q)).

Los procesos FSP que son definidos mediante la composición de dos procesos no auxiliares son llamados procesos compuestos y sus nombres contienen el prefijo $fsp(||)$. Así la composición en paralelo de los procesos FSP $fsp(IDLE)$ y $fsp(DOMAIN)$ es $fsp(||IDLE_DOMAIN = (IDLE||DOMAIN))$.

Finalmente, un FSP posee un número de palabras reservadas que son usadas antes de la definición de un proceso y estas fuerzan al MTSA a desarrollar operaciones complejas sobre el proceso. Por ejemplo, el comando $fsp(minimal)$ indica que MTSA construya el LTS/MTS minimal respetando fuertemente la semántica equivalente y la instrucción $fsp(deterministic)$ instruye para que MTSA construya el LTS minimal con respecto a su traza.

También el FSP permite definir propiedades FLTL. Un flujo que marca aquellos estados donde la cerámica es cocinada puede ser expresado en FSP con el siguiente código: $fsp(fluentCooking =< cook, finishedCooking > initially 0)$. $fsp(Cooking)$ se encuentra inicialmente en falso, este se torna verdadero con $fsp(cook)$ y este se torna falso nuevamente cuando ocurre $fsp(finishedCooking)$.

Sumarizando, FSP provee el soporte requerido para especificar LTSs and FLTL formulas. Dicho soporte es requerido para expresar el modelado del entorno y los objetivos del controlador.

3. MTSA COMO HERRAMIENTA DE MODELADO Y SÍNTESIS

MTSA es una herramienta de modelado y síntesis basada en sistemas de transiciones etiquetadas. En esta tesis es utilizada como marco de trabajo, cubriendo los siguientes requerimientos:

Modelado del comportamiento del robot, el mapa, las bandejas y las propiedades esperadas del sistema a través de una extensión de la sintaxis de FSP que permite describir propiedades como fórmulas FLTL y controladores como un conjunto de propiedades de progreso, seguridad, acciones falibles, controlables y no controlables

Síntesis del controlador expresado conforme a las propiedades declaradas, de forma que al componerlo con el entorno permita satisfacer los objetivos de progreso y seguridad

Ejecución del controlador sintetizado a través del marco de ejecución, controlando los componentes del sistema a través de mensajes específicos de dominio

En el capítulo siguiente se describe la extensión de la herramienta para permitir ejecutar el controlador sintetizado en el entorno físico.

3.1. Construcción

En MTSA, los modelos se describen a través de una extensión del lenguaje de Procesos de Estados Finitos (FSP). FSP es un lenguaje textual centrado en la construcción composicional de modelos complejos que originalmente fue usado para describir LTS.
FSP incluye varios operadores tradicionales para describir modelos de comportamiento, como puede ser el prefijo de acción ($->$), elección ($|$), composición secuencial (;), composición paralela ($||$) y mezcla.
La semántica de mezcla es tal que dadas dos descripciones parciales del mismo componente, el operador de mezcla devuelve una MTS que combina la información provista por las descripciones parciales originales.
Aún cuando los operadores composicionales permiten la consrucción de MTS complejas, construir los modelos a ser compuestos sigue siendo una tarea dificultosa de intenso trabajo que requiere 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 las especificaciones declarativas de los requerimientos, escenarios y casos de uso.
La palabra clave $constraint$ de MTSA se usa en conjunción 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 MTS que carateriza a todos los modelos LTS libres de deadlock que satisfacen la fórmula FLTL. Al sintetizar y mezclar modelos MTS obtenidos con definiciones FLTL, se puede construir de forma iterativa una MTS que carateriza 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 MTS 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 descripto en la especificación del escenario, provee una MTS que caracteriza a todas la implementaciones que satisfacen dicha especificación (i.e. la cota inferior del comportamiento esperado del sistema).

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.
MTSA soporta varios tipos de análisis, el más básico involucra la inspección de modelos MTS y está soportado a través de la construcción automática de representaciones visuales de los modelos MTS escritos usando FSP. Esta inspección queda sujeta al tamaño del modelo, esta limitación puede mitigarse haciendo uso de los operadores de minimización y ocultamiento.
Aunque la inspección y animación no permiten una exploración exhaustiva de los modelos MTS, MTSA implementa un número de técnicas de análisis automáticas para éste propósito. En particular, MTSA permite verificar si un modelo MTS satisface una propiedad expresada en FLTL. Un modelo MTS 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 MTS y una fórmula FLTL. Mientras que un Modelo MTS 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 deadlock o bien hay una combinación de implementaciones que exhiben deadlocks y otras que no.

3.3. Modelando el sistema

A continuación se presentan los modelos que representan el comportamiento de los distintos componentes del sistema. Se describe informalmente su función y se da la representación FSP de los mismos.

3.3.1. Identificando modelos

Para el caso de estudio propuesto se identifican cuatro sub-modelos para el ambiente, estos son: el robot, el mapa, las bandejas de entrada y salida y el mecanismo de reubicación. Se describen por separado prestando cuidado a los estados en los que deben interactuar para permitir que la composición paralela provea una buena descripción del ambiente en la forma en que se comporta ante la ausencia de un controlador.
Los procesos representan el comportamiento de los agentes en el dominio restringido de nuestro caso de estudio. Suponemos que la posición del robot puede expresarse en la especificación del mapa a partir de una posición inicial, y de las relaciones entre los nodos de control que capturan posición en la grila y orientación, esto no significa que, por ejemplo,

la posición del robot se exprese con suficiente detalle para otro dominio donde se podría preferir una codificación en el plano (x, y, θ) con una aproximación discreta conveniente.
La interacción entre el robot y el mapa queda capturado en la especificación del mapa, esto puede romper, al menos conceptualmente, la separación lógica entre ambos. Una solución alternativa podría expresar los procesos con alfabetos disjuntos y luego reemplazar el nombre de las acciones en el paso previo a la composición.

3.3.2. Robot

El robot es de tipo planar, capaz de seguir líneas negras, detectar puntos de intersección, girar a derecha, izquierda, dar media vuelta y permitir la carga y descarga de un carro en el que transporta los ítems.
Se restringe en el proceso del robot la capacidad de realizar dos giros seguidos, de forma que luego de un giro debe realizarse un seguimiento. Ésta restricción, que aquí se realiza como parte del modelo, podría codificarse como propiedad de seguridad, su motivación tiene que ver con reforzar (declarativamente) una noción intuitiva de progreso, ya que la probabilidad de recibir un mensaje de falla luego de un giro (el robot es incapaz de ubicarse nuevamente sobre la línea negra) es mayor que la de recibir un mensaje de falla luego de realizar un seguimiento.
El proceso de reubicación se codifica también en el modelo con la secuencia:

```
1 ROBOT_MOVEMENT = (robot.follow->ROBOT_REPLY
2         |robot.turnRight->ROBOT_REPLY
3         |robot.turnLeft->ROBOT_REPLY
4         |robot.turnAround->ROBOT_REPLY),
5 ROBOT_LOST = (robot.retry->ROBOT_RETRY),
6 ROBOT_RETRY = (robot.lost->ROBOT_LOST|robot.successRetry->ROBOT_MOVEMENT)...
```

Nótese que éste es un modelo simplificado del robot, cuya intención es expresar el funcionamiento del mecanismo de reubicación que opera en paralelo con el proceso del reubicador manual.

```
1 RELOCATOR = (robot.successRetry->RELOCATOR_RELOCATE),
2 RELOCATOR_RELOCATE = (relocator.go_N_0_0_N->RELOCATOR).
```

Asegurando que luego de un reintento el robot ha podido ubicarse sobre una línea negra, sin importar cual, y el reubicador manual lo llevará a un nodo conocido desde el cual podrá continuar progresando hacia el objetivo. Se describe también la capacidad de carga del robot, se puede entender que hay dos máquinas muy similares para el movimiento del robot y que la carga y descarga salta entre entre una y otra con un comportamiento mutuamente exclusivo, esto es, la carga lleva a una máquina equivalente donde se restringe la carga y se habilita la descarga, y la descarga lleva a una máquina donde se restringe la descarga y se habilita la carga.
A continuación la descripción completa del robot para uno de los ejemplos de esta tesis.

```
1 ROBOT = ROBOT_MOVEMENT_U,
2 ROBOT_MOVEMENT_U = (inTray.unload -> ROBOT_LOADING_MOVING
3         |robot.follow->ROBOT_REPLY_U
4         |robot.turnRight->ROBOT_SHOULD_FOLLOW_U
5         |robot.turnLeft->ROBOT_SHOULD_FOLLOW_U
6         |robot.turnAround->ROBOT_SHOULD_FOLLOW_U),
7 ROBOT_SHOULD_FOLLOW_U = (robot.lost->ROBOT_LOST_U|robot.success->ROBOT_FOLLOW_U),
```

```
8  ROBOT_FOLLOW_U = (inTray.unload -> ROBOT_LOADING_FOLLOWING|robot.follow->ROBOT_REPLY_U),
9  ROBOT_REPLY_U = (robot.lost->ROBOT_LOST_U|robot.success->ROBOT_MOVEMENT_U),
10 ROBOT_LOST_U = (robot.retry->ROBOT_RETRY_U),
11 ROBOT_RETRY_U = (robot.lost->ROBOT_LOST_U|robot.successRetry->ROBOT_MOVEMENT_U),
12 ROBOT_LOADING_MOVING = (inTray.unloaded -> ROBOT_MOVEMENT_L | inTray.fell ->
       ROBOT_MOVEMENT_U),
13 ROBOT_LOADING_FOLLOWING = (inTray.unloaded -> ROBOT_FOLLOW_L | inTray.fell ->
       ROBOT_FOLLOW_U),
14 ROBOT_MOVEMENT_L = (outTray.load -> ROBOT_UNLOADING_MOVING
15         |robot.follow->ROBOT_REPLY_L
16         |robot.turnRight->ROBOT_SHOULD_FOLLOW_L
17         |robot.turnLeft->ROBOT_SHOULD_FOLLOW_L
18         |robot.turnAround->ROBOT_SHOULD_FOLLOW_L),
19 ROBOT_SHOULD_FOLLOW_L = (robot.lost->ROBOT_LOST_L|robot.success->ROBOT_FOLLOW_L),
20 ROBOT_FOLLOW_L = (outTray.load -> ROBOT_UNLOADING_FOLLOWING|robot.follow->ROBOT_REPLY_L),
21 ROBOT_REPLY_L = (robot.lost->ROBOT_LOST_L|robot.success->ROBOT_MOVEMENT_L),
22 ROBOT_LOST_L = (robot.retry->ROBOT_RETRY_L),
23 ROBOT_RETRY_L = (robot.lost->ROBOT_LOST_L|robot.successRetry->ROBOT_MOVEMENT_L),
24 ROBOT_UNLOADING_MOVING = (outTray.loaded -> ROBOT_MOVEMENT_U | outTray.fell ->
       ROBOT_MOVEMENT_U),
25 ROBOT_UNLOADING_FOLLOWING = (outTray.loaded -> ROBOT_FOLLOW_U | outTray.fell ->
       ROBOT_FOLLOW_U).
```

3.3.3. Mapa

El mapa describe la distribución de los nodos de detención, relación entre ellos a través de las vías (las bandas negras que llevan de un puesto a otro) y la ubicación de las bandejas en la planta. También guarda conocimiento de la orientación del robot, de forma que su ubicación espacial puede darse por la tripla $< x, y, o >$ donde, si se piensa a la distribución de los puestos o nodos de intersección como una grilla regular, x define la columna sobre la que se encuentra el robot y la fila y o la orientación perteneciendo al conjunto $(norte, este, sur, oeste)$. También se especifica el efecto de ejecutar el mecanismo de reubicación manual con la transición:

```
1  N_X_Y_O = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> ...
```

Las bandejas se habilitan en una posición y orientación dada, por ejemplo:

```
1  N_0_1_S = (relocator.go_N_0_0_N -> N_0_0_N | ... | outTray.load -> N_0_1_S)...
```

A continuación una descripción completa del mapa.

```
1  MAP = N_1_0_N,
2  N_0_0_N = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_0_0_S |
       robot.turnRight -> N_0_0_E | robot.turnLeft -> N_0_0_W),
3  N_0_0_E = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_0_0_W |
       robot.turnRight -> N_0_0_S | robot.turnLeft -> N_0_0_N | robot.follow -> N_0_1_E),
4  N_0_0_S = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_0_0_N |
       robot.turnRight -> N_0_0_W | robot.turnLeft -> N_0_0_E | robot.follow -> N_1_0_S),
5  N_0_0_W = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_0_0_E |
       robot.turnRight -> N_0_0_N | robot.turnLeft -> N_0_0_S),
6  N_0_1_N = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_0_1_S |
       robot.turnRight -> N_0_1_E | robot.turnLeft -> N_0_1_W),
7  N_0_1_E = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_0_1_W |
       robot.turnRight -> N_0_1_S | robot.turnLeft -> N_0_1_N),
```

```
8  N_0_1_S = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_0_1_N |
       robot.turnRight -> N_0_1_W | robot.turnLeft -> N_0_1_E | robot.follow -> N_1_1_S |
       outTray.load -> N_0_1_S),
9  N_0_1_W = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_0_1_E |
       robot.turnRight -> N_0_1_N | robot.turnLeft -> N_0_1_S | robot.follow -> N_0_0_W),
10 N_1_0_N = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_1_0_S |
       robot.turnRight -> N_1_0_E | robot.turnLeft -> N_1_0_W | robot.follow -> N_0_0_N |
       inTray.unload -> N_1_0_N),
11 N_1_0_E = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_1_0_W |
       robot.turnRight -> N_1_0_S | robot.turnLeft -> N_1_0_N | robot.follow -> N_1_1_E),
12 N_1_0_S = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_1_0_N |
       robot.turnRight -> N_1_0_W | robot.turnLeft -> N_1_0_E),
13 N_1_0_W = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_1_0_E |
       robot.turnRight -> N_1_0_N | robot.turnLeft -> N_1_0_S),
14 N_1_1_N = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_1_1_S |
       robot.turnRight -> N_1_1_E | robot.turnLeft -> N_1_1_W | robot.follow -> N_0_1_N),
15 N_1_1_E = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_1_1_W |
       robot.turnRight -> N_1_1_S | robot.turnLeft -> N_1_1_N),
16 N_1_1_S = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_1_1_N |
       robot.turnRight -> N_1_1_W | robot.turnLeft -> N_1_1_E),
17 N_1_1_W = (relocator.go_N_0_0_N -> N_0_0_N | robot.turnAround -> N_1_1_E |
       robot.turnRight -> N_1_1_N | robot.turnLeft -> N_1_1_S | robot.follow -> N_1_0_W).
```

3.3.4. Bandejas

Las bandejas de entrada y salida representan el punto de contacto con otras líneas de la planta industrial o incluso con una fuente externa a la planta, por ejemplo un lote de ítems provistos por una tercera entidad en las líneas de montaje. Se incluye la posibilidad de perder el ítem durante la carga o la descarga, representando la carga con la tripla $(inTray.load, inTray.unloaded, inTray.fell)$ y $(outTray.load, outTray.loaded, outTray.fell)$ de lo que se considera un dominio falible.

```
1  INTRAY = (inTray.load -> inTray.unload -> (inTray.unloaded -> INTRAY | inTray.fell ->
       INTRAY)).
2  OUTTRAY = (outTray.load ->(outTray.fell -> OUTTRAY | outTray.loaded -> outTray.unload ->
       OUTTRAY)).
```

3.3.5. Reubicación manual

Dado que un mensaje de falla por parte del robot indica que su posición dentro del mapa es desconocida se introduce un mecanismo de reubicación, en este caso manual, que supone conocimiento de la posición del robot y de un nodo de reubicación de forma que el controlador pueda asegurar la satisfacción de los objetivos de sistema. Este componente particular podría reemplazarse por un procedimiento automático que incluya una cámara con capacidad de identificar la posición y orientación del robot y el mapa, y envíe mensajes de bajo nivel (control de los microrreductores) llevando al robot hasta el nodo de reubicación.

```
1  RELOCATOR = (robot.successRetry->RELOCATOR_RELOCATE),
2  RELOCATOR_RELOCATE = (relocator.go_N_0_0_N->RELOCATOR).
```

3.3.6. Definición del controlador

Para la definición del controlador es necesario explicitar el conjunto de acciones controlables, no controlables y falibles, esto puede expresarse en notación FSP utilizando el elemento de sintaxis **set**. Primero se describen los conjuntos según una partición lógica.

```
set ActionCommands =
    {robot.follow,robot.turnRight,robot.turnLeft,robot.turnAround,robot.retry}
set RelocateCommands = {relocator.go_N_0_0_N}
set ReplyCommands = {robot.lost,robot.success,robot.successRetry}
set FailActions = {robot.lost,inTray.fell,outTray.fell}
```

Se particiona luego el espacio de acciones entre controlables y no controlables.

```
set ControllableActions = {ActionCommands,inTray.unload,outTray.load}
set UncontrollableActions =
    {FailActions,RelocateCommands,robot.success,robot.successRetry,inTray.load,
    inTray.fell, inTray.unloaded,outTray.loaded,outTray.fell,outTray.unload}
```

Se define el conjunto universal para el sistema.

```
set Alphabet = {ControllableActions,UncontrollableActions}
```

Y el entorno como composición paralela de los agentes involucrados.

```
||Scenario = ( MAP || ROBOT || RELOCATOR || INTRAY || OUTTRAY).
```

Se definen las propiedades de seguridad como máquinas que restringen el comportamiento indeseado.

```
property NO_2_RIGHT = (Alphabet\{robot.turnRight}->NO_2_RIGHT |
    robot.turnRight->PREV_RIGHT ),
PREV_RIGHT = (robot.success->PREV_RIGHT2 | Alphabet\{robot.success}->NO_2_RIGHT),
PREV_RIGHT2 = (Alphabet\{robot.turnRight}->NO_2_RIGHT ).
property NO_2_LEFT = (Alphabet\{robot.turnLeft}->NO_2_LEFT | robot.turnLeft->PREV_LEFT ),
PREV_LEFT = (robot.success->PREV_LEFT2 | Alphabet\{robot.success}->NO_2_LEFT),
PREV_LEFT2 = (Alphabet\{robot.turnLeft}->NO_2_LEFT ).
property NO_2_TURNAROUND = (Alphabet\{robot.turnTurnAround}->NO_2_TURNAROUND |
    robot.turnTurnAround->PREV_TURNAROUND ),
PREV_TURNAROUND = (robot.success->PREV_TURNAROUND2 |
    Alphabet\{robot.success}->NO_2_TURNAROUND),
PREV_TURNAROUND2 = (Alphabet\{robot.turnTurnAround}->NO_2_TURNAROUND ).
```

A continuación los flujos que describen la condición de bandeja de entrada llena (**INTRAY_FULL**), elemento agregado a la bandeja de salida (**ADDED_TO_OUTTRAY**) y bandeja de salida vacía (**OUTTRAY_EMPTY**) y el flujo que describe las acciones falibles (**F_FAILURES**).

```
//DEFINED FLUENTS
fluent INTRAY_FULL = <inTray.load, inTray.unload> initially 0
fluent ADDED_TO_OUTTRAY = <outTray.loaded, outTray.unload> initially 0
fluent OUTTRAY_EMPTY = <outTray.unload, outTray.loaded> initially 1
fluent F_FAILURES = <FailActions,Alphabet\{FailActions}>

//DEFINED ASSERTIONS
assert ASSUME_ON_LOAD = (INTRAY_FULL)
assert ASSUME_ON_UNLOAD = (OUTTRAY_EMPTY)
```

```
10 assert GOAL_FOR_UNLOAD = ADDED_TO_OUTTRAY
11 assert Failures = F_FAILURES
```

El controlador se define como la restricción del entorno a los objetivos definidos como conjunto de propiedades de seguridad y progreso (**safety**,**liveness**), el conjunto de presunciones (**assumption**), el de acciones controlables (**controllable**) y el conjunto de acciones de falla (**failure**).
Cabe notarse que el objetivo de progreso describe que con infinita frecuencia la bandeja de salida debe cargarse, ya que es el destino en el que esperamos pueda dejarse la carga, bajo la presunción de que será descargada con inifnita frecuencia podemos suponer que el sistema, en caso de conseguir un controlador, podrá llevar de forma contínua carga desde la bandeja de entrada a la bandeja de salida.

```
1 controller ||C = (Scenario)~{Objective}.
2
3 controllerSpec Objective = {
4   safety = {NO_2_LEFT, NO_2_RIGHT}
5   failure = {Failures}
6   assumption = {ASSUME_ON_LOAD, ASSUME_ON_UNLOAD}
7   liveness = {GOAL_FOR_UNLOAD}
8   controllable = {ControllableActions}
9 }
```

4. ENACTMENT, O ENTORNO DE EJECUCIÓN

Enactment debe entenderse como la capacidad de poner en ejecución un controlador. En el marco de este trabajo se refiere a ejecutar un controlador sintetizado a partir de la especificación que describe un juego SGR(1) sobre dominios falibles.
Se entiende que la puesta en ejecución o enactment es entre otras cosas un área a desarrollar y una actividad de conocimiento. En su carácter cognitivo permite observar y refinar los controladores construidos, las técnicas empleadas para construirlos y la forma en que éstos se ajustan a un contexto preexistente, potencialmente desacoplado a la unidad de cómputo que realiza la síntesis.

4.1. La necesidad de ejecutar estrategias

La forma natural de expresar un problema en la sintaxis de procesos finitos es describiendo el comportamiento de cada proceso como un sistema de transiciones etiquetadas. Suponiendo que de esta forma se expresa con suficiente detalle el comportamiento observable de un agente.
En la sintaxis propuesta se separan las transiciones controlables de las no-controlables. Se propone a partir de esto una separación entre los **mensajes de control**, que se corresponden con las transiciones controlables en la especificación y con los mensajes de entrada para el agente, de los **mensajes de estado** que se corresponden con las transiciones no controlables en la especificación y mensajes de salida en el agente. Un caso particular de las transiciones no controlables que puede echar más luz sobre la distinción es el de los timeouts o límites de tiempo. La transición controlable podría ser un mensaje de control o un pedido a un servicio particular y la transición no controlable podría ser un mensaje de estado, una respuesta por parte del servicio o un evento de timeout, que estaría capturando la falta de respuesta dentro de un período aceptable de tiempo. En esta tesis trabajamos con nodos controlables (donde todas las transiciones de salida son controlables) y no controlables (donde todas las transiciones de salida son no controlables). En un estado particular, no controlable, podría suceder que exista más de una transición de salida, una representando un mensaje de estado, otra representando un mensaje de timeout para expresar la posibilidad de que una máquina no responda en un lapso razonable de tiempo. Estos pequeños ejemplos introducen algunas nociones relacionadas al proceso de modelado, como pudo verse en los procesos escritos en el capítulo 3.3.

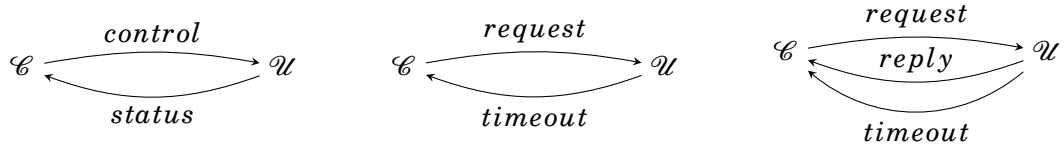


Fig. 4.1: Transición entre estados controlables y no controlables: control-estado (izq.), request-timeout (centro), request-timeout,reply(der.)

Se indentifica un subconjunto de las transiciones no controlables en relación al modelo

de fallas, indicando estados en los cuales una transición puede suceder pero con cierta noción de equidad. Estas transiciones del modelo de falla deben capturar un comportamiento no sistemático, independiente entre sí que va a dar lugar (eventualmente) a la ejecución de otra transición en ese mismo estado. En esta tesis la tripla $(try_i, suc_i, fail_i)$ permite representar la naturaleza falible pero no sistemática del movimiento del robot (se instancia como $(robot.follow, robot.destinationReached, robot.lost)$ o para el caso de las bandejas, donde un ítem puede caer al ser cargado o descargado como $(inTray.unload, inTray.unloaded, inTray.fell)$), que siguiendo una línea, puede perder rastro de la misma, pero ésto no se interpreta como un comportamiento potencialmente malicioso por parte del entorno.

4.2. El componente de ejecución: Enactor

El componente de ejecución debe respetar un esquema de comunicación asíncrona. Se espera que pueda alcanzarse un nivel de abstracción que permita que el control discreto sea suficiente para controlar y monitorear el comportamiento de los enactors.
Se puede pensar que cada enactor expone una API. Los métodos públicos serán equivalentes a los mensajes de control y los eventos disponibles equivalen a los mensajes de estado. Puede suceder que el vínculo entre un mensaje en el dominio de control discreto y su contraparte en el dominio real oculte información, por ejemplo, en nuestro caso vincularemos a través del enactor correspondiente los mensajes del protocolo de comunicación con transiciones del sistema discreto, cada vez que se envíe un mensaje de control (representado por la etiqueta correspondiente a la transición) se crea en el enactor un paquete conforme al protocolo de comunicación y se lo envía a través de la interfaz inalámbrica. Cada vez que la interfaz de comunicación recibe un mensaje del robot, el adaptador (enactor) lo interpreta y produce un evento que contiene la información relacionada a la transición no controlable (mensaje de estado).

En la figura 4.2 se representa la relación entre el modelo utilizado para construir la especificación del controlador y su implementación. $\mathscr{C}$ representa a los estados controlables en el modelo y $\mathscr{U}$ representa a los no controlables. c representa a la implementación del controlador en el entorno de ejecución y a representa al enactor en calidad de adaptador entre los mensajes del entorno de ejecución y los del dominio del componente (potencialmente) externo al entorno de ejecución. Se ve como las transiciones en el modelo se corresponden con los mensajes de control y estado en el entorno de ejecución.

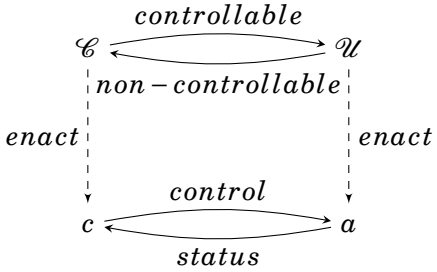


Fig. 4.2: Relación de ejecución entre el modelo y los componentes de ejecución

La relación entre la implementación en la tecnología correspondiente y el modelo se

muestra en la figura 4.3 representando al componente de ejecución como *e* para indicar que se está hablando de un objeto **Enactor** que se ejecuta dentro de la herramienta. Según la arquitectura escogida las transiciones controlables y no controlables se relacionan con llamadas a función y eventos respectivamente.

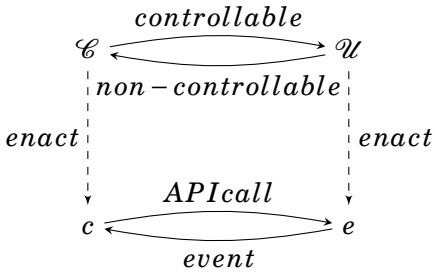


Fig. 4.3: Relación de ejecución entre el modelo y la implementación final de los componentes de ejecución

En la figura 4.5 se expande la interacción del **Enactor** con el componente externo, en este caso el cliente de comunicación del robot N6 (*r* en el esquema) para mostrar un flujo de mensajes de un entorno de ejecución en marcha. Aquí los mensajes de control se traducen en mensajes serializados que se envían a través del emisor de radio y los mensajes de estado se traducen a su vez de los mensajes serializados recibidos por la misma interfaz.

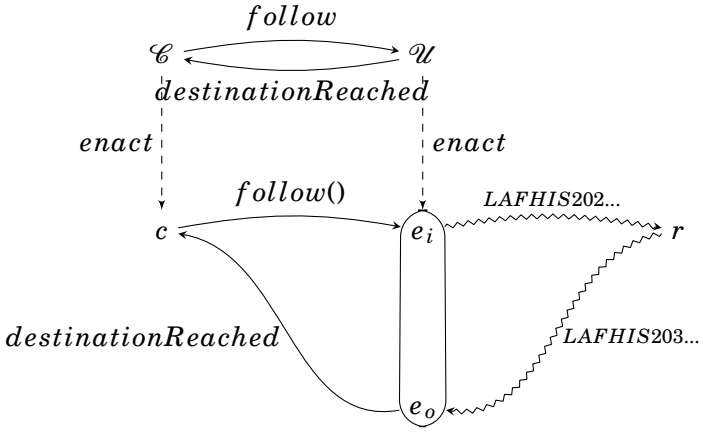


Fig. 4.4: Esquema de ejecución para el robot N6, vinculando los mensajes con el intercambio de mensajes serializados

Si se sustrae la comunicación por radio se puede exponer otra relación interesante entre las transiciones del modelo de eventos y el componente en ejecución. Se puede entender que el par $< control, status >$ encierra la ejecución de un bloque de control contínuo que eventualmente produce un mensaje de estado.

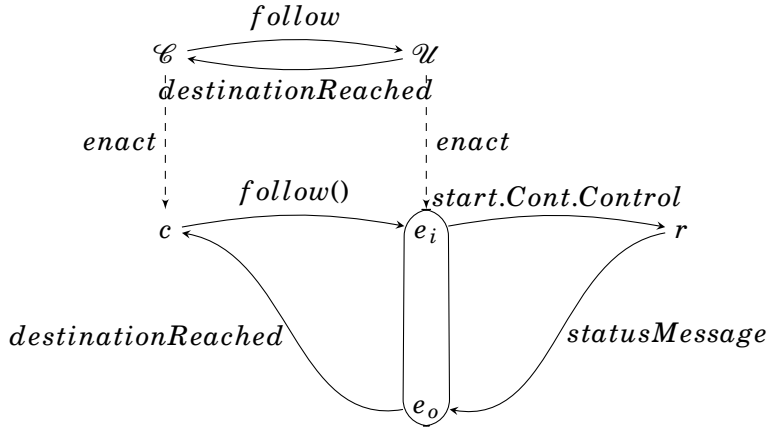


Fig. 4.5: Esquema de ejecución para el robot N6, vinculando los mensajes con la ejecución y reporte de estado de un bloque de control contínuo

Este pequeño ejemplo expone la adecuación de las técnicas de ejecución para esquemas de control contínuo y otros esquemas de control de sistemas embebidos que suelen utilizar máquinas de estado finito para representar su comportamiento (ya sea explícita o implícitamente).

4.3. Extensión de la herramienta

En el capítulo 6 se detallan las modificaciones necesarias para que un controlador coordine a los componentes de ejecución en un entorno físico. La intención es permitir ejecutar el controlador sintetizado a partir de la especificación dentro de la herramienta MTSA.

5. ROBOT MODELO N6 Y EXTENSIONES DE SOFTWARE/HARDWARE

Se espera que el robot cuente con un mecanismo que le permita transitar la planta en recorridos que visiten puntos distinguibles, que pueden representar las estaciones de carga, descarga y cruce de caminos. El desplazamiento autónomo se consigue utilizando una técnica de seguimiento de línea. Esto involucra:

- pintar las líneas correspondientes a los caminos en el piso de la planta
- pintar con un valor distintivo los cruces entre caminos
- brindar soporte de detección y seguimiento de líneas en el robot

El trazado de los caminos puede hacerse con pintura o con un cable eléctrico. En este trabajo se utilizará pintura o tinta negra para representar los caminos y sensores infrarrojos en el robot para poder detectar líneas en el piso.
El robot se desplazará tratando de mantener los sensores infrarrojos a ambos costados de la línea, si detecta una desviación significativa (diferencia de valor entre los sensores) compensará modificando la velocidad de rotación de cada rueda acorde a la magnitud del desvío.
En un intento de permitir recorridos variables en la grilla de caminos se introduce un valor intermedio para los cruces de forma que el robot pueda decidir de forma autónoma cuando se ha completado un tramo del camino.
La presencia de línea se percibe en el robot como un rango de lectura de un valor bajo en al menos en uno de los sensores. La presencia de un rango de valor alto se interpreta como consecuencia de que al menos uno de los sensores se encuentra sobre el mapa, pero fuera de la línea. Los valores del mapa en los cruces se interpreta al obtener una lectura en un rango intermedio de valores.

5.1. Arquitectura del robot N6

El modelo N6 es un robot producido en Argentina por la empresa RobotGroup con fines didácticos. Se trata de un robot planar con dos microrreductores operando cada rueda de forma independiente, dos fotosensores ubicados al frente y hacia la base, un sensor de ultrasonido y otro de radio ubicado en la parte superior.
La alimentación es autónoma a través de tres pilas AA y el controlador de los motores DC está integrado en la placa.

Es entregado con bibliotecas provistas por Atmel Corporation, Arduino y utilidades, configuración del controlador y bootloader de RobotGroup.

5.1.1. Arquitectura del procesador

El robot cuenta con un procesador ATMega32u4 producido por Atmel Corporation. Se trata de una arquitectura de 8 bits con un conjunto de instrucciones RISC operando sobre 32 registros de proposito general. La memoria de programa es de tipo flash con 32 KBytes de capacidad, la memoria volátil de datos es una SRAM de 2.5 KBytes y la memoria no volátil de datos es un EEPROM de 1 KByte.

Fig. 5.1: Vista de frente (izq.) y de perfil (der.) del robot modelo N6

Los registros están conectados directamente a la unidad aritmético lógica (ALU) permitiendo así acceder a dos registros en un mismo ciclo de reloj. Tiene 26 líneas de entrada/salida de propósito general, cuatro contadores/temporizadores y un contador de alta velocidad. Tiene una USART, un interfaz serie de dos vías orientada a bytes, 12 canales ADC de 10 bits, sensor de temperatura en el chip, un temporizador programable de vigía y una interfaz JTAG para permitir seguimiento de ejecución en el chip.

5.2. Extensiones físicas

Se extendió el soporte de hardware en el robot con una placa de comunicación XBee que permite la colaboración entre el robot y un nodo de control. La placa XBee se conecta a través de una de las interfaces serie y transmite la información de forma serial a otro nodo XBee.
El montaje de la placa requirió una mínima modificación en la estructura de la misma y un cableado entre uno de los puertos serie de la placa central y la placa de comunicación, como así también las líneas de alimentación desde la placa central a la placa XBee.

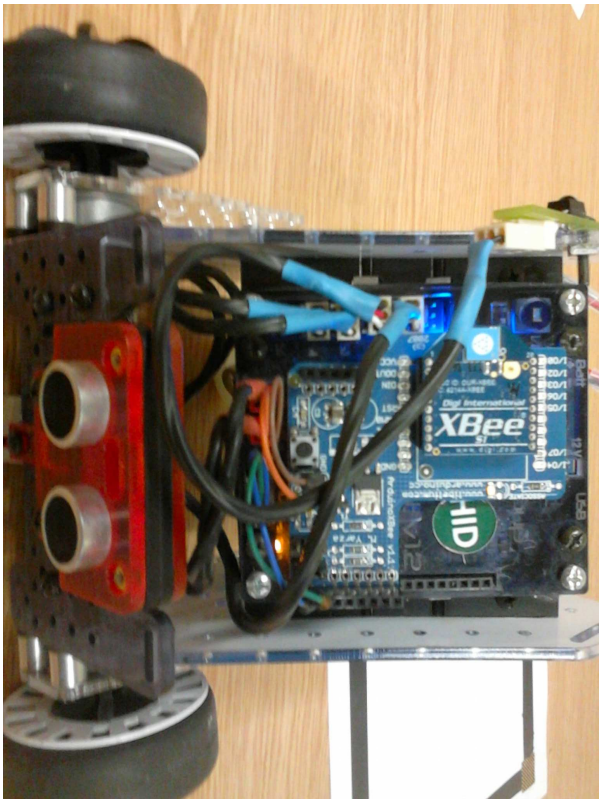


Fig. 5.2: Placa XBee adaptada al robot modelo N6

El nodo de control cuenta con otra placa de comunicación conectada a uno de sus puertos serie.

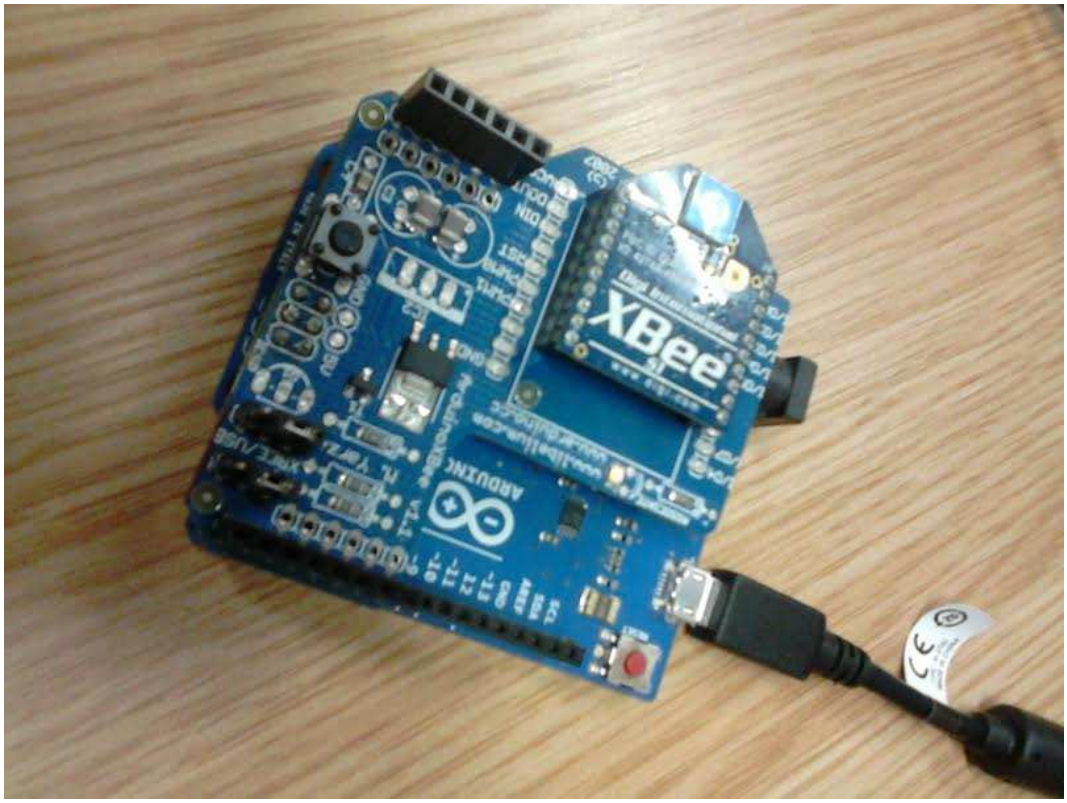


Fig. 5.3: Placa XBee conectada al puerto serie del nodo de control

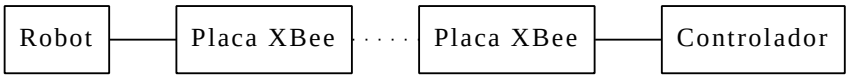


Fig. 5.4: Comunicación entre el robot y el controlador

5.3. Fuente de fallas

La arquitectura general del robot, comprendiendo las bibliotecas provistas, el soporte de hardware y la plataforma de software escrita para el mismo dejan, en su conjunto, lugar a fallas. Debe notarse que el control de velocidad en las ruedas es de lazo abierto, sin un mecanismo de retroalimentación que verifique el comportamiento consistente de los actuadores luego de ejecutada la logica de control. Por este motivo el resultado de ejecutar un giro a izquierda, derecha o una media vuelta es de naturaleza falible. No hay un mecanismo confiable para asegurar que el robot ha realizado un giro de aproximadamente noventa o ciento ochenta grados, estas acciones son aproximadas. Si bien puede entenderse como una falla de soporte o de arquitectura para el robot, un controlador sintetizado sobre un dominio falible, debe poder dar garantías de progreso aún frente a errores de este tipo. La terna $(robot.turnright, robot.success, robot.lost)$ caracteriza este tipo de errores.
Los bloques de control proveen un mecanismo básico de seguridad de forma que al terminar su ejecución detienen el avance de las ruedas. Esto asegura, que ante una falta de respuesta por parte del nodo de control, el robot se mantendrá inactivo y en el lugar.
Existe otra fuente de fallas, no intencional, en el protocolo de comunicación por la auscencia

de un control de sanidad o checksum. Esto se debe a que el protocolo se modeló sobre la base de un protocolo industrial que funciona sobre una capa de transporte que provee este control. En nuestro caso los paquetes se envían directamente sobre una capa de más bajo nivel, sin ningún control de este tipo.

5.4. Software

La funcionalidad requerida del robot según lo presentado debería permitir:

- Iniciar comunicación inalámbrica con un nodo de control
- Mantener el intercambio de mensajes con el nodo de control
- Cerrar la comunicación inalámbrica con un nodo de control
- Iniciar y mantener el seguimiento de una línea hasta alcanzar un cruce u obtener una lectura errónea (i.e. no se puede leer línea)
- Informar cuando se haya alcanzado un cruce y detenerse
- Informar cuando se haya alcanzado una lectura errónea y detenerse

El robot cuenta con un procesador `atmega32u4` que puede programarse en lenguaje ensamblador y que también cuenta con un compilador de `C` y `C++` (`avr-gcc`) y su respectivo debugger (`avr-gdb`).
Para este trabajo utilizamos como base las bibliotecas provistas por `RobotGroup` que incluyen una versión modificada de las bibliotecas provistas `Atmel Corporation` y la plataforma `Arduino`. Éstas exponen una interfaz de uso que abstrae la comunicación y configuración del hardware al nivel más bajo, programando también la memoria del procesador a través de los `fuses` y `lockbits`.
Utilizamos el `bootloader` provisto por `RobotGroup` que permite cargar el programa en la memoria del robot a través de una conexión `USB`.
En adelante se detallará la arquitectura del subsistema que satisface la funcionalidad mencionada anteriormente, que puede comprenderse como la lógica de control y comunicación del robot.

5.4.1. Arquitectura general

Se supone un funcionamiento en modalidad maestro-esclavo donde el robot recibe mensajes de control de un nodo central y envía a este mismo mensajes de estado. Una vez recibido un mensaje de control, el robot realiza de forma autónoma la tarea representada por ese mensaje y una vez finalizada envía otro mensaje informando el estado en el que se encuentra.
La asignación de responsabilidades a componentes se realiza de la siguiente manera.

N6 desacopla el acceso a los sensores y actuadores del robot a través de una semántica de mayor nivel, permitiendo acceder a los valores de lectura y escritura con descripciones relacionadas a la configuración actual, también es el responsable de asignar el procesador de forma ordenada a los otros componentes

MessageManager tiene conocimiento de las colas de comunicación y de los serializadores, deserializadores que permiten llevar un objeto de mensaje hacia y desde su representación serial, expone una semántica que permite preguntar si existe un mensaje en la cola, pedirlo apropiadamente como un objeto y enviarlo a un destinatario

Conjunto de mensajes cada mensaje es representado por un objeto y está relacionado a un serializador/deserializador que es responsable de llevarlo desde y hacia su representación serial

Conjunto de comandos el objeto de comando encapsula el comportamiento que el robot debe ejecutar en un momento dado, proveen una semántica de inicialización, ejecución y limpieza

La figura 5.5 intenta representar, de forma simplificada, la relación entre los componentes. El componente etiquetado como `N6` abstrae el acceso a los puertos de entrada y salida y se comunica con el controlador de mensajes para evaluar si hay algún mensaje de control esperando ser leído, o un mensaje de estado por enviar. Si, en efecto existe un mensaje de control que exige la ejecución de un comando, el comando referido es instanciado y puesto en ejecución.

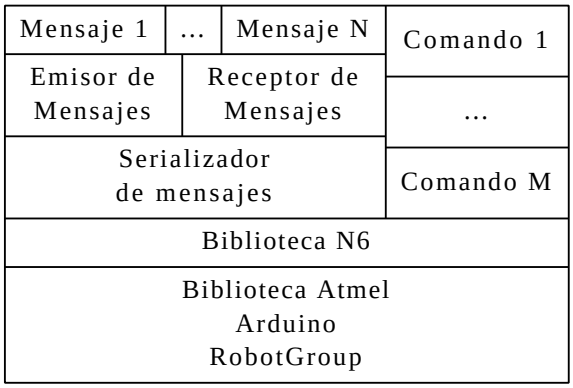


Fig. 5.5: Esquema general de la arquitectura del robot

Se analiza la recepción de un mensaje de control, su procesamiento y consecuente puesta en marcha del comando relacionado. Al describir esta interacción se describen parcialmente los objetos relacionados, junto con sus estructuras de representación, mensajes intercambiados y noción general de los algoritmos.

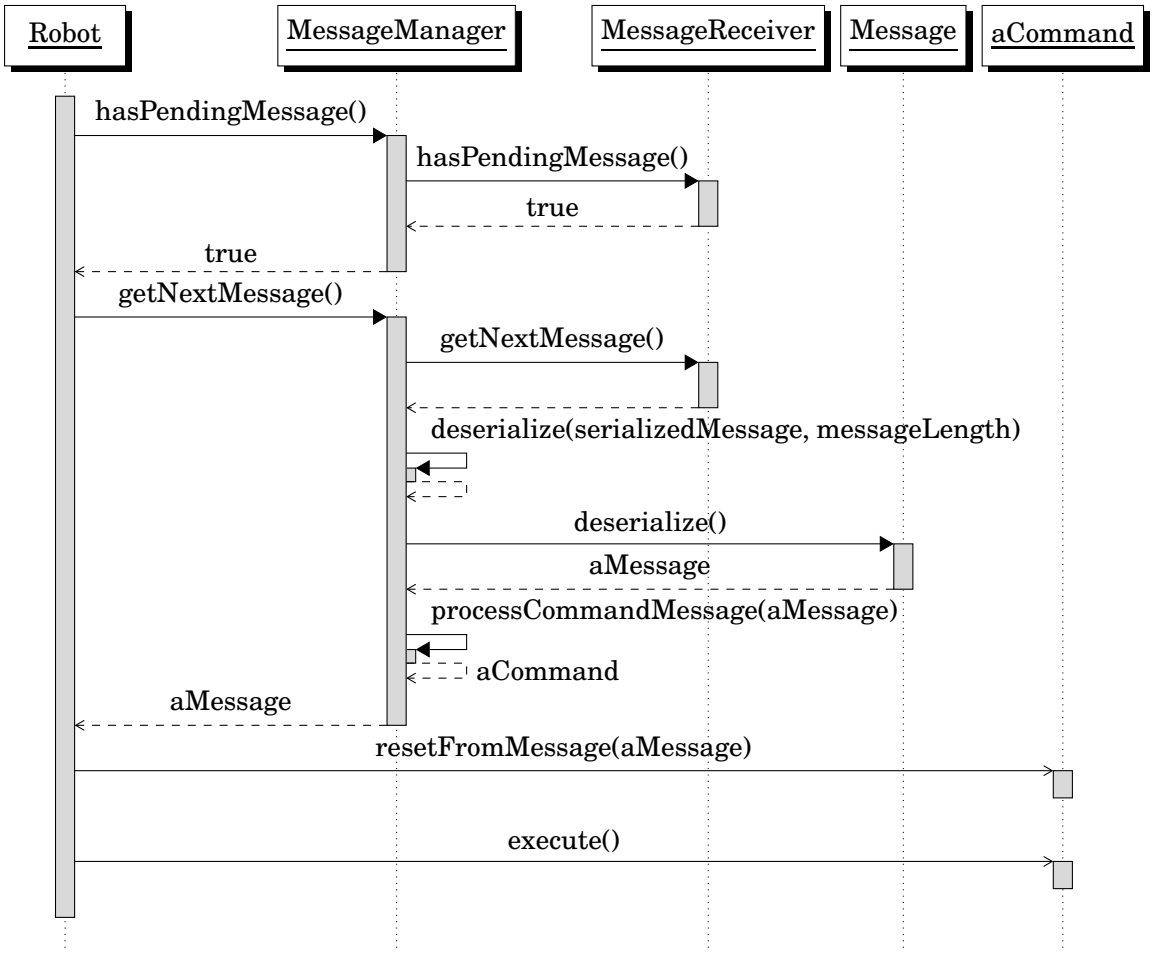


Fig. 5.6: Procesando un mensaje de control.

El punto de partida se define en el momento en que un nodo de control envía un mensaje serializado a través de la antena de emisión, la placa de radio del robot recibirá la información transmitida en forma secuencial y mantendrá una copia de la misma en su buffer.
El objeto que abstrae el software y cumple la función del scheduler (`N6`) recibe un mensaje de actualización (`update`) y envía un mensaje al controlador de mensajes (`MessageManager`) para evaluar si existe un mensaje esperando en la cola (`hasNextMessage`). El controlador envía el mensaje a quién éste conoce como receptor de mensajes (`MessageReceiver`). En verdad ésta clase y su contraparte el emisor de mensajes (`MessageSender`) son de tipo abstracto y en la práctica su funcionalidad es cubierta por una única implementación que recibe y envía mensajes a través de un puerto serie (`SerialSenderReceiver`). Si el envío del mensaje `hasNextMessage` desde `N6` da como resultado un valor verdadero, entonces se pide al `MessageManager` el próximo mensaje en cola a través de un envío del mensaje `getNextMessage`, que devuelve un objeto mensaje, involucrando a `MessageManager` que pide al objeto de clase del mensaje relacionado (e.g. `FollowLineMessage`) un objeto de mensaje materializado a través del mensaje `deserialize`. Entonces `N6` consigue una referencia al comando relacionado a través de una tabla que representa una función cuyo dominio es el de los identificadores de mensaje y con imagen en los comandos existentes.
El comando así relacionado recibe dos mensajes, primero el de `resetFromMessage` que le permite inicializar o limpiar estructuras y luego `execute` que ejecuta la lógica de control

del comando.
A continuación se detallan interacciones relevantes a este caso de estudio.
El objeto `MessageManager` contiene dos tablas relacionadas a la serialización y deserialización de un mensaje. La primera va a ser referida como mid y relaciona un código de identificación de mensaje con el índice de la función de serialización/deserialización que se encuentra referenciada con un puntero en la tabla que será referida como $serializers$.

$$mid(x) : int \rightarrow int, serializers : int \rightarrow (char* \rightarrow Message)$$

De forma que una vez leído el código de mensaje (que por otro lado tiene ubicación conocida dentro de la cadena), el mensaje puede conseguirse con:

$$serializers(mid(code))$$

La existencia de una tabla para resolver la relación entre el código de mensaje y el índice del serializador en otra tabla está motivada unicamente por las limitaciones de memoria en el dispositivo. De forma análoga se relacionan los códigos de mensaje de los mensajes de control con los comandos a ejecutar como respuesta.

5.4.2. Lógica de control

Como se ha visto cada mensaje de control se relaciona con un comando que representa el comportamiento esperado por parte del robot.
Si bien se ha dicho que el módulo `N6` cumple con la función de un scheduler, se debe aclarar que se trata de un scheduler de naturaleza colaborativa, donde cada proceso (en este caso comando) libera el procesador de forma activa, esto quiere decir que no se sigue una política de desalojo forzoso. Cada comando extiende la definición de la clase `Command` sobreescribiendo el método `update`. En cada ciclo de ejecución el componente `N6` envía éste mensaje a su comando activo, el comando a su vez ejecuta el ciclo de control (que se supone acotado de forma de no bloquear el comportamiento del robot en su totalidad) y libera al procesador saliendo de la función (`update`). Sí el comando considera que ha concluído su tarea o que ha alcanzado un estado de error marca su variable de estado como terminada, emite en caso de ser necesario el mensaje de estado correspondiente y concluye la ejecución de la función liberando así al procesador.

Resultado: true si la ejecución termina correctamente o se alcanza el objetivo, false si se alcanza un estado de error
ejecutar lógica de control;
si *se alcanza un estado de error* **entonces**
  enviar mensaje de error;
  marcar estado del comando como terminado;
  devolver false;
sinó, si *se consigue un estado objetivo* **entonces**
  enviar mensaje de confirmacion;
  marcar estado del comando como terminado;
  devolver
en otro caso
  devolver true y liberar el procesador;

Algoritmo 1: Esquema general de un comando

Debido a la limitada capacidad de memoria de programa y de datos del dispositivo se ha implementado un conjunto reducido de comando que permitan cubrir la funcionalidad esperada para el escenario propuesto. Se describe su funcionamiento en términos generales.

FollowLineCommand define un comportamiento en el que se utiliza la lectura de los sensores infrarrojos para mantener al robot centrado respecto de una línea negra que se pinta sobre el piso. Se utiliza la diferencia de valor entre los sensores para compensar la velocidad de los motores. Si se detecta un número de lecturas erróneas (por arriba del valor de negro esperado en ambos sensores) se envía un mensaje de error y se detiene la ejecución del comando, si se realiza una lectura de valor intermedio (lectura de valor definido en un rango inferior al rango de lectura de piso, blanco, y superior al rango de lectura de línea, negro) se envía un mensaje de éxito y se detiene la ejecución del comando

TurnCommand se controlan los motores del robot para realizar un giro de noventa grados a cualquiera de los lados o dar media vuelta, el comportamiento específico del comando depende del mensaje que lo haya iniciado, una vez relizado el giro se envía un mensaje de éxito y se detiene la ejecución del comando

SetMotorSpeedCommand se actualiza la velocidad de los motores con el valor recibido a través del mensaje y se detiene la ejecución del comando

5.4.3. Protocolo de comunicación

Para controlar de forma remota cada robot se diseña e implementa un reducido protocolo de comunicación. El control se hace intercambiando mensajes de control y estado. Los mensajes son de longitud variable y están divididos entre un encabezado de longitud fija y el contenido de datos (**payload**).
La comunicación se realiza entre un **iniciador** y un **receptor** como se muestra en el diagrama 5.7. El receptor acepta el mensaje de inicio de comunicación si no se encuentra intercambiando mensajes con otro nodo. El **iniciador** envía un mensaje de inicio **CommunicationStartMessage** que puede ser respondido con un **CommunicationStartAcknowledgeMessage** dándose inicio al intercambio de mensajes de control y estado o con un **CommandErrorMessage** indicando que no puede inicarse la comunicación con el **receptor** por un motivo especificado en el código de error.

Una vez que el intercambio de mensajes de control y estado finaliza el **iniciador** envía un mensaje **CommunicationStopMessage** y ambas partes consideran la comunicación cerrada y realizan las tareas de mantenimiento necesarias. Si las partes no están esperando respuesta de un mensaje enviado previamente deberían enviar regularmente (en un intervalo cercano a los $200ms$) un mensaje **KeepAliveMessage** para mantener abierto el canal. A continuación se describe el formato de mensaje y se da una descripción de los mensajes utilizados en este trabajo.

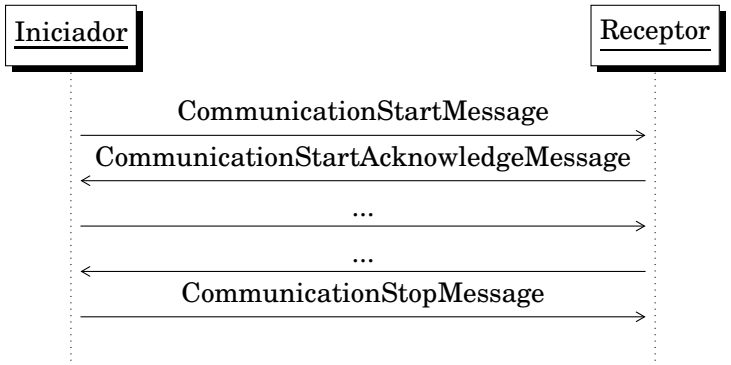
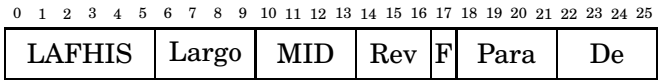


Fig. 5.7: Procesando un mensaje de control.

El paquete de mensaje es de longitud variable pero su cabecera es de longitud fija. El formato general se describe en la figura 5.8.



Campo	Descripición
`LAFHIS`	Encabezado de sincronización
`Largo`	Largo del mensaje expresado en bytes
`MID`	Código de identifiación del mensaje
`Rev`	Número de revisión del mensaje
`F`	Byte de máscara
`De`	Código del remitente del mensaje
`Para`	Código del destinatario del mensaje

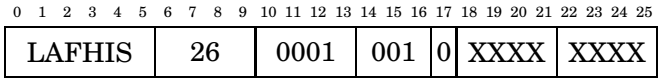
Fig. 5.8: Formato general del paquete de comunicación

Cabe aclarar de los campos descriptos algunos detalles: el campo de *Revisión* permite extender el protocolo con nuevas versiones respetando la retrocompatibilidad, el campo de *Flag* o máscara permitiría marca en un futuro la posibilidad de enviar una respuesta a modo de *piggybacking*.
Si una cola de recepción, por ejemplo en la placa de comunicación del robot, contiene una cadena de cualquier longitud que no contiene el encabezado de sincronización, ésta se descarta y se continúa leyendo hasta encontrarlo. Una vez ubicado se lee hasta la posición del valor de `Largo` del mensaje para saber si se han recibido suficientes bytes como para formar el mensaje. De no ser así se contínua leyendo. Se aprovecha también que el código de mensaje se encuentra en una posición conocida para buscar a partir de éste al deserializador

correspondiente.

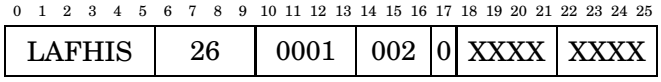
5.4.3.1. 001.CommunicationStartMessage

Mensaje de inicio de comunicación. Es enviado por el iniciador para comenzar la comunicación con el nodo receptor, quien puede recibir como respuesta un mensaje **CommunicationStartAcknowledgeMessage** (5.4.3.2) o un mensaje **CommandError**(5.4.3.4)



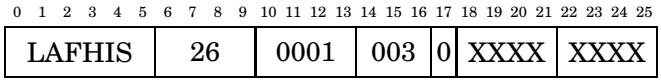
5.4.3.2. 002.CommunicationStartAcknowledgeMessage

Mensaje de confirmación al mensaje de inicio de comunicación. Es enviado por el receptor como confirmación del inicio de la comunicación.



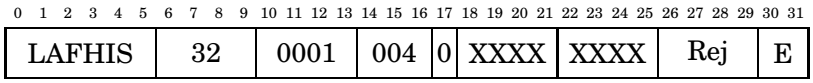
5.4.3.3. 003.CommunicationStopMessage

Mensaje de fin de comunicación. Es enviado por el iniciador para indicar que se finaliza la comunicación.



5.4.3.4. 004.CommandErrorMessage

Mensaje de error en respuesta a un mensaje recibido.



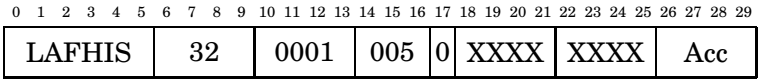
Campo	Descripición
`Rej`	Código del mensaje rechazado
`E`	Código de error indicando el motivo por el cual se ha rechazado el mensaje

El significado del código de error se explica en la tabla 5.4.3.4.

Código	Descripición
-2	El largo del mensaje no se corresponde con el valor provisto
-3	No se encuentra el caracter de fin de mensaje
-4	Se desconoce el código de mensaje (MID)
-5	Mensaje rechazado, puede deberse a que el receptor se encuentra en comunicación con otro iniciador
-6	El nodo receptor no tiene ninguna comunicación en curso
-2	Se deconoce el código de comando

5.4.3.5. 005.CommandAcceptedMessage

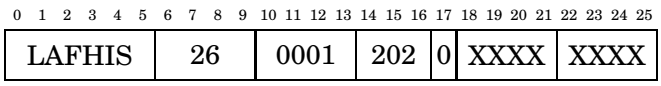
Mensaje de confirmación en respuesta a un mensaje de control.



Campo	Descripición
Acc	Código del mensaje aceptado

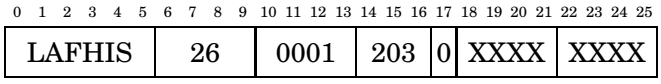
5.4.3.6. 202.FollowLineMessage

Mensaje de control que indica que debe iniciarse un comando de seguimiento de línea. Puede ser respondido por un **CommandAcceptedMessage**(5.4.3.5) o con un **CommandErrorMessage**(5.4.3.4)



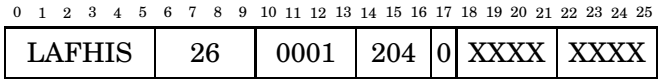
5.4.3.7. 203.DestinationReachedMessage

Mensaje de estado que indica que el robot ha encontrado un nodo de intersección durante la ejecución de un comando.



5.4.3.8. 204.RobotLostMessage

Mensaje de estado que indica que el robot no ha podido realizar una lectura válida o por algún motivo no puede determinar su ubicación.



5.4.3.9. 205.TurnLeftMessage

Mensaje de control que indica que debe iniciarse un comando de giro a la izquierda. Puede ser respondido por un **CommandAcceptedMessage**(5.4.3.5) o con un **CommandErrorMessage**(5.4.3.4)

0 1 2 3 4 5	6 7 8 9	10 11 12 13	14 15 16	17	18 19 20 21	22 23 24 25
LAFHIS	26	0001	205	0	XXXX	XXXX

5.4.3.10. 206.TurnRightMessage

Mensaje de control que indica que debe iniciarse un comando de giro a la derecha. Puede ser respondido por un **CommandAcceptedMessage**(5.4.3.5) o con un **CommandErrorMessage**(5.4.3.4)

0 1 2 3 4 5	6 7 8 9	10 11 12 13	14 15 16	17	18 19 20 21	22 23 24 25
LAFHIS	26	0001	206	0	XXXX	XXXX

5.4.3.11. 207.TurnAroundMessage

Mensaje de control que indica que debe iniciarse un comando de medio giro. Puede ser respondido por un **CommandAcceptedMessage**(5.4.3.5) o con un **CommandErrorMessage**(5.4.3.4)

0 1 2 3 4 5	6 7 8 9	10 11 12 13	14 15 16	17	18 19 20 21	22 23 24 25
LAFHIS	26	0001	207	0	XXXX	XXXX

5.4.3.12. 911.KeepAliveMessage

Mensaje enviado entre las partes para mantener el canal de comunicación abierto.

0 1 2 3 4 5	6 7 8 9	10 11 12 13	14 15 16	17	18 19 20 21	22 23 24 25
LAFHIS	26	0001	911	0	XXXX	XXXX

Para este trabajo el protocolo se ha implementado una versión en lenguaje **C++** para el nodo de recepción, que corresponde al robot **N6** y una versión en **Java** para el nodo de control, que corresponde a la máquina que ejecuta el entorno de enactment.

5.5. Dificultades y lecciones aprendidas

El objetivo de esta tesis es probar la adecuación de las técnicas de modelado y síntesis antes mencionadas en un contexto industrial que hace uso de una tecnología particular, el robot modelo N6. Si bien se intenta consumir la teoría relacionada con este tipo de controladores y las técnicas para producirlos, lo que se podría llamar el costado implementativo fue la principal fuente de dificultades. La arquitectura limitada, principalmente en lo que toca a la **memoria principal** del procesador del robot, (debe recordarse que se trata de 2.5kbytes

para almacenar todos las estructuras vivas para el robot) inició un trabajo de escritura y reescritura del código de control y comunicación para evitar un desborde de la memoria. Esto limita por un lado la cantidad de bloques de control que pueden ejecutarse en un momento (se limita por diseño a solamente uno) y a la cantidad de lo que podría llamarse factorías de comandos y mensajes del protocolo de comunicación, ya que cada uno de éstos registra una función de inicialización o deserialización/serialización en la estructura principal del robot. El tamaño de la **memoria de programa y datos** (son espacios separados en esta arquitectura) también limita de forma bastante similar la funcionalidad que el robot puede exponer. Probablemente se podría haber tomado un enfoque más estructurado, que evite la sobrecarga de memoria que se produce como resultado de la división de responsabilidades en clases, con macros y un uso más intensivo de punteros a función. Al momento en que éstas dificultades se volvieron más evidentes el trabajo sobre la lógica del robot se encontraba muy avanzado y la solución práctica fue reducir la cantidad de mensajes del protocolo de comunicación, la longitud de los mismos y deshacerse de lógicas de control no fundamentales.
La falta de un **emulador competente** o la posibilidad de realizar **on chip debugging** implicó que la tarea de depuración y seguimiento se realice a partir de un método de punto medio. Esto quiere decir que cuando se detectaba una anomalía, la forma de ubicar el código defectuoso (salvando el caso de una sobrecarga de memoria que suele producir reseteos recurrentes del procesador) era dar una señal sonora (utilizando el buzzer embebido en el robot) para indicar que se había superado cierta porción de código con éxito, por ejemplo, si se parte el programa en dos, y se coloca la llamada al buzzer luego de la mitad del código puede suponerse (salvo situaciones que involucren la concurrencia de más de una condición en el error) que la primera mitad fue ejecutada con éxito, luego se desplaza la llamada al buzzer a la mitad de la porción de código que aún no se ha probado, si no se llega a ejecutar la llamada (no se oye el pitido) se sabe que hay al menos un error en la primera mitad de la segunda porción del programa, así se sigue dividiendo y refinando.
En relación a la comunicación inalámbrica, el código original del robot, tal como es entregado por RobotGroup, hace uso de una versión de la biblioteca de Arduino en la que el puerto serie es implementado con un **único buffer**, esto quiere decir que se utiliza una misma estructura interna para almacenar los datos leídos y los escritos, de forma tal que si se escribe un valor mientras se encontraba aún leyendo, se pisa el valor anterior desde el puntero al fin del buffer circular en adelante. La solución frente a esto es forzar lecto/escrituras excluyentes, de forma que no se corrompan los datos que se encuentran en el buffer del puerto serie.
La idea de utilizar el mismo par de sensores para el seguimiento de línea y la detección de puntos de intersección introduce un problema generado por la aparición de un **tercer valor de lectura**. Los robots que realizan seguimiento de línea suelen discriminar solamente entre un valor de línea y uno de base, de forma que el espacio de valores puede dividirse facilmente, incluso si hace falta reservar bandas de valor para las lecturas inciertas (esto quiere decir, valores para los cuales no es posible determinar uno de los dos valores). Con la aparición de un tercer valor en un rango que se encuentra entre los valores de negro pleno y blanco pleno, los errores introducidos en la lectura se vuelven mucho más significativos, ya que los rangos sobre los cuales se decide si un valor es negro, blanco o gris son mucho más angostos. La lectura del valor es de naturaleza errática, depende de la intensidad de luz ambiente, de su temperatura, de la distancia de los sensores a la banda y de la calidad de impresión del mapa. Incluso el desgaste producido por las sucesivas pasadas del robot sobre el mapa se vuelve mucho más evidente a la hora de realizar estas lecturas trivaluadas. Si

bien el problema sigue presente, se lo ha mitigado con un mecanismo sencillo bajo el cual una lectura se determina por ejemplo negra, si se han realizado varias lecturas sucesivas con el mismo valor.
Si bien han surgido varios inconvenientes de menor magnitud, se evidencia las dificultades asociadas a trabajar con un caso de estudio que involucra varias arquitecturas y en particular una con un soporte tan reducido, al menos si se lo compara con tecnologías más habituales o incluso más robustas. Si bien se podría pensar en extender el alcance del modelado y la síntesis hasta las capas de más bajo nivel para intentar satisfacer ciertas propiedades deseables en este sub-sistema, parece más interesante atacar las limitaciones desde el modelado de alto nivel, por ejemplo conteniendo los errores emergentes en la ejecución con las herramientas provistas por la extensión de dominios falibles o el modelado de un proceso de reubicación manual.
Se evidencia también la dificultad de llevar adelante, y a la par, un trabajo de implementación del soporte físico (se entiende por esto la programación del código que se ejecuta en el robot) junto al de modelado y síntesis. Probablemente la interacción con un especialista o un grupo, con conocimiento específico de dominio y experiencia previa en la arquitectura de destino, permita una exploración más ágil y provechosa de la problemática de la ejecución de estrategias sobre un soporte de hardware/software particular.

6. EXTENSIÓN DE MTSA

Se presenta el trabajo realizado para ejecutar los controladores sintetizados a través de la herramienta **MTSA**, una caracterización de los agentes de ejecución (*Enactor*) y ejemplos de adaptadores escritos siguiendo estas guías.

6.1. Framework de enactment

6.2. Adaptadores

Se discute la propuesta e implementación del componente de ejecución con intención de integrarlo a la herramienta de modelado, chequeo y síntesis. El lenguaje en el que se escribe el componente es **Java**.
Para cada componente de ejecución se escribe un adaptador que respeta la arquitectura Ya que la respuesta de un agente en el marco de una ejecución es potencialmente asíncrona, i.e. en el caso en que se envía un mensaje y se espera uno de varios mensajes en algún momento futuro, se decidió en la implementación del componente de ejecución relajar el esquema de intercambio de mensajes haciendo uso de eventos. Se presentan inicialmente dos objetos, el de recepción de eventos y su contraparte de emisión, luego el componente de ejecución será una composición natural de ambos, permitiendo tanto recibir mensajes (representando la situación en la que se envía un mensaje de control) o emitiéndolos a través de eventos (mensajes de estado).

```
public abstract class TransitionDispatcher<Action> {
        public synchronized void
            addTransitionEventListener(ITransitionEventListener<Action> listener){
                ...
        }

        protected void fireTransitionEvent(Action action) throws Exception{
                ...
        }
}
```

Fig. 6.1: Objecto de emisión de mensajes (**TransitionDispatcher**)

En la figura 6.1 se presenta una simplificación del objeto de emisión que muestra basicamente el mecanismo por el cual un colaborador que implemente la interfaz **ITransitionEventListener** puede registrarse para recibir los mensajes de un enactor (**addTransitionEventListener**), y la forma en la que el enactor envía un mensaje a sus suscriptores (**fireTransitionEVent**).
Se abstrae el tipo de datos de la acción (que representa una transición en el sistema de transiciones etiquetadas), y puede entenderse por el momento que el mismo es una cadena.

```
public interface ITransitionEventListener<Action> {
       public void handleTransitionEvent(TransitionEvent<Action>
           transitionEvent) throws Exception ;
}
```

Fig. 6.2: Objecto de recepción de mensajes (**ITransitionEventListener**)

En la figura 6.2 se presenta una simplificación del objeto de recepción de mensajes que muestra basicamente el mecanismo por el cual un colaborador se declara como receptor de mensajes (**handleTransitionEvent**). En conjunción con el mecanismo de registro para un emisor (**addTransitionEventListener**) se cubre la funcionalidad necesaria para permitir el pasaje de mensajes entre agentes de ejecución.

6.3. Esquema de control

A modo de prueba, y en respuesta a la necesidad de seleccionar la acción controlable en cada estado controlable para una ejecución particular, se define un componente **Controller**. Se trata de una especificación del tipo **Enactor** con un método abstracto *takeNextAction* donde se define una lógica para seleccionar la transición a tomar dentro de las habilitadas para el estado. A la hora de realizar este trabajo se han implementado el esquema **TakeFirstController** que toma la primera transición según el orden definido por la estructura interna de representación y **RandomController** que toma una transición controlable de forma aleatoria.

6.4. Ejecutando el controlador

En la sección 3.3 fueron presentados los procesos que describían a cada componente en la sintaxis extendida de FSP. Los componentes disponibles para la ejecución son implementados en el paquete `ar.uba.dc.lafhis.enactment`, luego la instanciación de adaptadores disponibles se describe como entradas en el archivo de configuración `context.xml`. A continuación se presenta un extracto del mismo.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">
   <context:annotation-config/>
        <!-- Enactors -->
        <bean id="N6" class="ar.uba.dc.lafhis.enactment.robot.N6Robot">
          <constructor-arg index="0" value="N6 Robot"/> <!-- Name-->
          <constructor-arg index="1" value="robot.success"/>
          <constructor-arg index="2" value="robot.fail"/>
          <constructor-arg index="3" value="robot.lost"/>
          <constructor-arg index="4" value="robot.follow"/>
```

```
        <constructor-arg index="5" value="robot.turnLeft"/>
        <constructor-arg index="6" value="robot.turnRight"/>
        <constructor-arg index="7" value="robot.turnAround"/>
        <constructor-arg index="8" value="robot.readSensors"/>
        <constructor-arg index="9" value="robot.readSensorsReply"/>
        <constructor-arg index="10" value="robot.readUSSensor"/>
        <constructor-arg index="11" value="robot.readUSSensorReply"/>
        <constructor-arg index="12" value="robot.retry"/>
        <constructor-arg index="13" value="robot.successRetry"/>
    </bean>
    <!--
            ...
     -->
    <!-- Enactor Factory -->
    <bean id="enactorFactory" class="enactment.EnactorFactory"/>
    <!-- Controller Strategies-->
    <bean id="controllerScheduler"
        class="ar.uba.dc.lafhis.enactment.TakeFirstController">
            <constructor-arg index="0" value="Take First Controller"/>
    </bean>
    <bean id="randomScheduler"
        class="ar.uba.dc.lafhis.enactment.RandomController">
            <constructor-arg index="0" value="Random Controller"/>
    </bean>
    <!-- Scheduler Factory -->
    <bean id="schedullerFactory" class="enactment.SchedulerFactory"/>
</beans>
```

Las entradas del archivo de configuración describen los objetos que podrán instanciarse en la ejecución, sus parámetros tienen relación directo con los parámetros de entrada de sus constructores.
En el curso de una ejecución los pasos a seguir serían éstos:

Especificación Se declara la especificación en la sintaxis ya presentada, tanto de los procesos, propiedades y definición del controlador

Síntesis Se sintetiza el controlador

Configuración Se seleccionan los adaptadores y el esquema de control pertinentes a la ejecución (`Enactment->Options` en la herramienta)

Ejecución Se pone en marcha la ejecución (`Enactment->Run Model` en la herramienta)

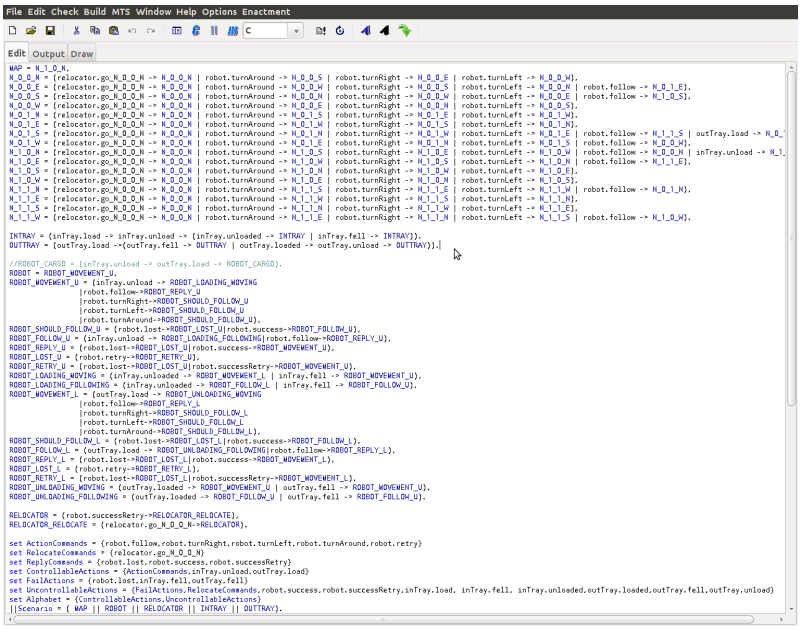


Fig. 6.3: Describiendo los procesos, propiedades y especificación del controlador en MTSA

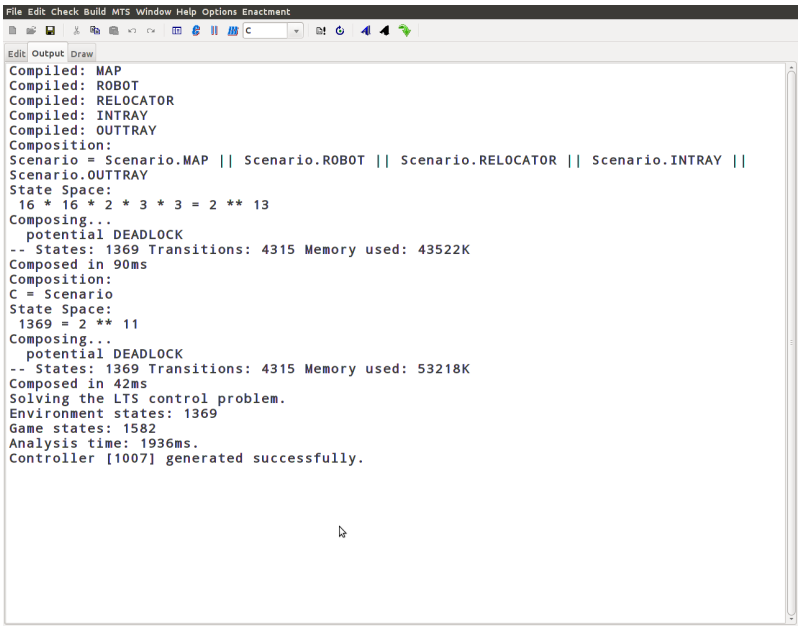


Fig. 6.4: Sintetizando el controlador en MTSA

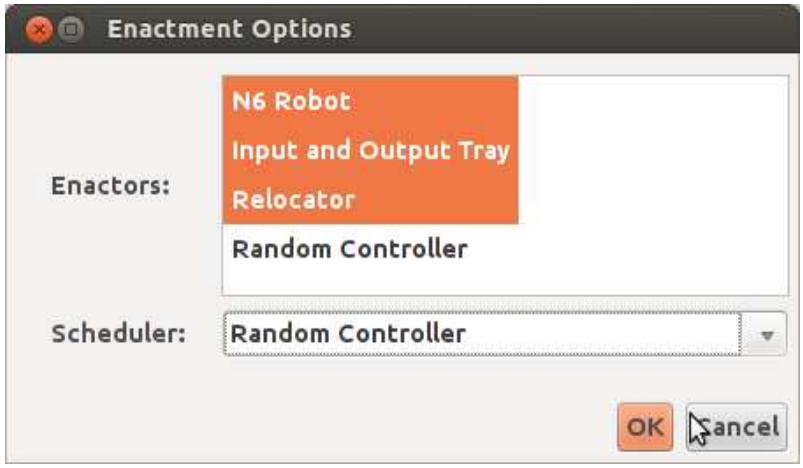


Fig. 6.5: Seleccionando los adaptadores y el equema de control

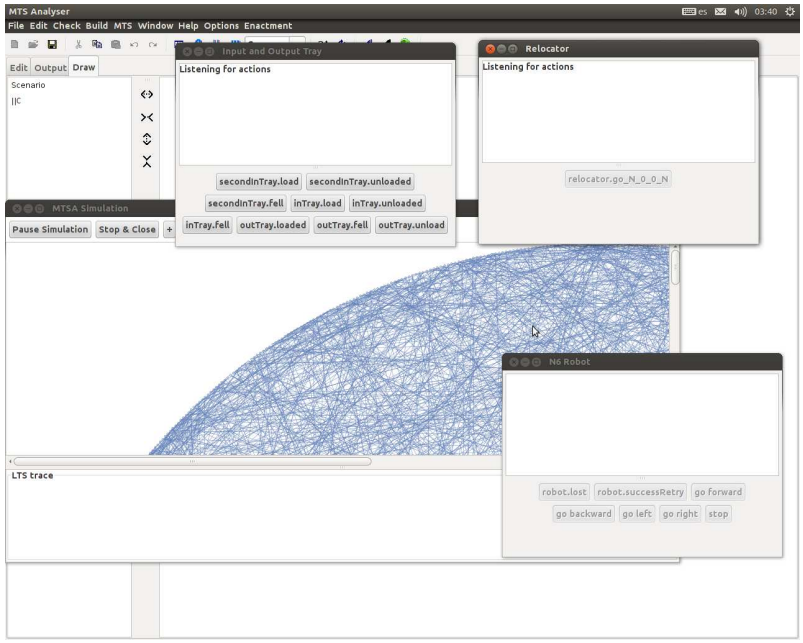


Fig. 6.6: Ejecutando el controlador

7. EVALUACIÓN

Contando con la extensión de la herramienta MTSA, el soporte de hardware/software en el robot N6, la definición y confección del mapa y las bandejas, y de los adaptadores y esquemas de control necesarios, resta definir las dimensiones de cambio sobre las que se ha de evaluar la adecuación de las técnicas y los individuos que han de representar una posición en el espacio de pruebas.

7.1. Casos de estudio

Se definen seis casos distintos sobre un mapa rectangular de cuatro nodos, un robot posicionado en la esquina del nodo $(1,0,N)$ que significa, la segunda fila, primera columna mirando al norte, la bandeja de entrada se encuentra habilitada en $(1,0,N)$ y la de salida en $(0,1,S)$. Las diferencias entre los casos se detallan a continuación:

Caso A el robot debe llevar carga de la bandeja de entrada a la de salida con infinita frecuencia

Caso B como el caso A pero con la conexión entre $(1,0)$ y $(1,1)$ bloqueada por un obstáculo

Caso C como el caso B pero con la restricción de que el robot no puede hacer dos giros a izquierda ni dos giros a derecha como restricción de seguridad

Caso D como el caso A pero con la restricción de que el robot no puede hacer dos giros a izquierda como restricción del proceso que describe el movimiento del robot

Caso E como el caso A pero también puede tomar carga en $(0,0,S)$

7.2. Cambios de configuración

La idea es presentar cambios en la descripción del problema e interpretarlos como cambios de configuración, para mostrar como la técnica permite sintetizar controladores frente a la eventualidad de un cambio sin precisar un trabajo adicional por debajo de la especificación. Obviamente que, en el caso en que se precise funcionalidad no provista por el conjunto de máquinas disponibles, esto pierde validez y se deberá implementar la funcionalidad de bajo nivel, el adaptador y extender la configuración del entorno de ejecución.

7.2.1. Causas de cambio de configuración

Los cambios de configuración son de carácter heterogéneo, tanto en su descripción como en sus causas. Pueden ser motivados por un cambio funcional o un refinamiento del dominio, ya que, por ejemplo al poner en ejecución una estrategia, se gana conocimiento del sistema y de la interacción de las partes. Como consecuencia de ésto se enriquece el razonamiento y esto puede resultar en una modificación en la descripción de los procesos o las propiedades de sistema.
Uno podría por ejemplo desear restringir, como es el caso, la ejecución de dos giros consecutivos a izquierda o derecha al observar que éstos presentan una mayor probabilidad de falla

que si se ejecutan de forma aislada. O podría suceder que, ante un cambio en el entorno (se agrega una bandeja de entrada) se deba redefinir él o los procesos que describe o describen la parte afectada.

7.2.2. Resultados

A modo de referencia se presentan algunos datos cuantitativos de los controladores sintetizados para los casos A hasta E. Las pruebas se ejecutaron en una máquina *Intel® Core™2 Duo CPU E8500 @ 3.16GHz × 2* con 3.8Gb de memoria principal con un sistema operativo *Ubuntu Release 12.04 64-bit Kernel Linux 3.2.0-60-generic*. Los controladores fueron sintetizados, sus estrategias ejecutadas y para cada una se ha creado un archivo en disco para la traza de ejecución, un video a través de una cámara que ha capturado la ejecución sobre el mapa, un archivo con el controlador resultante y otro archivo con los datos de tiempo de procesamiento y demás valores relacionados con el proceso de síntesis.
Si bien se había planeado extender esta prueba incrementando el tamaño del mapa por potencias de dos, esto resultó impracticable ya que rápidamente los controladores eran imposibles de conseguir por falta de memoria o por cotas temporales en el proceso de síntesis.

Caso	#Estados entorno	#Trans. entorno	Memoria Usada	Tiempo composición	#Estados juego	Tiempo síntesis	#Estados controlador
A	1369	4315	21696K	162ms	1582	1813ms	1007
B	1297	4039	57341K	5ms	1510	1176ms	763
C	1989	6037	22061K	147ms	2202	1655ms	815
D	1036	3338	36109K	105ms	1241	333ms	1241
E	2758	10079	27354K	166ms	1582	1813ms	1007

Se puede ver que para el resto de los casos los resultados son bastante similares.
Durante las ejecuciones se ha comprobado que en tanto las presunciones valgan, el sistema satisface los objetivos, aún frente al dominio falible. Esto se traduce informalmente en lo siguiente: en tanto las presunciones no valgan (no hay carga en la bandeja de entrada) el controlador libera al robot para que se mueva con libertad por el mapa, pero en cuanto aparece carga en la bandeja de entrada, el robot se dirige a ésta, intenta cargar y llevar directamente a la bandeja de salida. En el caso de la falla por pérdida del ítem, vuelve al comportamiento antes descripto, salvo que haya caído al descargar en la bandeja de salida siendo que la bandeja de entrada ya se ha cargado nuevamente, en cuyo caso vuelve a recoger carga de la bandeja de entrada. En el caso de que el robot se pierda siguiendo una línea o girando, se dispara el mecanismo de reubicación (en este caso manual que implica colocar al robot en la posición $(0,0,N)$) y a partir de ésto el robot continua con el comportamiento anterior, esto es, si se encuentra cargado intentará descargar en la bandeja de salida, caso contrario esperará a que aparezca carga o irá a buscarla a la bandeja de entrada.

7.3. Discusión

En el curso de esta tesis se ha intentado probar la adecuación de las técnicas de modelado y síntesis al caso de una línea de distribución que hace uso de un robot modelo N6. Esto ha motivado la escritura de código tanto para el robot como para la extensión de la herramienta, los adaptadores y las especificaciones correspondientes, dando lugar a un trabajo desplegado a lo ancho entre varios componentes y en diverso grado de abstracción.

La interacción con el mundo físico a través de los sensores y actuadores (en este caso concentrados en el robot) ha sido fuente de refinamiento del control de bajo nivel y ha motivado el uso de la extensión de dominios falibles a la hora de modelar y sintetizar. Por otro lado se ha prestado como campo de prueba sobre la forma de describir los procesos, las propiedades de sistema y en general los modos de relacionar el modelo con los componentes de ejecución.
La puesta en marcha de las estrategias o controladores da muestra de forma perceptiva de su calidad, cabe preguntarse por ejemplo si es correcto que un robot recorra el mapa sin mayor criterio sólo porque las presunciones no valen, luego no es necesario satisfacer el objetivo. Esto podría (si llegase a probarse útil y correcto), motivar el uso de una transición de no operación en el proceso que describe al robot, de forma que un esquema de ejecución pueda, siendo que se encuentra habilitada en un nodo controlable, tomar ésta por sobre las otras.
El uso de transiciones falibles ha demostrado ser muy provechoso para describir las situaciones donde una operación puede fallar de forma no sistemática como se ha dicho anteriormente, en particular para el movimiento del robot, que efectivamente falla. No se trata de una falla simulada como la de las bandejas sino una falla real causada por errores de lectura y condiciones irregulares en el entorno.
La extensión de la herramienta para soportar la ejecución de las estrategias convierte a la herramienta en un entorno de trabajo, donde no sólo se modela un problema sino que se construye un controlador que permite satisfacer los requerimientos y se lo pone en ejecución para validar que la ejecución muestra es conforme a lo esperado.

7.4. Conclusiones

Si bien se cree que el objetivo inicial está satisfecho se sustraen varias ideas o conclusiones de esta tesis. Por un lado se probó la adecuación de la técnica a entornos afine a aquellos que hacen uso de un robot planar en una planta de distribución. Cabe preguntarse qué características se espera que tenga un problema para validar la adecuación de una parte de la teoría o de las herramientas. Por ejemplo, parece evidente que para el caso de estudio de esta tesis el dominio falible no es sólo conveniente sino que necesario. Dificílmente puedan sintetizarse controladores sin esta extensión.
Se podría realizar un relevamiento sobre objetivos de sistema para problemas de índole industrial o relacionados al uso de robots, para validar que estos objetivos expresados en términos de un juego SGR(1) sobre dominios falibles sean suficientes.
El entorno de ejecución se encuentra hoy día acoplado a la herramienta de modelado y síntesis, pero podría sin mayor inconveniente existir independientemente de ésta. Esto permitiría desacoplar el proceso de modelado y síntesis del de ejecución, mejorando tiempos para pruebas repetitivas de un controlador, dando la posibilidad de ejecutar un controlador en otro nodo físico, o incluso distribuyéndolo entre varios nodos.
La característica regular del mapa motiva una noción (intuitiva) de que ciertas estructuras deberían permitir reducir el espacio sobre el cual se construye el controlador.
Hay un punto particular que sería interesante modelar a futuro. La motivación es esta: si por alguna falla sistemática en el canal de comuniaciones el porcentaje de mensajes erróneos comienza a aumentar, y los reintentos son infrutuosos, se podría incorporar un mecanismo de timeout que lleve al robot a un estado estable. Probablemente haciendo uso del mecanismo de reubicación y enviando un mensaje que restituya el estado lógico del ro-

bot a la configuración inicial. En particular sería un escenario interesante para probar un mecanismo de timeout y recuperación, que parece justificado en entornos industriales.

Bibliografía

[1] D. Fischbein, N. R. D'Ippolito, G. Brunet, M. Chechik, and S. Uchitel. Weak alphabet merging of partial behaviour models. In *IEEE Transactions on software engineering and Methodology*, to appear, New York, NY, USA, 2010. ACM.

[2] 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.

[3] M. Jackson. *Software requirements & specifications: a lexicon of practice, principles and prejudices*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[4] M. Jackson. The world and the machine. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 283–292, New York, NY, USA, 1995. ACM.

[5] R. Kazhamiakin, M. Pistore, and M. Roveri. Formal verification of requirements using spin: A case study on web services. In *Proceedings of the Software Engineering and Formal Methods, Second International Conference*, SEFM '04, pages 406–415, Washington, DC, USA, 2004. IEEE Computer Society.

[6] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19:371–384, July 1976.

[7] A. V. Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, volume 0, page 0249, Los Alamitos, CA, USA, 2001. IEEE Computer Society Washington, DC, USA, IEEE Computer Society.

[8] 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.

[9] J. Magee and J. Kramer. *Concurrency: state models & Java programs*. Wiley New York, 2006.

[10] J. Magee, J. Kramer, and D. Giannakopoulou. Analysing the behaviour of distributed software architectures: a case study. In *FTDCS*, pages 240–247. IEEE Computer Society, 1997.

[11] D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41 – 61, 1995.

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

[13] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, 26:978–1005, October 2000.

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