xx(11の552 1)

CINVESTAV

Centro de Investigación y de Estudios Avanzados del I.P.N.
Unidad Guadalajara

Modelado y Verificación por Comprobación de Modelos de una Clase de Sistemas de Procesamiento Industrial

Tesis que presenta:

Cesar Alberto Hernandez Arana

para obtener el grado de:
Maestro en Ciencias

en la especialidad de:
Ingeniería Eléctrica

Directores de Tesis

Dr. Raúl Ernesto González Torres
Dr. Arturo Sánchez Carmona

Guadalajara, Jal., Mayo de 2004.

Modelado y Verificación por Comprobación de Modelos de una Clase de Sistemas de Procesamiento Industrial

Tesis de Maestría en Ciencias
Ingeniería Eléctrica

Por:
Cesar Alberto Hernandez Arana
Ingeniero en Computación
Universidad de Guadalajara 1997-2001

Becario del CONACyT, expediente no. **169900**

Directores de Tesis
Dr. Raúl Ernesto González Torres
Dr. Arturo Sánchez Carmona

CINVESTAV del IPN Unidad Guadalajara, Mayo de 2004.

Agradecimientos:

A Dios, a mi Familia, a mi Novia, a mis Amigos, a mis Asesores, a mis Profesores y a CONACYT

IV

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Contenido del Capítulo

En este capítulo se justifica el hecho de utilizar la verificación formal como herramienta para el desarrollo de sistemas de control de procesos. En la sección 1.2 se esboza una descripción general del ciclo de desarrollo de un sistema de control. Es fundamental saber que el sistema trabaja adecuadamente. Tradicionalmente se ha utilizado la simulación para corroborar este hecho, pero la simulación no es suficiente para garantizar el correcto funcionamiento del sistema y se complementa con la incorporación de la verificación formal como una herramienta en el desarrollo del sistema de control. En la sección 1.3 se muestra a la verificación formal como una técnica que garantiza si la especificación se cumple o no en el modelo del sistema. Dos técnicas comunes para realizar la verificación formal son la prueba de teoremas y la comprobación de modelos. En la sección 1.4 se muestra el estado del arte de la verificación formal de los controladores lógicos para sistemas industriales utilizando la lógica de escalera como lenguaje de programación. En la sección 1.5 se plantean problemas vistos en el ámbito de la verificación formal de controladores lógicos y se expresa la motivación para realizar este trabajo. En la sección 1.6 se muestran los objetivos que pretendemos cubrir y las contribuciones que se hicieron en este trabajo. Por último en la sección 1.7 se describe el contenido de cada capítulo de la tesis.

1.2. Los Sistemas de Procesos

Hoy en día muchas de las empresas dedicadas a la industria de proceso tienen sus procesos de producción automatizados y requieren que dicha automatización sea por completo confiable, es decir que el sistema de control de procesos realice adecuadamente su función. En muchos sistemas de control de procesos una falla es inaceptable ya que representaría poner en riesgo el equipo o incluso vidas humanas. Por lo tanto es necesario asegurar que el sistema trabajará tal y como se requiere. Para lograr este objetivo, el personal encargado del desarrollo del sistema de control debe contar con las herramientas necesarias para corroborar que su sistema realiza las tareas para las cuales fue diseñado. En el ciclo de desarrollo de un sistema en general, se pueden encontrar varias etapas comunes: definición y captura de los requerimientos, análisis, diseño, implementación y pruebas; incrustadas en estas etapas siempre se realizan modificaciones y correcciones de errores encontrados a lo largo del desarrollo del sistema. Los errores y correcciones siempre están en relación directa con lo buena que haya sido la captura de requerimientos, el análisis y el diseño del sistema.

Los errores encontrados en las últimas etapas del ciclo de desarrollo del sistema son los más caros en reparar, por eso es deseable encontrar los errores lo más pronto posible para ahorrar tiempo y dinero [17]. Tradicionalmente este tipo de sistemas son corregidos por medio de métodos basados en simulación y pruebas. La simulación es una técnica que se realiza con un modelo del sistema obtenido de la etapa de diseño y que consiste en ejecutar las funciones del sistema en diversos escenarios para evaluar su comportamiento. Si en la simulación se detecta un error el diseñador lo analiza y hace las modificaciones necesarias al diseño del sistema de acuerdo a su experiencia. Una vez rectificado el error se vuelve a correr la simulación sobre el nuevo modelo del sistema. En ocasiones la corrección del error no es adecuada o hay más de una situación en la que ocurre el error y todo esto conlleva a la pérdida de tiempo. La deficiencia notable de la simulación es que aunque se hayan realizado muchas pruebas y correcciones no se tendrá la certeza de que el sistema no contiene errores, ya que por lo general los sistemas son muy grandes y no se pueden evaluar todas las posibles combinaciones de su comportamiento.

Por otra parte, la técnica de verificación formal es una herramienta relativamente nueva que ofrece la certeza matemática para saber si el sistema cumple realmente con los requerimientos descritos. Una de las ventajas de las verificación formal es que se puede utilizar en etapas tempranas de ciclo de desarrollo y evitar implementar un sistema con errores. Las técnicas de verificación formal basadas en máquinas de estados finitos (MEFs) han probado ser herramientas útiles en su aplicación a diseños industriales [52, 6]. Sin embargo como el diseño es descrito en términos de variables booleanas, se tiene que tratar con el problema de explosión de estados dado por la combinación de valores de las variables involucradas.

La verificación formal puede ser utilizada complementariamente con la simulación con el fin de garantizar que exista una similitud en los resultados obtenidos en ambas técnicas [31].

En este trabajo nos enfocamos en la verificación formal de la clase de sistemas de procesamiento industrial automatizados por medio de controladores lógicos programables (PLCs), en específico aquellos que están programados por medio del lenguaje de escalera (LE).

1.3. Verificación de Sistemas

Como hemos señalado, en los sistemas de control de procesos con frecuencia se utiliza la simulación como método sistemático para encontrar errores y más recientemente se han incorporado técnicas de verificación formal basadas en diversos enfoques. El incremento de tareas cada vez más especializadas hace a su vez que los sistemas de control sean cada vez más grandes y complejos, pero el requisito de que el sistema cumpla con su función es el mismo. Con la verificación formal podemos atacar errores que se producen en la etapa de diseño. Utilizamos un modelo del sistema de control obtenido en el diseño, al verificarlo formalmente podemos encontrar errores que incluso con la simulación no los habríamos encontrado nunca. Éste trabajo "extra" de modelado del sistema de control y su verificación es bien invertido ya que nos permite ahorrarnos la implementación de un sistema que pueda contener errores.

1.3.1. Verificación Formal

La técnica de verificación formal es una demostración matemática de la consistencia entre la especificación y la implementación de un diseño. La principal habilidad de la verificación formal es probar todos los posibles escenarios del sistema diseñado, muchos de los cuales son difíciles o incluso intratables mediante la simulación. Algunos de estos escenarios pueden ser muy complejos o por completo pasados por alto por el diseñador. Por lo general la verificación formal se hace con un modelo del sistema diseñado, el cual debe cumplir con ciertas propiedades que son imprescindibles en el funcionamiento del mismo. Las propiedades son descritas en un formalismo matemático, generalmente mediante lógica.

Las técnicas de verificación formal se agrupan en *prueba de teoremas*[1] y *técnicas basadas en MEFs.*

Prueba de teoremas

La verificación formal mediante prueba de teoremas es la técnica más general de verificación, en la cual una especificación y su implementación[2] son normalmente expresadas por medio de fórmulas de lógica de primer orden u orden superior [23]. Sus relaciones de equivalencia o implicación son tomadas como un teorema a ser probado dentro de un sistema de pruebas que utiliza un conjunto de axiomas y reglas de inferencia. La propiedad a verificar es una consecuencia de las proposiciones que describen el comportamiento del sistema dado tomadas como hipótesis. De esta manera, la prueba de teoremas es el proceso de buscar una demostración, mediante los axiomas y las reglas, que garantice que la propiedad es realmente una consecuencia lógica de las proposiciones. La ventaja principal de este método es que puede tratar directamente con sistemas que tienen un número infinito de estados. La principal desventaja de éste método es que los procedimientos no son totalmente algoritmizables y se requiere de mucha experiencia para verificar cualquier diseño utilizando prueba de teoremas. Algunos de los probadores de teoremas más ampliamente utilizados por la comunidad de verificación son HOL [24], PVS [41], Nqthm [7], ACL2 [12] y STeP [5].

[1] El término que se emplea en la literatura en inglés es *theorem proving*

[2] La implementación se refiere al diseño del sistema a ser verificado

Técnicas basadas en MEFs

Los métodos basados en MEFs ofrecen la ventaja de que el procedimiento de verificación puede ser automatizado y se clasifican en las siguiente categorías [25]:

- *Comprobación de Equivalencias*
  La equivalencia de una especificación y su implementación es corroborada, por ejemplo la equivalencia de funciones, equivalencia de autómatas, etc.

- *Comprobación de Modelos*[3]
  La especificación es en forma de una fórmula de lógica cuya validez es determinada con respecto a la semántica del modelo provista por la implementación.

Comprobación de Equivalencias

Es utilizada para probar la equivalencia funcional de dos representaciones de diseño modeladas al mismo o diferente nivel de abstracción [6, 30]. Puede dividirse en dos categorías: una es la comprobación de equivalencias combinacional y la otra es la comprobación de equivalencias secuencial.

En comprobación de equivalencias combinacional la funcionalidad de las dos descripciones son convertidas a una forma canónica las cuales son entonces estructuralmente comparadas. Se pueden utilizar *diagramas de decisión binaria* (BDD)[3]. Los dos diseños combinacionales son modelados por funciones booleanas y son probados equivalentes mediante la comparación de sus formas normales. Si se utilizan diagramas de decisión binaria reducidos y ordenados (ROBDDs)[3], los cuales tienen la propiedad de la forma canónica, es suficiente probar que ambos grafos son isomorfos.

La comprobación de equivalencias secuenciales es utilizada para verificar la equivalencia entre dos diseños secuenciales en cada estado. La comprobación de equivalencias secuenciales considera el comportamiento solo de los dos diseños e ignora los detalles de implementación. Puede verificar la equivalencia entre un modelo definido al nivel de transferencia entre registros (RTL) y netlist[4] o en RTL y un modelo

[3]El término que se emplea en la literatura en inglés es model checking

[4]Consiste en dar una lista de componentes del sistema, sus interconexiones y las entradas y salidas.

comportamental [10]. Para verificar la equivalencia de estos modelos se necesita una manipulación eficiente de las funciones de salida, estado siguiente y de los conjuntos de estados. Dos modelos son considerados equivalentes si y solo sí para cada secuencia de entrada ambos generan la misma secuencia de salida. La desventaja de ésta técnica es que no puede manejar diseños muy grandes debido a que sucede el problema de explosión de estados muy rápido.

Comprobación de Modelos

La técnica de verificación por comprobación de modelos [16] es la más estudiada en la literatura de verificación formal y la que ha mostrado ser capaz de tratar con diseños industriales [52]. La verificación por comprobación de modelos es un método en el cual el sistema es modelado mediante alguna estructura de estados y transiciones, comúnmente redes de Petri, grafos de control de flujo (CFG)[18], estructuras de Kripke ó autómatas finitos, por mencionar algunos. Por otro lado, también se requiere un lenguaje para representar las propiedades que se quieren probar en el sistema, usualmente lógica temporal lineal proposicional (LTLP) [32, 35] ó alguna lógica computacional arbórea (CTL)[15][5] El procedimiento de comprobación de modelos implica una búsqueda exhaustiva en todo el espacio de estados alcanzable del sistema para determinar la validez o invalidez de la propiedad en el modelo. En caso de que la propiedad no sea válida, el procedimiento calcula un contraejemplo que muestra la secuencia de estados del sistema donde la propiedad no se cumple. Se garantiza que la búsqueda siempre termina porque se realiza sobre un espacio de estados finito. Su aplicación no requiere de una persona experimentada en verificación para que guíe la prueba. Además, el contraejemplo es una ventaja extra de la cual carecen otras técnicas.

Verificación formal mediante comprobación de modelos

El proceso de realizar la verificación formal de un sistema mediante comprobación de modelos se puede dividir en tres grandes actividades: modelado, especificación y

[5]En muchos casos se emplea el mismo lenguaje con el que se modela el sistema.

prueba.

- **Modelado**

  La primer actividad consiste en representar el diseño del sistema con un formalismo aceptado por el método de comprobación de modelos. En muchos casos esto es una traducción directa en la que sólo se escribe el diseño con la sintaxis que acepta el método. En otros casos, dependiendo de las limitaciones de memoria, el modelo se construye haciendo una abstracción del diseño para eliminar todos aquellos detalles irrelevantes a la prueba. Por ejemplo, cuando se modelan circuitos, es muy común razonar en términos de compuertas y valores booleanos, y no en niveles de voltaje. De manera parecida, cuando se razona acerca de protocolos de comunicación, el interés básico está en el intercambio de mensajes y no en el contenido de ellos. En este punto se enfoca parte de nuestro trabajo, modelando directamente la descripción del sistema mediante una traducción de una sintaxis restringida del lenguaje de escalera a una estructura de estados y transiciones.

- **Especificación**

  Antes de poder realizar la prueba, es necesario establecer todas las propiedades que el modelo debe cumplir. Se acostumbra describir las propiedades por medio de lenguaje natural, pero para poder hacer un procesamiento de éstas es necesario representarlas con algún formalismo [51]. Para los diseños de hardware y software es común utilizar la lógica temporal, la cual puede describir propiedades dinámicas del sistema.

- **Prueba**

  Por conveniencia siempre se han utilizado herramientas de software que realizan esta actividad. De esta manera, sólo basta ingresar el modelo y la propiedad en sus lenguajes aceptados correspondientes y la herramienta los procesa y arroja un resultado. La persona encargada de realizar la verificación formal normalmente interactúa con la herramienta para darse cuenta si la propiedad se cumple en el modelo o no.

Si la herramienta encargada de efectuar la prueba da como resultado un contra-ejemplo puede deberse a alguna de las siguientes razones:

- La propiedad realmente no se cumple en el modelo del sistema diseñado. El diseño se debe corregir y nuevamente realizar la prueba de las propiedades para saber si ahora sí se cumplen en el nuevo modelo.

- El modelo es correcto pero la propiedad que se especificó realmente no tiene por qué cumplirse en el modelo. Habrá que revisar si realmente esa propiedad la debe tener el sistema.

- La codificación de la propiedad que se quiere probar no se hizo adecuadamente, es decir, no expresamos lo que realmente deseamos. Este punto es importante ya que si bien no es necesario tener experiencia para realizar la prueba, si es necesario saber codificar una propiedad en el formalismo utilizado.

- El modelo del sistema no corresponde con los comportamientos del diseño del sistema. Esto es, se están quitando o agregando comportamientos que no tiene el diseño del sistema. En pocas palabras se ha modelado incorrectamente. Es importante que en el método de modelado se puedan representar fielmente los comportamientos del diseño del sistema.

Comprobación de modelos explícita y simbólica

La comprobación de modelos se puede realizar de dos maneras: con cálculos explícitos ó con simbólicos. Estos últimos se basan principalmente en los BDDs, donde el modelo del sistema y la propiedad son representados con BDDs. La ventaja principal de esta implementación es que, cuando se aplica correctamente, se reduce en gran medida la cantidad de memoria requerida para almacenar todo el espacio de estados que se genera al momento de realizar la prueba. La desventaja es que se necesita mucho cálculo para obtener un estado en particular. La verificación por comprobación de modelos utilizando BDDs está orientada, en general, a sistemas de circuitos digitales debido a la lógica binaria que emplean. Algunas herramientas que realizan comprobación de modelos con cálculos simbólicos son: SMV (Symbolic Model Checking),

desarrollada por K. McMillan [37], una extensión de SMV. llamada NuSMV [14], MOCHA es una herramienta para especificación de sistemas y verificación modular de sistemas heterogéneos [2], VIS es una herramienta que realiza verificación, síntesis y simulación de sistemas de estados finitos [8].

La comprobación de modelos mediante cálculo explícito trabaja directamente con las estructuras de estados y transiciones que representan el modelo del sistema y la propiedad a ser verificada. La ventaja principal de utilizar un procedimiento explícito es que el tipo de sistemas que pueden ser verificados sólo se ve limitado por el poder de expresividad de la estructura de estados y transiciones con la que se modela el sistema y siempre se tiene la capacidad de avanzar gradualmente en el procedimiento de prueba. Siempre podemos obtener un estado en una operación sencilla de búsqueda. Algunas herramientas que han utilizado comprobación de modelos con cálculos explícitos son: SPIN, desarrollada por G. Holzmann en los laboratorios Bell [26], Murphi [19] y la herramienta VR desarrollada por el grupo de verificación formal del Cinvestav [43], la cual utilizamos como comprobador de modelos en este trabajo.

El problema de explosión de estados

Una de las principales desventajas del método de comprobación de modelos es conocida como el *problema de explosión de estados.* El problema sucede cuando en el sistemas existen muchos componentes que interactúan entre sí ó si en el sistema intervienen variables que pueden asumir muchos valores diferentes. Existe una gran variedad de técnicas para reducir el espacio de estados, entre las más comunes están: reducción de orden parcial [42], simulación [9, 34], bisimulación [22, 20] técnicas de abstracción [16] y simetrías [29, 21, 16].

1.4. Trabajo Relacionado

La programación de PLCs por medio del lenguaje de escalera es una de las técnicas más ampliamente utilizadas en los sistemas de control industrial. A pesar del surgimiento de otras técnicas en los últimos veinte años, aproximadamente el 50 % de los sistemas de control operando en la industria de manufactura en los Estados

Unidos es aún programado por medio de lenguaje de escalera [1]. Escenarios similares ocurren en otros sectores industriales. El alto costo de corregir errores generados en las etapas de diseño y que no se detectan sino hasta la implementación hace deseable poder asegurar que el controlador cumple con las propiedades requeridas desde las etapas tempranas de diseño. En esta sección se esbozan algunos trabajos sobre la verificación formal de PLCs. Todos ellos se enfocan en el modelado del sistema de control por medio de lenguaje de escalera y en la verificación formal por comprobación de modelos utilizando lógica temporal. En [4] se ofrece una descripción adicional sobre verificación formal de PLCs.

En [40] el método de verificación recibe 2 entradas: un modelo de estados y transiciones para el sistema a ser verificado y una lista de preguntas sobre el funcionamiento del sistema. El modelado del sistema se hace a través de la conversión del equipo de procesos, los procedimientos de operación y el software de control dados en lenguaje de escalera a un grafo de estados y transiciones etiquetado. Se utilizaron dos casos de estudio en este trabajo. El primero es un sistema de combustión, el cual se modeló a partir de la secuencia de operación deseada. El segundo ejemplo es un sistema de alarma, el cual se modeló a partir del diagrama de escalera. Se mencionan dos reglas para construir el grafo a partir del diagrama de escalera: 1) cada transición en el grafo se determina por las variables de entrada (variables independientes). 2)Cada variable independiente es usada para actualizar las variables dependientes (que son las que se asocian a bobinas) utilizando el diagrama de escalera. El resultado determina los estados generados. El modelo que se muestra en el ejemplo dado no corresponde con las dos reglas mencionadas para realizar la conversión. Más bien parece que el modo de construcción del modelo fue guiado por la secuencia de operación deseada. Además de que no se permite que dos variables de entrada cambien de un estado a otro, lo cual si es posible en un sistema real y no está restringido por las dos simples reglas. No se especifica el nombre del verificador de modelos pero se intuye que es la versión primitiva de SMV.

En [39] se modela mediante una traducción que se hace para una sintaxis restringida, únicamente se traducen contactos normalmente abiertos y cerrados en conjunción

o disyunción, así como la asignación de esos arreglos a las respectivas bobinas de cada escalón. La traducción se efectúa de manera casi directa del diagrama de escalera. Cada escalón es definido por las operaciones de negación, conjunción y disyunción, dependiendo de la forma en que los elementos se encuentren enlazados. Para cada bobina se crea un bloque de tipo *next* en SMV, y el cuerpo ese bloque lo constituye la lógica del escalón que activa la bobina correspondiente. El método es bastante intuitivo y no se necesita ser un experto en SMV para haber inferido la forma de modelar aún sin haber leído el artículo. En lo correspondiente al modelado del sistema es todo lo que se dice, después se expone un caso de estudio de una alarma que es multiplicada para probar la capacidad de SMV.

En [44] el enfoque de modelado de este trabajo fue unir todos los componentes y subcomponentes involucrados en el sistema de proceso en un modelo el cual captura las interacciones de los elementos con el sistema. Para modelar la lógica de escalera se definen las entradas, las salidas y las variables internas. Se consideran los temporizadores como módulos separados. Se define cada escalón como una función de las entradas, variables internas y salidas. Se agrupan los escalones del diagrama de escalera en aquellos que ya han sido evaluados y actualizados y aquellos que no. Se define un estado estable como aquél en donde ningún relevador del escalón cambiará su valor a menos que haya un cambio en las variables de entrada, éste estado se alcanza después de varios ciclos del PLC. El sistema se modeló como un circuito síncrono, en donde cada ciclo corresponde a un paso en la relación de transición. Se asumió que el entorno externo (variables de entrada) no puede cambiar hasta que el PLC ha alcanzado un estado estable. Una de las metas de verificación fue buscar los casos en donde no se llega a un estado estable. Los cambios en el sistema pueden ocurrir cada cierto intervalo de tiempo. Para modelar el hardware del proceso se hicieron módulos y se unieron. Para modelar el comportamiento humano se incorporaron entradas no determinísticas al sistema. Las propiedades referentes al funcionamiento del sistema fueron verificadas en SMV y fueron dadas en CTL. Para el modelado del sistema mencionan que se redujo el número de variables de estado introducíendo abstracciones y cabe señalar que el proceso de construcción del modelo duró 2 meses. En ninguna parte se establece como se construyó el modelo con todos los elementos mencionados

ni como es que se unieron los diversos módulos mencionados y en las especificaciones no se sabe como es que se verificó haber alcanzado un estado estable.

En [47] se muestra otra manera de traducir los programas PLC descritos mediante lenguaje de escalera. El sistema a ser controlado es modelado por un diagrama de bloques de sistema de condiciones y eventos (C/E [46]) y es transformado a código SMV. En este trabajo se introducen modelos de la dinámica de la planta mediante los diagramas de bloques de sistema C/E y se representan las relaciones temporales entre el programa PLC y la planta para detectar posibles problemas debido a la duración finita del tiempo de ciclo del PLC. La traducción se efectúa solo con variables booleanas, asignación estática (las variables son asignadas solo una vez en el programa), no hay funciones especiales ni bloques de función y no hay saltos a subrutinas. Se define un modulo SMV para el programa PLC que define el controlador, siguiendo las siguientes reglas: 1)definir un módulo de variables de entrada para cada sensor o comando de entrada, 2)definir una variable interna para cada salida del controlador, 3)introducir una función *next* para cada escalón en el programa PLC y 4) uso de transiciones no determinísticas para entradas. Para cada red de sistema C/E se hace un módulo SMV, de acuerdo a 7 reglas definidas. La verificación se efectúa definiendo un modulo *main* el cual combina el controlador, los módulos de la planta y la especificación. Cada transición entre estados del modelo SMV corresponde a un ciclo del PLC. En este trabajo si se detalla la manera de modelar el programa en lenguaje de escalera a código SMV Y se especifica como modelar los elementos de la planta descrito mediante el sistema C/E. Se necesita experiencia en SMV para modelar correctamente los diferentes módulos y enlazarlos en el módulo main. En el ejemplo se muestra un estado prohibido del sistema el cual es detectado el la etapa de verificación, dicho estado prohibido no se modela con el método propuesto, lo que se hace es simplemente evitar que suceda modificando el controlador. En este trabajo no se muestran resultados de eficiencia.

En [50] el método de modelado consta de dos fases principales: 1)el comportamiento de los programas PLC es modelado como un sistema de transiciones sincronizadas por intercambio de mensajes, el cual consiste en la definición de la semántica operacional y 2)la implementación en una herramienta para verificar las propiedades de

seguridad y vivacidad. El modelado de la semántica operacional comprende dos fases: 1)definición de la semántica basada en el IEC 61131-3 y 2)definición del sistema de transiciones que representa los elementos del lenguaje y la definición del grupo de reglas para formar elementos como un programa. La semántica operacional consiste de un sistema de transiciones que modela el ciclo de ejecución del PLC junto con un sistema de transiciones que representa las reglas comportamentales del lenguaje. Para el lenguaje de escalera el enfoque es parecido a una compilación del programa en un sistema de transiciones. El ciclo del PLC comprende 3 fases: lectura de entradas, computo y escritura de salidas. La fase de computo es usada para comenzar los diferentes subprogramas y al final de esta fase los programas son procesados. El sistema de transiciones es definido por las reglas de ejecución, la interpretación de las primitivas y el patrón de ejecución cíclico del PLC. Para este sistema de transiciones cada estado se etiqueta con el valor de las variables del programa más una variable auxiliar que lleva la cuenta de los ciclos de ejecución. Cada transición modela ya sea un evaluación de un escalón o el inicio del programa. Los escalones son evaluados secuencialmente. El estado fin de ciclo representa el fin del programa. Cada sistema de transiciones se debe traducir a código SMV para su posterior verificación, la codificación a SMV puede ser explícita (se construye todo el sistema de transiciones) o implícita (la representación del sistema de transiciones es simbólica). La verificación de las propiedades se hace codificándolas también en SMV. Este método es laborioso y requiere la traducción a un modelo de estructuras y transiciones y luego a SMV, en donde no se especifica la manera en que se efectúa la codificación. En ningún lugar se dice como se maneja el intercambio de mensajes.

En [48] se modela un amplio subconjunto del lenguaje de escalera incluyendo saltos y bloques del tipo Time On Delay (TON). El modelo es un autómata de estados. Las restricciones que se toman en cuenta son: 1)todas las variables son booleanas, 2)cada escalón se compone por la parte de prueba seguida de la parte de asignación, 3)Solo hay un programa en el PLC, 4)la evaluación es secuencial, 5)la evaluación significa computar el resultado de la parte de prueba del escalón y actualizar las variables de la parte de asignación, 6)puede interrumpirse la secuencia por una instrucción de salto y 7)se asume la ciclicidad de PLC. A un programa en lenguaje de escalera

se le asocia un conjunto de configuraciones. Una configuración es una descripción instantánea del estado del programa y es formalmente definida como una evaluación de todas las variables del programa. Estas variables incluyen las variables definidas y una variable especial llamada localización del control, lc. La semántica operacional se describe formalmente mediante la relación Φ entre configuraciones, es decir como una configuración c es seguida de una configuración c'. La definición de Φ tiene dos casos principales: 1)dentro de la ejecución del ciclo, el sistema se mueve de una configuración c a la siguiente c' mediante la evaluación del escalón actual en c. En la configuración c, la parte de prueba del escalón produce el valor v. Entonces c' es obtenida de c, actualizando con v las bobinas en la parte de asignación del escalón. 2)Al final de la ejecución del ciclo, el cual corresponde al máximo valor de lc. El sistema lee del entorno los nuevos valores de las variables de entrada y pone lc en 0 para un nuevo ciclo. Φ es descrita por una formula matemática como en [38]. Se desarrolló un traductor que acepta programas en LE y produce una definición del automata asociado en SMV. Para la declaración de los bloques TON existe un módulo SMV que define su operación lógica. Se utiliza lógica temporal lineal y computacional arbórea para la construcción de las propiedades a ser verificadas. No se analiza el comportamiento temporal del bloque TON, el cual es modelado solo lógicamente. En este trabajo no se muestran resultados.

En [52], se utiliza la verificación formal para investigar si una planta está apropiadamente equipada con instrumentos de medición, control y acción y también para confirmar la operación segura de la misma. Para esto el controlador lógico y los componentes de la planta son modelados a partir del diagrama de tuberías e instrumentos en una metodología propia para formar el llamado diagrama de eventos de control de procesos (PCED). El PCED esta dividido en 7 capas que cumplen una función específica, en cada capa intervienen tres tipos de nodos los cuales son llamados nodos objeto, nodos de entrada-salida y nodos de computación, todos estos nodos nos sirven como enlace entre las capas. Se efectúa una conversión a SMV mediante la modularización de los componentes que intervienen en los procesos (e.g. controlador, válvulas) y que fueron modelados en alguna de las capas del PCED. Una vez que se han modularizado los componentes se crea el módulo principal en SMV que deberá

instanciar los módulos correspondientes a cada componente. Los requerimientos de seguridad que se quiere investigar son formulados en CTL. Éste método de modelado contempla el comportamiento completo de la planta y no solo del controlador asociado. Para modelar el controlador se utiliza únicamente un módulo de SMV el cual se modela de la misma manera que en [44]

En [53] se muestra la verificación automática de sistemas de control mediante la traducción de los diagramas de escalera que implementan el sistema control a un modelo de autómatas temporizados [45]. El modelo puede ser verificado utilizando la herramienta de comprobación de modelos Uppal2k. La idea es incorporar el método de traducción a dicha herramienta. Los escalones del diagrama de escalera son traducidos instrucción por instrucción. Cada instrucción es modelada por un conjunto de transiciones entre 2 o más estados. Se han derivado algoritmos de traducción para un subconjunto de instrucciones del LE (el cual no se especifica). El modelo es generado con respecto a la propiedad que está siendo verificada y solo las partes del programa que afectan las variables en la propiedad son incluidas. No se especifica como se hace el modelado con la incorporación de la propiedad a verificar, ni como funciona el verificador.

En [54] se realiza una traducción de lenguaje de escalera a autómatas temporizados, con éste tipo de modelos es posible verificar programas que incluyan temporizadores. El verificador de modelos utilizados es Uppal2k [53]. La traducción se efectúa haciendo multiples transformaciones hasta llegar al automata temporizado del modelo: primero se efectúa un algoritmo de abstracción que lo que hace es únicamente construir una parte del programa que involucre a la propiedad a ser verificada únicamente. El algoritmo básico traduce la parte del programa a un autómata temporizado de la siguiente forma: 1)la parte del programa es traducida escalón por escalón a un autómata temporizado, en donde existe un estado inicial del escalón y dependiendo del cableado se habilitan las transiciones hacia otros estados hasta llegar al estado final del escalón. 2)Las entradas son modeladas como variables que cambian de valor no determinísticamente. Cada entrada es modelada por un pequeño autómata. Las salidas son modeladas como variables utilizadas en el modelo del programa. 3)La ejecución del programa se modela como un automata temporizado para cumplir con

la ejecución cíclica del PCL. Los diversos autómatas son sincronizados vía canales de comunicación. Al termino del algoritmo básico se obtiene un modelo para cada instrucción del programa y se conectan vía transiciones de acuerdo a la estructura del escalón. El modelo resultante puede ser simplificado. En algunos casos el modelo intermedio tiene que ser mejorado, por ejemplo añadiendo temporizadores. Se detalla un caso de estudio y las propiedades verificadas, así como los resultados obtenidos y los tamaños de los modelos generados. El método es un enfoque diferente a los estudiados. No se detalla la forma en que los diversos modelos se comunican ni la forma en que trabaja la herramienta de verificación.

En resumen, en la mayoría de los trabajos se utiliza SMV como comprobador de modelos y la diferencia principal entre esos trabajos es la manera de traducir los controladores dados en lenguaje de escalera a código SMV. Algunos incorporan elementos adicionales como temporizadores o saltos a subrutinas. En la tabla 1.1 se muestra información referente a los trabajos revisados. Las propiedades son especificadas en CTL y en [52] también en LTL, en [53] no se especifica el formalismo en que son expresadas las propiedades y en [54] las propiedades son especificadas en términos del lenguaje Uppaal2k.

#	Modelado del controlador	Sintaxis ó mejora	Estructura modelo
[40]	2 reglas de modelado	No especificado	Grafo edos. trans. etiq.
[39]	Bloques *next*	Restringida de LE	Manejada por SMV
[44]	Módulos SMV	Integra temporizadores	Manejada por SMV
[47]	Módulo controlador y módulo diag. C/E en SMV	Restringida LE	Diagrama C/E
[50]	En SMV explícito o implícito	Restringida de LE	Grafo edos. trans.
[48]	Módulo SMV por escalón	Incorpora TON	Manejada por SMV
[52]	PCED para planta controlador módulo SMV	Restringida de LE	Manejada por SMV
[53]	Sistema de transiciones	No especificado	Grafo edos. trans.
[54]	Autómata por escalón	Restringida de LE	Grafo edos. trans.

Tabla 1.1: Comparación entre los trabajos revisados.

1.5. Motivación

En la industria de proceso se debe tener la plena seguridad de que los procedimientos de operación y sus controladores asociados corresponden a modelos que cumplen con todas las restricciones impuestas. La descripción del controlador asociado por medio de LE es aún una de las formas preferidas por los diseñadores. En algunos de los trabajos revisados el paso de modelado es laborioso o requiere conocimiento adicional de un lenguaje determinado; lo ideal es que la traducción a un modelo formal sea a partir de una descripción general y de una forma sencilla. El trabajo revisado de verificación de controladores descritos por medio del LE es mediante cálculo simbólico utilizando SMV (o alguna de sus extensiones), únicamente en dos trabajos se utiliza un verificador diferente.

Los sistemas generalmente contienen muchos componentes que interactúan entre sí, generar un modelo de un sistema grande por lo general provoca un gasto de memoria considerable o de tiempo de verificación (si es que puede generarse el modelo o verificarse). Por lo general las especificaciones sólo hacen referencia a comportamientos que no involucran a todos los componentes del sistema, una opción es generar modelos que únicamente contengan el comportamiento de los elementos que aparecen en la especificación.

La verificación formal es una herramienta viable para saber si nuestro sistema hace realmente su función. La verificación formal garantiza que una propiedad es válida o no en el modelo del sistema. Si utilizamos comprobación de modelos garantizamos una exploración total del espacio de estados y por lo mismo nos enfrentamos a la explosión de estados cuando tratamos con sistemas con muchos componentes.

La motivación principal para el desarrollo de este trabajo fue el explorar y aplicar técnicas de reducción del espacio de estados en el modelado y verificación de una clase de sistemas de procesamiento industrial.

La incorporación de técnicas de reducción del espacio de estados puede hacer tratable un sistema que sea demasiado grande.

1.6. Objetivo

Establecer métodos y herramientas para verificar formalmente el funcionamiento de los controladores asociados a sistemas de procesos descritos por medio de LE, que incluya:

- Modelado de sistemas utilizando Autómatas de Büchi Generalizados Etiquetados (ABGEs). El modelado del sistema debe capturar el funcionamiento del proceso y ser capaz de elaborar modelos con la información descrita en la especificación. Los ABGEs han mostrado ser estructuras convenientes para la verificación formal y se cuenta con algoritmos eficientes para su manipulación.

- Construir herramienta que incorpore el método de modelado.

- Construir plataforma de software que habilite las herramientas de generación de fórmulas de LTL y comprobación de modelos para ser utilizadas eficientemente en el proceso de verificación.
  Se deberá diseñar una interfaz gráfica que englobe el proceso de verificación:
  - Modelado del Sistema.
  - Captura de propiedades del sistema.
  - Prueba por comprobación de modelos.

- Uso de herramientas de [43] y [11] para realizar la comprobación de modelos.

- Introducir técnicas de reducción del espacio de estados en la herramienta a desarrollar.

- Investigar el uso de técnicas de cálculo simbólico junto con cálculo explícito.

Las principales contribuciones de este trabajo son:

- Formalización e implementación de un método de modelado a partir de la traducción de una sintaxis restringida del lenguaje de escalera a un ABGE.

- Formalización e implementación de una técnica de reducción del espacio de estados basada en la simulación por la proyección del conjunto de variables que aparecen en la propiedad a ser verificada.

- Implementación de la identificación de contraejemplos espurios producidos en la verificación de modelos abstractos.

- Formalización e implementación de una técnica de refinamiento para el tipo de modelos abstractos generados.

1.7. Contenido de la Tesis

En el capitulo 2 se muestra la teoría relativa a los controladores lógicos programables y se detalla la parte del estándar correspondiente, llamado IEC 1131-3. En ese mismo capítulo se muestran algunas definiciones referentes a los autómatas de Büchi generalizados etiquetados y de la lógica temporal lineal proposicional. En el capítulo 3 se detalla un método de construcción de modelos utilizando una sintaxis restringida del LE. Se plantea la conveniencia de utilizar conocimiento adicional del sistema para elaborar un modelo fiel al funcionamiento físico del mismo. En el capítulo 4 se esbozan algunas técnicas de reducción del espacio de estados y se muestra la técnica elegida, la cual se basa en una técnica de simulación por la proyección del conjunto de variables implicadas en la propiedad a verificar. En el capítulo 5 se detalla la manera de corroborar si un contraejemplo obtenido en la verificación de un modelo abstracto es en realidad un contraejemplo en el modelo concreto o si sólo se trata de un contraejemplo espurio. Así mismo, se muestra la manera de realizar un refinamiento sobre un modelo abstracto. En el apéndice A mostramos detalles de implementación y funcionamiento de la herramienta desarrollada para la construcción de modelos. En el apéndice B mostramos la interfaz diseñada, la cual reúne el proceso de verificación: modelado del sistema, descripción de propiedades y prueba. Así como también la forma en que podemos realizar el análisis de un contraejemplo y refinamientos en modelos abstractos.

Capítulo 2

Preliminares

2.1. Contenido del Capítulo

En este capítulo se introduce la teoría referente a los controladores lógicos programables y algunas definiciones sobre autómatas de Büchi generalizados etiquetados y lógica temporal lineal proposicional, las cuales se utilizarán constantemente en el resto de este trabajo. El capítulo comienza con una introducción a los controladores lógicos programables (PLCs) en la sección 2.2. Posteriormente en la sección 2.3 se bosqueja el estándar IEC 1131 el cual se refiere al manejo de los PLCs. En la sección 2.4 se establecen los fundamentos de la parte 3 del estandar IEC 1131 la cual se refiere a los lenguajes de programación de los PLCs. Se menciona más a detalle en la sección 2.5 el lenguaje de programación de escalera. En la sección 2.6 se muestra la definición de los autómatas de Büchi generalizados etiquetados (ABGEs), los cuales utilizaremos en el modelado de sistemas. Por último en la sección 2.7 se esbozan los fundamentos de la lógica temporal líneal proposicional utilizada para la descripción de propiedades del sistema.

2.2. Controladores Lógicos Programables

En la industria de proceso la automatización es fundamental para lograr los objetivos de producción. Para realizar de manera correcta y eficiente el proceso de automatización es necesario contar con un dispositivo o conjunto de dispositivos que se encarguen de regular las acciones de los elementos involucrados. Éstos dispositivos se denominan sistemas de control. Los PLCs fueron originalmente diseñados para reemplazar los sistemas de control basados en relevadores y paneles de control cableados. En figura 2.1 se muestra un sistema PLC clásico.

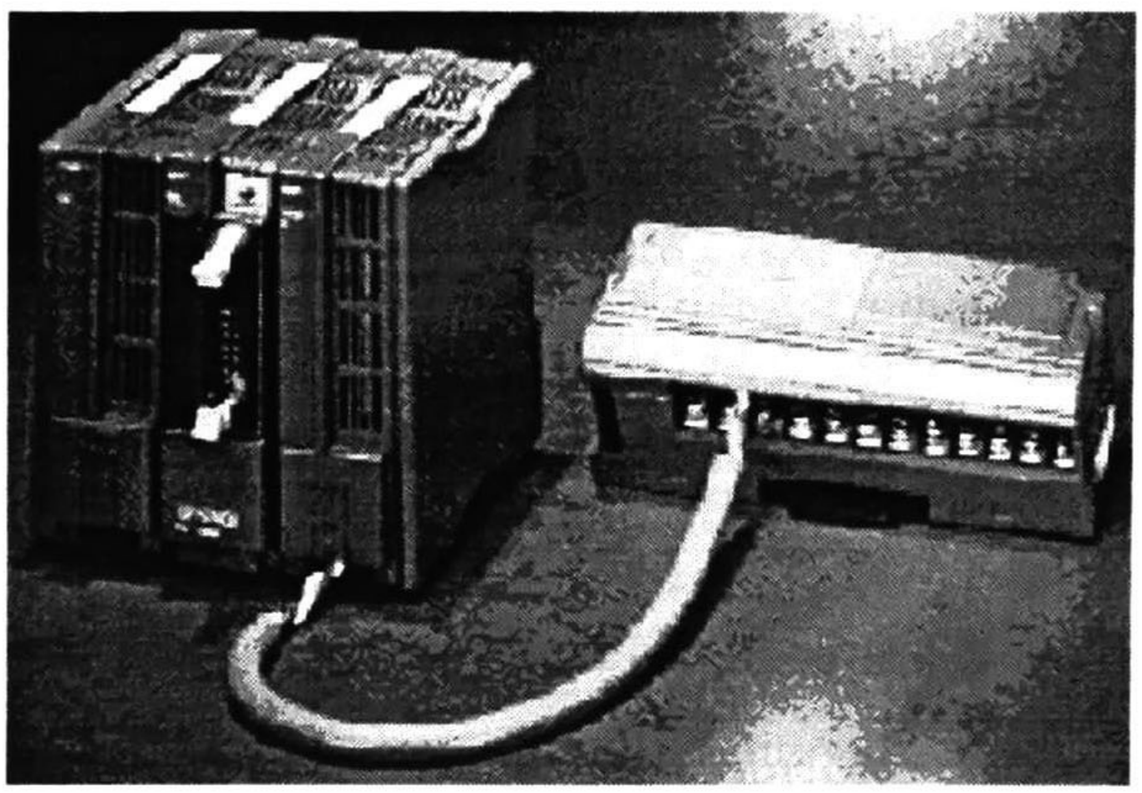


Figura 2.1: Sistema PLC.

La función más básica ejecutada por los PLCs es examinar el estado de las entradas, ejecutar un programa almacenado en él y emitir señales a través de sus salidas para controlar dispositivos externos. La combinación de entradas para producir una salida es llamada lógica de control. Se requieren combinaciones lógicas para llevar a cabo un plan de control o programa. Este plan de control es almacenado en memoria utilizando un dispositivo de programación que introduce el programa al sistema. El procesador periódicamente examina[1] el plan de control en memoria en un orden secuencial. La cantidad de tiempo requerido para examinar las entradas, salidas, ejecutar la lógica de control y activar las salidas es llamado tiempo de ciclo de operación.

[1] scan es el término que se emplea en inglés

En la figura 2.2 se esquematiza el ciclo de operación de un sistema PLC. Al comienzo del ciclo se hace una lectura de las entradas y se asignan a las variables correspondientes, una vez que se han leído las entradas se comienza la lectura del programa de control y se ejecuta dicho programa. Terminada la ejecución del programa de control se comienza el ciclo de actualización de las salidas, activando y desactivando las salidas correspondientes. Terminado ese proceso se vuelve a comenzar el ciclo de

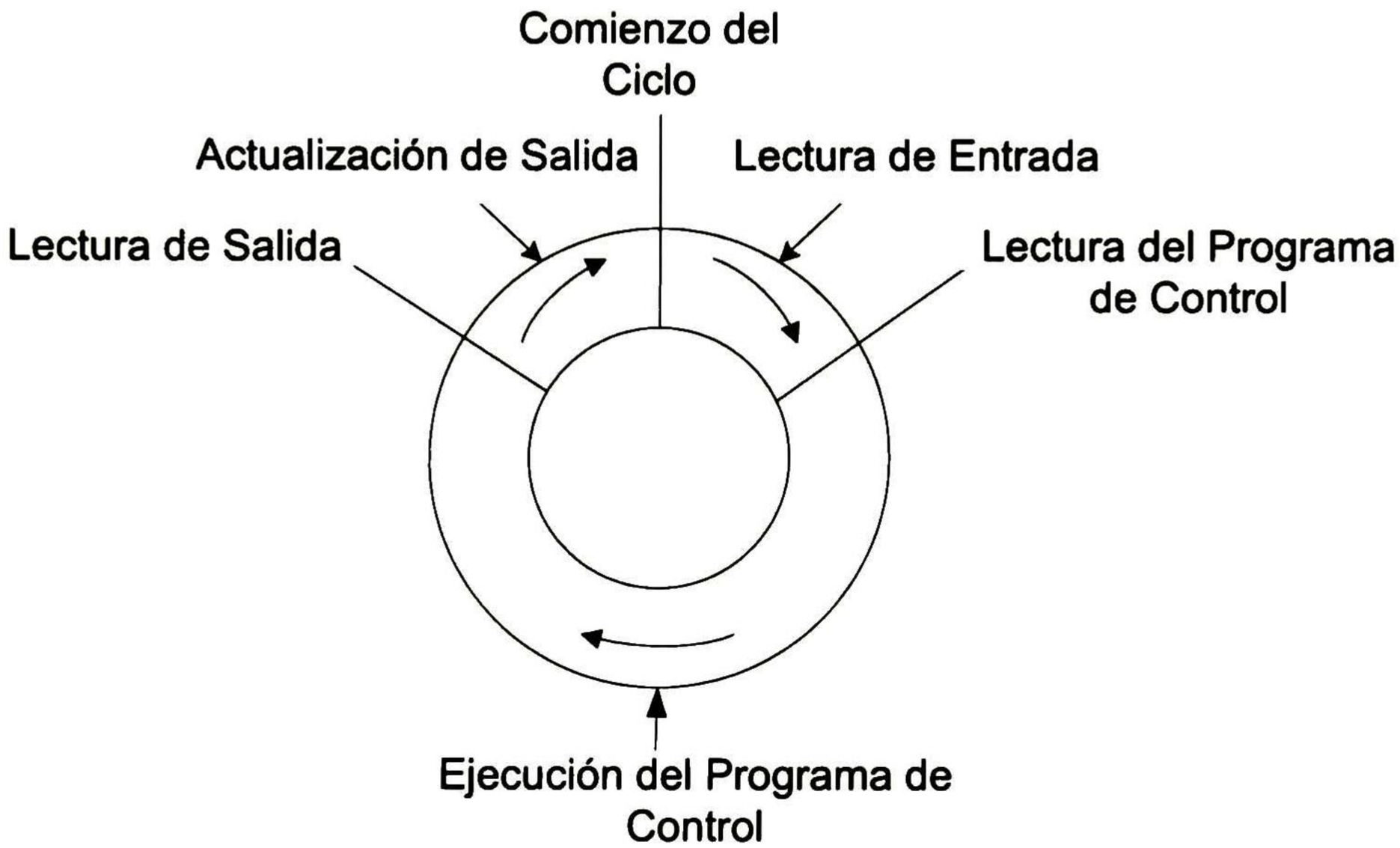


Figura 2.2: Ciclo de operación.

operación.

2.2.1. Componentes Básicos de los Sistemas PLC

Un PLC siempre consiste de un procesador, una unidad de memoria, un sistema de entrada/salida, un lenguaje de programación, un dispositivo de programación y una fuente de poder. El diagrama de componentes del sistema PLC se muestra en la figura 2.3. En la figura 2.4 se muestra la conexión física de un sistema PCL y se relaciona con la figura 2.3 de la siguiente manera: el cuadro *circuitos de entrada* toma

las señales de dispositivos como botones, interruptores, sensores, etc. y representa el sistema de entrada del PLC. En el cuadro *unidad lógica* se encuentran las partes correspondientes a la memoria, el procesador y el programa que contiene la lógica de control del sistema. La unidad lógica se encarga de la ejecución del programa de control así como de la interacción con los circuitos de entrada y salida. La fuente de poder es representada por la línea de corriente *120V ac.* Por último el cuadro *circuitos de salida* emite señales a dispositivos como válvulas, luces, bombas, etcétera, y representa el sistema de salida del PLC.

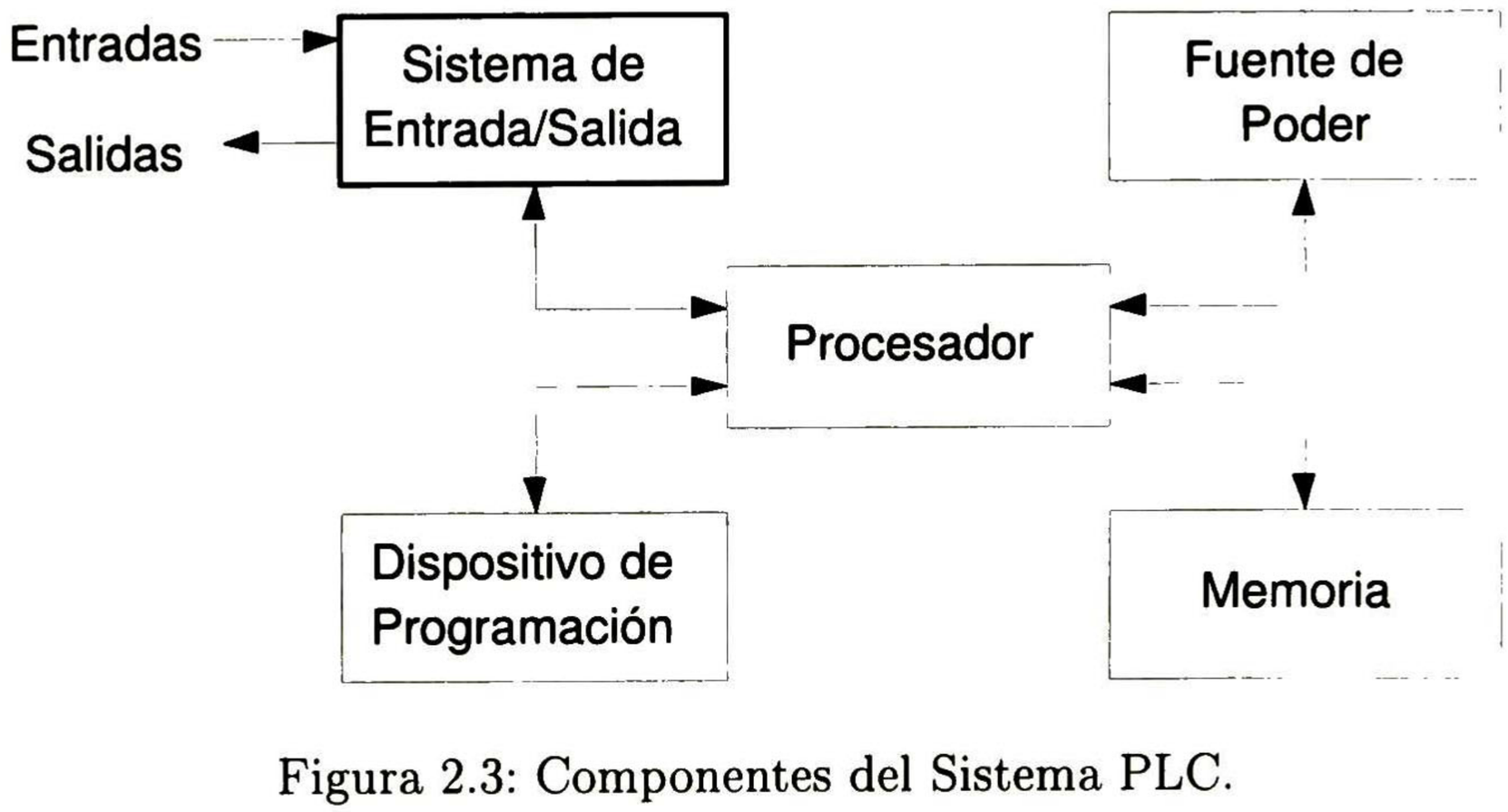


Figura 2.3: Componentes del Sistema PLC.

- El procesador.

  El procesador consiste de uno o más microprocesadores y otros circuitos integrados que ejecutan la lógica, control y funciones de memoria del sistema PLC. El procesador lee las entradas, ejecuta la lógica determinada por el programa, ejecuta cálculos y activa las señales de salida. El procesador controla el ciclo de operación del PLC. Este ciclo de operación consiste de una serie de operaciones ejecutadas secuencial y repetidamente. Durante la lectura de la entrada (input scan), el PLC examina las entradas de los dispositivos externos para ver si alguna señal está presente o ausente, esto es, si los dispositivos de entrada están en el estado de encendido (On) o apagado (Off). El estado de éstas entradas

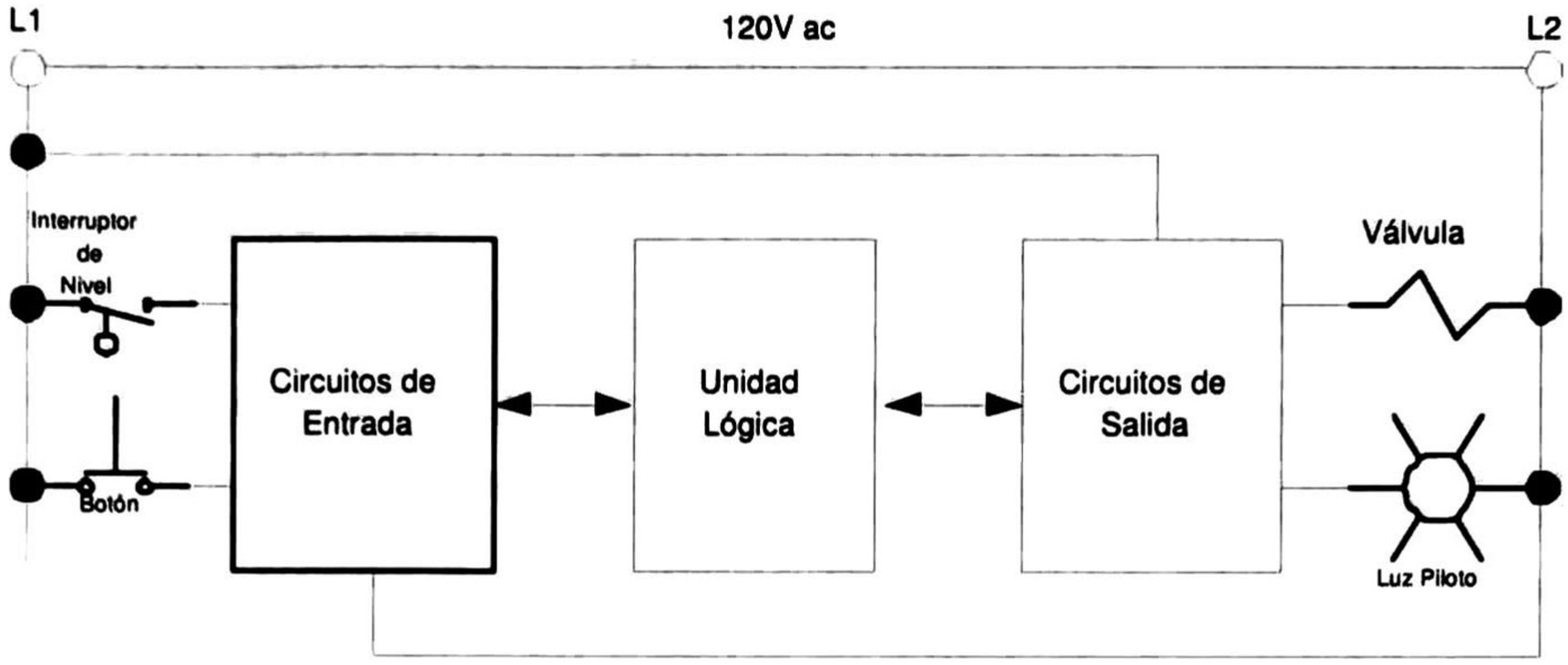


Figura 2.4: Interconexión de componentes externos con el sistema de entradas y salidas.

es temporalmente almacenado en una tabla de entradas o archivo de entrada. Durante la ejecución de programa (program scan), el procesador lee las instrucciones del programa de control, utiliza el estado de las entradas de la tabla de entradas y determina si una salida será o no energizada. El estado resultante de las salidas es escrito en la tabla de salidas o archivo de salida. Basado en los datos de la tabla de salidas el PLC energiza o desenergiza su circuito de salida asociado el cual controla los dispositivos externos.

- Memoria.

  La memoria es utilizada para almacenar el programa de control para el PLC. Se localiza usualmente en la misma zona que la unidad central de procesamiento (CPU). La información almacenada en memoria determina de que manera los datos de entrada y salida serán procesados. La memoria almacena piezas individuales de datos llamadas bits. Un bit tiene 2 estados: 1 ó 0. Las unidades de memoria son montadas en tarjetas de circuitos y son usualmente especificadas en incrementos de miles o "K", donde 1K son 1024 palabras (e.g. $2^{10} = 1024$ de espacio de almacenamiento). La complejidad del plan de control determina la cantidad de memoria requerida. El usuario puede accesar a dos áreas de memoria en el sistema PLC: archivos de programa y archivos de datos. Los

archivos de programa almacenan el programa de control, archivos a subrutinas y archivos de error. Los archivos de datos almacenan los datos asociados con el programa de control, tales como estado de los bits de entrada/salida, valores de contadores y temporizadores y valores acumulados y otras constantes o variables almacenadas. Estas dos áreas de memoria son llamadas de usuario o memoria de aplicación. El procesador tiene también un área exclusiva o memoria de sistema que dirige y ejecuta actividades operacionales tales como la ejecución del programa de control y la coordinación de la lectura de las entradas y la actualización de las salidas.

- Sistema de Entrada/Salida. El sistema de entrada salida provee la conexión física entre el equipo de procesos y el microprocesador. Este sistema utiliza varios circuitos de entrada o módulos para tomar las señales de los dispositivos externos utilizados para sensar y medir las cantidades físicas de los procesos tales como movimiento, nivel, temperatura, presión, flujo o posición. En respuesta al estado sensado o valores medidos, el procesador activa los módulos de salida. Estos módulos dirigen dispositivos como válvulas, motores, bombas y alarmas para ejercer control sobre una máquina o un proceso.

- Fuente de Poder. La fuente de poder convierte la linea de voltaje de ac a voltaje dc para suministrar energía a los circuitos electrónicos en el sistema PLC. Esta fuente de poder rectifica, filtra y regula el voltaje y corriente para suministrar la cantidad correcta de voltaje y corriente al sistema.

- Dispositivos de Programación. Los dispositivos de programación son utilizados para ingresar, almacenar y monitorear el software del PLC. Pueden ser sistemas dedicados o sistemas basados en computadoras personales. Normalmente se conecta el dispositivo de programación al sistema PLC solo mientras se programa, se arranca o hay problemas en el sistema de control. En otro caso el dispositivo de programación se encuentra desconectado del sistema.

- Lenguajes de Programación. Los lenguajes de programación permiten al usuario comunicarse con el PLC vía el dispositivo de programación. Los fabricantes de

PLCs utilizan algunos lenguajes de programación diferentes, pero todos ellos utilizan instrucciones para comunicar un plan de control básico al sistema. Un plan de control es definido como un conjunto de instrucciones ordenadas en una secuencia lógica para controlar las acciones de un proceso o máquina. Ciertas reglas gobiernan la manera en que las instrucciones son combinadas así como la forma de las instrucciones. Estas reglas e instrucciones se combinan para formar un lenguaje. Los cuatro tipos más comunes de lenguajes encontrados en los sistemas PLCs son:

- Lenguaje de escalera(LE).
- Diagrama de bloques funcionales (FBD).
- Lista de instrucciones (IL).
- Texto estructurado (ST).

2.3. Estándar IEC 1131

El estándar internacional 1131 emitido por Comisión Electrotécnica Internacional (IEC) es una colección de estándares sobre los PLCs y sus periféricos asociados, consiste de las siguientes partes:

- Parte 1. Información general. Establece las definiciones e identifica las principales características relevantes a la selección y aplicación de controladores programables y sus periféricos asociados.
- Parte 2. Requerimientos de equipo y pruebas. Especifica los requerimientos de equipo y las pruebas relacionadas para los controladores programables y sus periféricos asociados.
- Parte 3. Lenguajes de programación. Define como un conjunto mínimo los elementos básicos de programación, reglas sintácticas y semánticas para los lenguajes de programación más comúnmente utilizados, incluyendo los lenguajes gráficos de diagramas de escalera y diagramas de bloques funcionales y los lenguajes textuales de lista de instrucciones y texto estructurado;

- Parte 4. Reporte técnico. Provee un panorama general y las líneas de aplicación del estándar para el usuario final de los controladores programables.

- Parte 5. Especificación de servicios de mensajes. Define la comunicación de los datos entre los controladores programables y otros sistemas electrónicos utilizando la especificación de mensajes industrial (MMS[2]).

- La parte 6 está reservada para uso futuro.

- Parte 7. Control de lógica difusa. Define los elementos básicos de programación para la lógica de control difusa y como se utiliza en los controladores programables.

- Parte 8. Provee una guía de desarrolladores de software para los lenguajes de programación definidos en la parte 3.

2.4. Estándar IEC 1131 Parte 3

La parte del estándar IEC 1131 a la cual nos enfocamos en nuestro trabajo es la referente a la sección 3, en la cual se delinean los lenguajes de programación de los PLCs y sus elementos básicos tanto de sintaxis como de semántica.

El estándar IEC 1131-3 es el primer esfuerzo real para normalizar los lenguajes de programación usados en automatización industrial. Han participado 7 empresas internacionales añadiendo 10 años de experiencia en el área de automatización industrial. El resultado ha sido 200 páginas de texto, con 60 tablas incluyendo tablas de características, con la especificación de la sintaxis y semántica de una colección unificada de lenguajes de programación, incluyendo el modelo de software global.

Una manera de ver el estándar es dividiéndolo en dos partes

1. Elementos comunes

2. Lenguajes de programación

[2]MMS por sus siglas en inglés, de acuerdo al estándar internacional ISO/IEC 9506

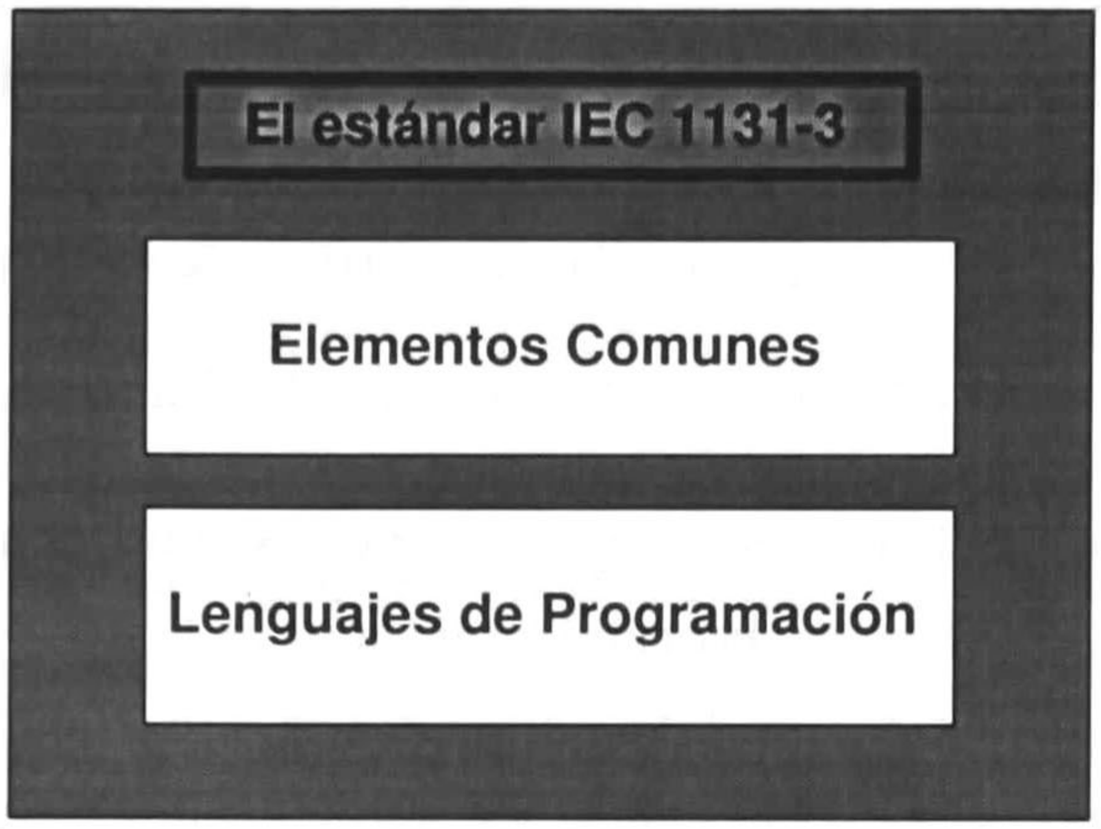


Figura 2.5: Partes del estándar IEC 1131-3.

2.4.1. Elementos Comunes

Ésta cláusula define los elementos gráficos y textuales los cuales son comunes a todos los lenguajes de programación de los PLCs.

- Caracteres impresos.

  En esta parte se detallan cual es el conjunto de caracteres utilizados para representar los elementos comunes, así como la definición de los identificadores, palabras clave, utilización de espacios y comentarios.

- Representación de los datos externos.

  Se define la forma en que se debe representar la información referente a: numeros enteros y reales, cadenas de caracteres y símbolos especiales y cantidades de tiempo.

- Tipificación de datos.

  Los tipos de datos son definidos dentro de los elementos comunes. La tipificación de datos previene errores en etapas tempranas. Es usada para definir los tipos de cualquier parámetro a ser utilizado. Esto evita, por ejemplo, dividir una fecha por un entero. Los tipos de datos comunes son: binarios (booleanos), enteros,

reales, octetos (byte), palabras (doble octeto), así como también fechas, por ejemplo cadenas tipo hora_del_día. Basados en estos tipos de datos se pueden construir y definir tipos de datos personalizados, conocidos como tipos de datos derivados.

- Variables.

  Las direcciones de hardware (e.g. entradas y salidas) son asignadas explícitamente a las variables en las configuraciones, recursos o programas. De esta manera es creado un alto nivel de independencia, soportando la reusabilidad del software. El alcance de las variables son normalmente limitadas a la unidad de organización del programa en la cual ellas son declaradas, e.g. locales o globales. Esto significa que sus nombres pueden ser reutilizados en otras partes sin ningún conflicto, eliminando así otra fuente de errores, e.g. variables temporales. Se les puede asignar un valor inicial al arranque o rearranque, con el fin de tener el valor correcto en estos puntos de operación.

- Unidades para la organización del programa (POU).

  Dentro de IEC 1131-3 las funciones, bloques funcionales y los programas son llamados Unidades Organizativas del Programa, o POUs.

  1. Funciones. IEC ha definido funciones normalizadas y funciones definidas por el usuario. Algunas funciones normalizadas son: ADD (suma), ABS (valor absoluto), SQRT (raíz cuadrada), SIN (seno) y COS (coseno). Las funciones definidas por el usuario, una vez escritas, pueden ser usadas repetidamente.

  2. Bloques Funcionales (FB). Estos son el equivalente a los circuitos integrados (IC) o a los módulos de control discreto-analógicos, representando funciones de control especializado. Ellos contienen tanto datos como algoritmos. Los FBs tienen una interfaz bien definida así como un IC o un módulo de control discreto tipo caja negra. De esta forma ellos dan un clara separación entre los diferentes niveles de programadores o personal de mantenimiento. Una vez definido puede ser usado una y otra vez en el

mismo programa, diferentes programas, o más aún en diferentes proyectos. Esto lo hace altamente reutilizable. Los bloques funcionales pueden ser escritos en cualquiera de los lenguajes IEC, y la mayoría de los casos hasta en lenguajes de alto nivel como el lenguaje C.

3. Programas. Con los bloques constructivos anteriormente mencionados, se puede decir que un programa es una red de funciones y bloques funcionales. Un programa puede ser escrito en cualquiera de los lenguajes de programación definidos.

- Gráfica de función secuencial(SFC).

  La SFC describe gráficamente el comportamiento secuencial de un programa de control. Es derivado de las Redes de Petri y la norma Grafcet IEC 848, con los cambios necesarios para convertir la representación de una norma para documentación a un conjunto de elementos de control ejecutables. En la figura 2.6 se muestra una grăfica de función secuencial que representa la operación para llenar y vaciar un tanque.

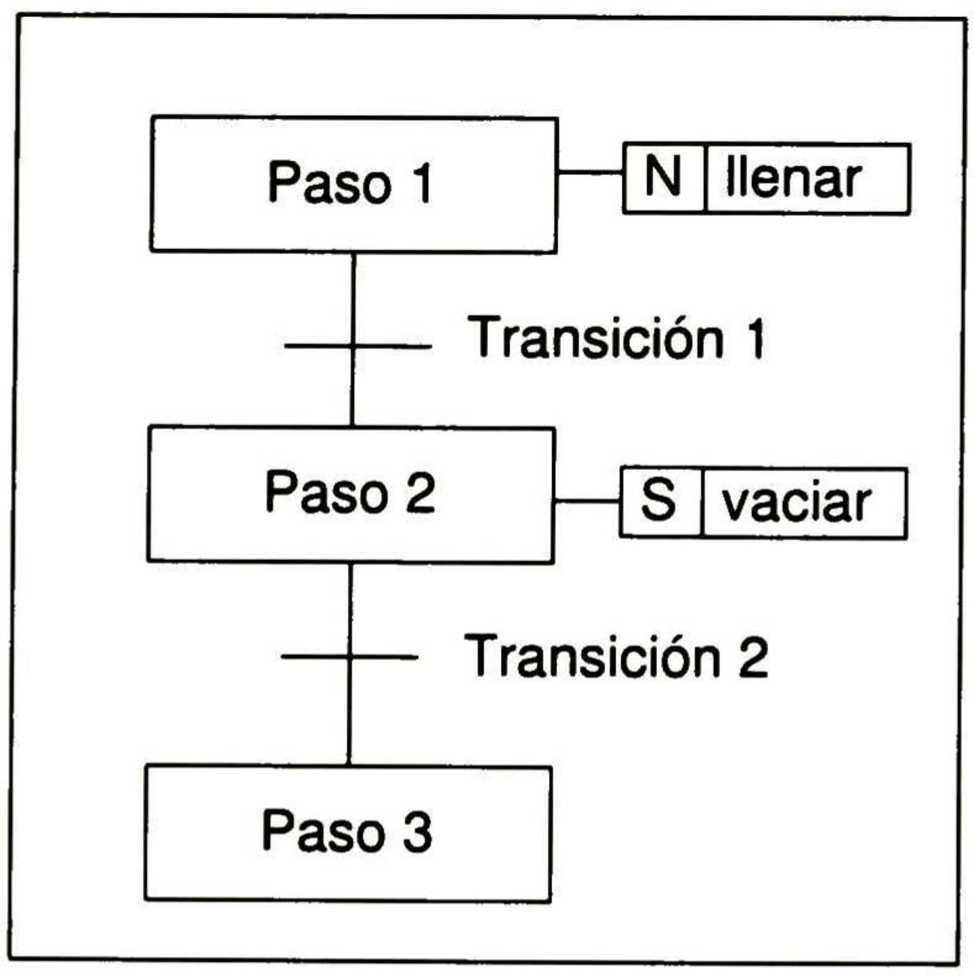


Figura 2.6: Gráfica de Función Secuencial. En el paso 1 se llena un tanque, se cumple la condición para activar la transición 1 y se procede a vaciar el tanque en el paso 2.

La SFC estructura la organización interna de un programa, y ayuda a descomponer un problema de control en partes más manejables, manteniendo una visión del todo. El SFC consiste de *pasos*, enlazados con *bloques de acción* y *transiciones*. Cada paso representa un estado particular del sistema bajo control. Una transición es asociada a una condición, la cual, cuando es cierta, causa que el paso anterior a la transición sea desactivado, y el siguiente paso sea activado. Los pasos son interconectados a bloques de acción, realizando estos ciertas acciones de control. Cada elemento puede ser programado en cualquiera de los lenguajes definidos por la IEC, incluyéndose a si misma la SFC. Se pueden programar secuencias alternativas y mas aun secuencias paralelas. Por ejemplo, una secuencia es usada para un proceso primario, y la segunda para monitorear globalmente condiciones operativas de límite.

- Configuración, recursos y tareas.

  En el nivel más alto, el software total requerido para solucionar un problema particular de control puede ser definido como una *configuración*. Una configuración es especifica a un tipo particular de sistema de control, incluyendo el arreglo de hardware, e.g. recursos de procesamiento, direcciones de memoria para los canales de I/O y demás capacidades del sistema. Dentro de la configuración se pueden definir uno o mas *recursos*, se puede ver un recurso como un elemento que es capaz de ejecutar programas escritos en el estándar de la IEC. Así mismo dentro de los recursos se pueden definir una o mas *tareas*. Las tareas controlan la ejecución de un conjunto de *programas* y/o *bloques funcionales*. Estos últimos pueden ser ejecutados periódicamente o en la ocurrencia de un evento disparador específico, tal como un cambio en una variable. La figura 2.7 muestra como están dispuestos los elementos de una configuración.

  Los programas son construidos mediante el uso de un número de diferentes elementos de software escrito en cualquiera de los lenguajes definidos por la IEC. Un programa consiste típicamente, de una red de funciones y bloques funcionales, los cuales son capaces de intercambiar datos. Las funciones y bloques funcionales son los bloques básicos de construcción, conteniendo estructura de

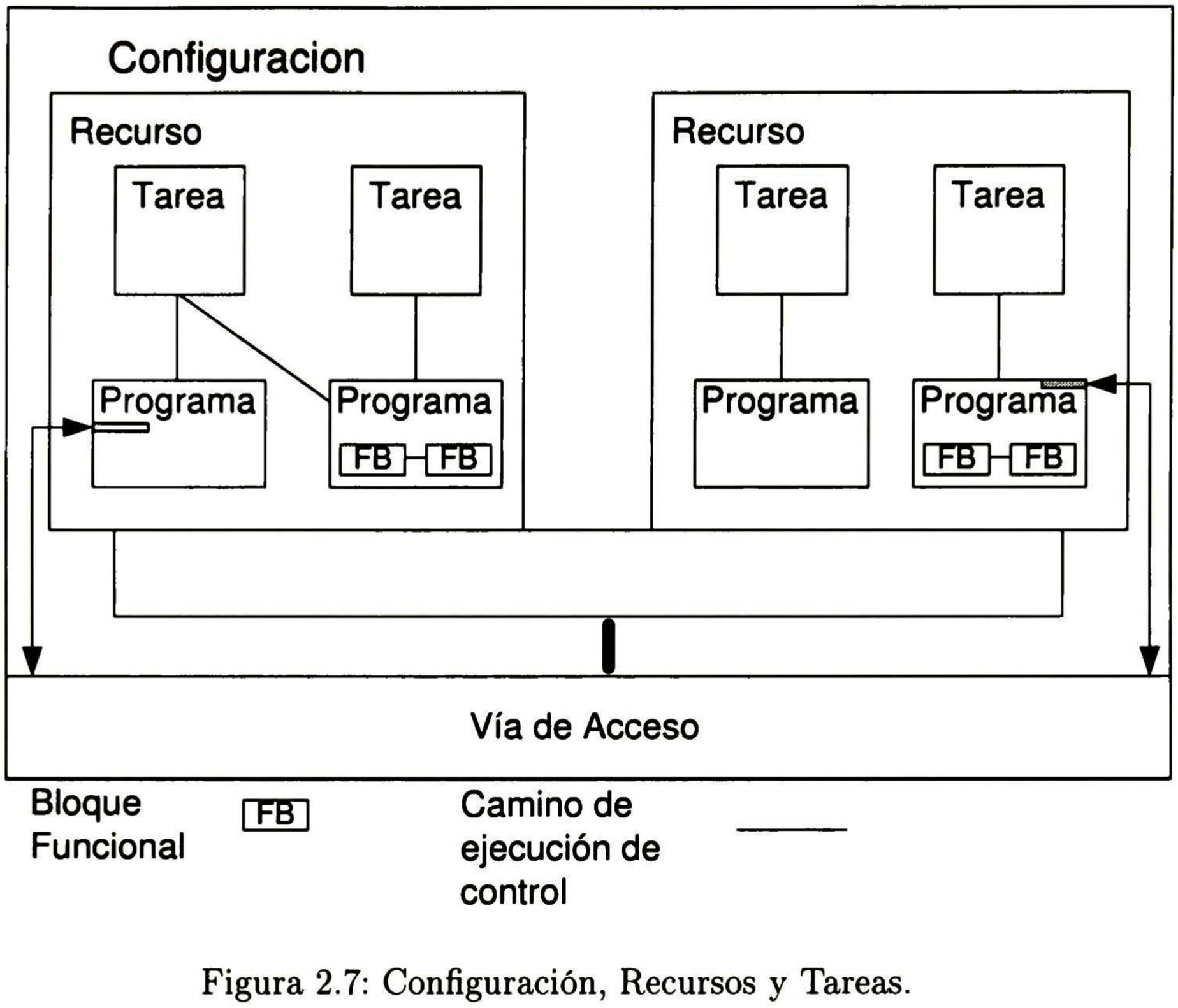


Figura 2.7: Configuración, Recursos y Tareas.

datos y un algoritmo. El estándar IEC 1131-3 es adecuado para un amplio rango de aplicaciones de control.

2.4.2. Lenguajes de Programación

Dentro de la norma son definidos cuatro lenguajes de programación. Esto significa que su sintaxis y semántica ha sido también definida, no dejando ningún espacio para los dialectos. Una vez que han sido aprendidos, se pueden usar en una gran variedad de sistemas basados en esta norma.

Los lenguajes consisten de dos versiones textuales y dos gráficas:

1. Gráficos:

   - Diagrama de escalera.

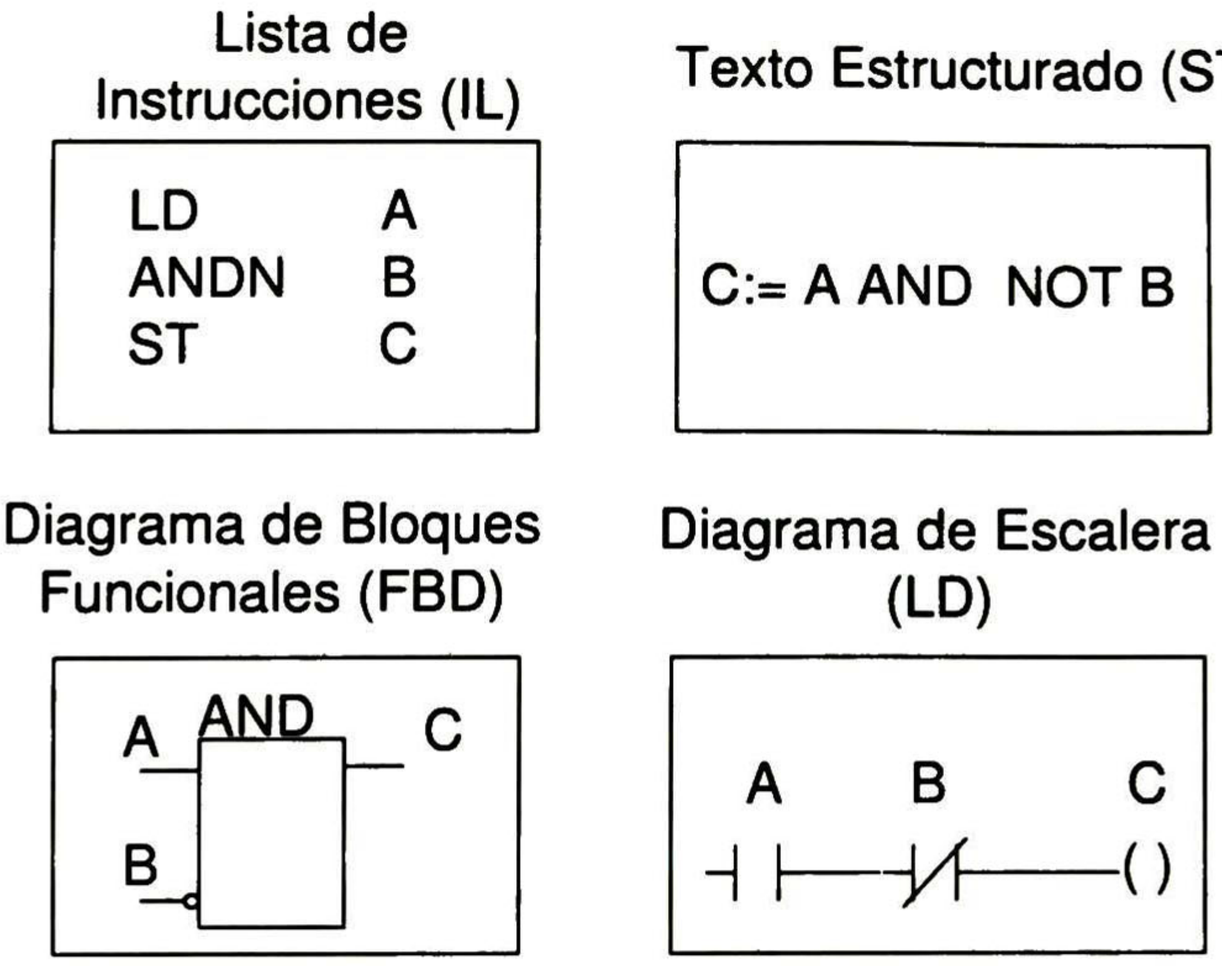


Figura 2.8: Lenguajes de Programación de PLCs

- Diagrama de bloques funcionales.

2. Textuales:
   - Lista de instrucciones.
   - Texto estructurado.

En la figura 2.8, los cuatro lenguajes describen la misma parte de un programa sencillo.

La selección del lenguaje a ser usado depende de:

- La preparación del programador.
- El problema a resolver.
- El nivel de descripción del problema.
- La estructura del sistema de control.
- La interfaz con diferente personal o departamentos.

Los cuatro lenguajes están interconectados: ellos proveen una colección de programación común.

Los Diagramas de Escalera tiene sus raíces en Estados Unidos. Están basados en un representación gráfica de la Lógica de Escalera por Relevadores (RLL).

La lista de Instrucciones es su contraparte Europea. Es un lenguaje textual, que se asemeja al lenguaje ensamblador.

Diagrama de Bloques Funcionales es muy común a la industria de procesos. Este expresa el comportamiento de funciones, bloques funcionales y programas como un conjunto bloques gráficos interconectados, parecido a diagramas de circuitos electrónicos. El sistema se observa en términos del flujo de señales entre elementos de procesamiento.

Texto Estructurado es un lenguaje muy poderoso con sus raíces en el ADA, Pascal y C. Puede ser usado para la definición de bloques funcionales muy complejos, los cuales pueden ser utilizados dentro de cualquiera de los otros lenguajes.

2.5. Lenguaje de Escalera (LE)

Un programa en LE habilita al PLC a probar y modificar datos por medio de símbolos gráficos estandarizados. Estos símbolos están configurados en redes de una manera similar a un "escalón" de un diagrama de lógica de escalera de relevadores. Las redes del diagrama de escalera están acotadas sobre rieles de energía a la izquierda y derecha.

2.5.1. Representación de Líneas y Bloques

Los elementos del lenguaje gráfico son dibujados utilizando caracteres del conjunto de caracteres del ISO/IEC 646 o utilizando elementos gráficos o semigráficos mostrados en la figura 2.9.

Las líneas pueden ser extendidas por el uso de conectores. Ningún almacen de datos o asociación con elementos de datos será asociado con el uso de conectores; así, para evitar ambigüedad, deberá haber un error si el identificador utilizado como

etiqueta del conector es el mismo nombre que el de otro elemento nombrado en la misma unidad de organización del programa.

2.5.2. Dirección del Flujo en las Redes

Una red es definida como el conjunto maximal de elementos gráficos interconectados, excluyendo los rieles derecho e izquierdo. Se deberá proveer para asociar con red o grupo de redes en un lenguaje gráfico una etiqueta de red delimitada a la derecha por dos puntos (:). Esta etiqueta deberá tener la forma de un identificador o un entero decimal sin signo. El ámbito de una red y su etiqueta deberá ser local a la unidad de organización del programa en la cual la red está localizada.

Los lenguajes gráficos son utilizados para representar el flujo de una cantidad conceptual a través de una o más redes que representan el plan de control, esto es:

- "Flujo de energía", análogo al flujo de energía eléctrica en un sistema de relevadores electromecánico, típicamente utilizado en diagramas de escalera de relevadores.
- "Flujo de señal", análogo al flujo de señales entre elementos de sistemas de procesamiento de señales, típicamente usados en los diagramas de bloques de funciones.
- "Flujo de actividad", análogo al flujo de control entre elementos de una organización o entre los pasos de un secuenciador electromecánico, típicamente utilizado en las gráficas de funciones secuenciales.

La correcta cantidad conceptual deberá fluir a lo largo de las líneas entre elementos de una red de diagrama de escalera de izquierda a derecha.

2.5.3. Evaluación de las Redes

El orden en el cual las redes y sus elementos son evaluados no es necesariamente el mismo orden en que están etiquetados o son mostrados. Similarmente, no es necesario que todas las redes sean evaluadas antes de la evaluación de una red dada que

N°	Característica	Ejemplo
1	Líneas horizontales: carácter menos ISO/IEC 646	- - - - -
2	Líneas verticales: carácter línea vertical ISO/IEC 646	|
3	Conexión horizontal/vertical: carácter más ISO/IEC 646	| - - + - - |
4	Cruce de líneas sin conexión: caracteres ISO/IEC 646	| | - - - - - - | - - - - | |
5	Esquinas conectadas y no conectadas: caracteres ISO/IEC	| | - - - - + + - - - - | - - - - + - + + - - - - | | |
6	Bloques con líneas conectadas: caracteres ISO/IEC 646	| + - - - - - - - + - - - | | - - - | | - - - | | + - - - - - - - + |
7	Conectores utilizando caracteres ISO/IEC 646: Conector Continuación de una línea conectada	- - - - - - >OTTO> >OTTO - - - - - ->

Figura 2.9: Representación de líneas y bloques

pueda ser repetida. Sin embargo, cuando el cuerpo de una unidad de organización de programa consiste de una o más redes, los resultados de la evaluación en la red en dicho cuerpo deberán ser funcionalmente equivalentes a la observacia de las siguientes reglas:

1. Ningún elemento de una red deberá ser evaluado hasta que los estados de sus entradas han sido evaluados.

2. La evaluación de un elemento de la red no estará completa hasta que los estados de todas sus salidas han sido evaluados.

3. La evaluación de la red no está completa hasta que las salidas de todos sus elementos han sido evaluados, incluso si la red contiene uno de los elementos de control de ejecución definidos en la siguiente sección.

4. Dentro de una unidad de organización del programa escrito en LE, el orden en el cual son evaluadas las redes deberá ser de arriba hacia abajo como aparecen en el diagrama, excepto si este orden es modificado por los elementos de control de ejecución definidos en 2.5.4.

Una trayectoria de retroalimentación se dice que existe en una red cuando la salida de una función o bloque de función es usada como la entrada a una función o bloque de función el cual la precede en la red; la variable asociada es llamada variable de retroalimentación. Por ejemplo, la variable booleana RUN es una variable de retroalimentación en el ejemplo mostrado en la figura 2.10. Las trayectorias de retroalimentación en los diagramas de escalera están sujetas a las siguientes reglas:

1. Las variables de retroalimentación deberán ser inicializadas por uno de los mecanismos definidos en la cláusula 2.4.2 del estándar IEC 1131-3. El valor inicial deberá ser usado durante la primer evaluación de la red.

2. Una vez que el elemento con una variable de retroalimentación como salida ha sido evaluado, el nuevo valor de la variable de retroalimentación deberá ser usado hasta la siguiente evaluación del elemento.

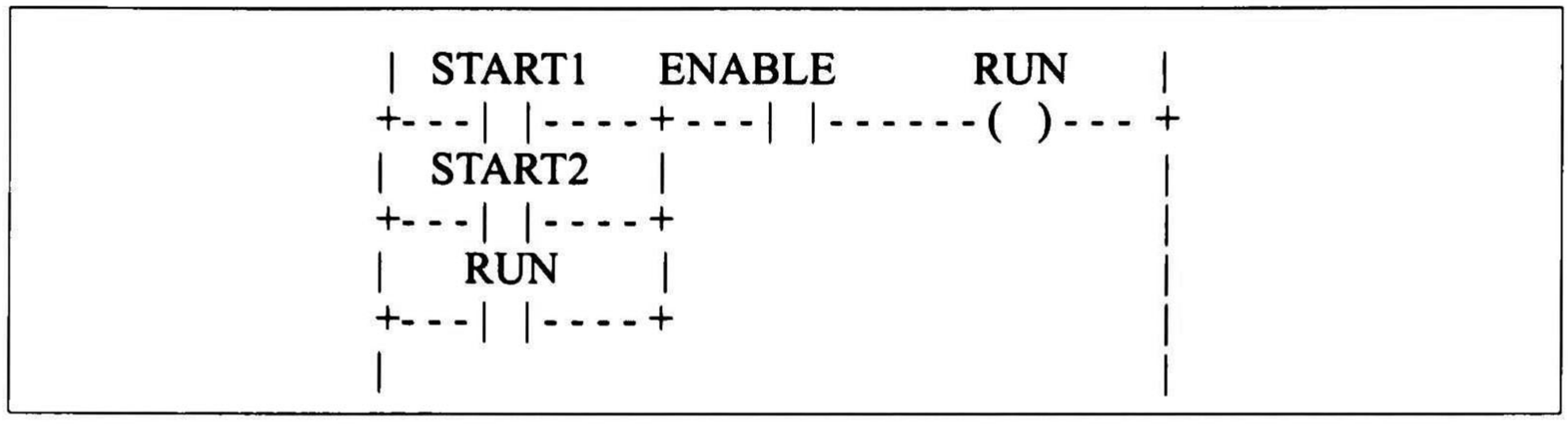


Figura 2.10: Retroalimentación de trayectoria.

2.5.4. Elementos de Control de Ejecución

La transferencia de control de un programa en el LE será representado por los elementos gráficos mostrados en la figura 2.11. Los saltos deberán ser mostrados por una línea de señal booleana terminada en una flecha de doble cabeza. La línea de señal para un salto condicional deberá originarse en una variable booleana, en la salida booleana de una función o un bloque de función o en la línea de flujo de energía del diagrama de escalera. Una transferencia del control del programa a la etiqueta de la red designada deberá ocurrir cuando el valor booleano de la señal es 1. Los saltos incondicionales son un caso especial del salto condicional.

El objetivo de un salto debe ser una etiqueta de red dentro de la unidad de organización del programa en la cual el salto ocurre, e.g si el salto ocurre dentro de un bloque ACTION...END_ACTION, el objetivo del salto deberá estar dentro del mismo bloque.

Los regresos condicionales de funciones y bloques de función deberán ser implementados utilizando una instrucción RETURN. La ejecución del programa deberá ser transferida de regreso a la entidad que la invocó cuando la entrada booleana es 1, y deberá continuar en la forma normal cuando la entrada booleana sea 0. Los regresos incondicionales deberán ser provistos por el fin físico de la función o bloque de función, o por un elemento RETURN conectado al riel izquierdo en el diagrama de escalera.

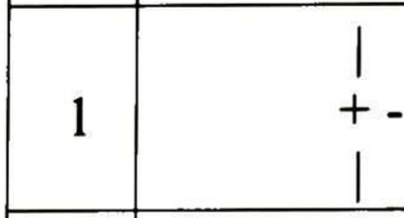
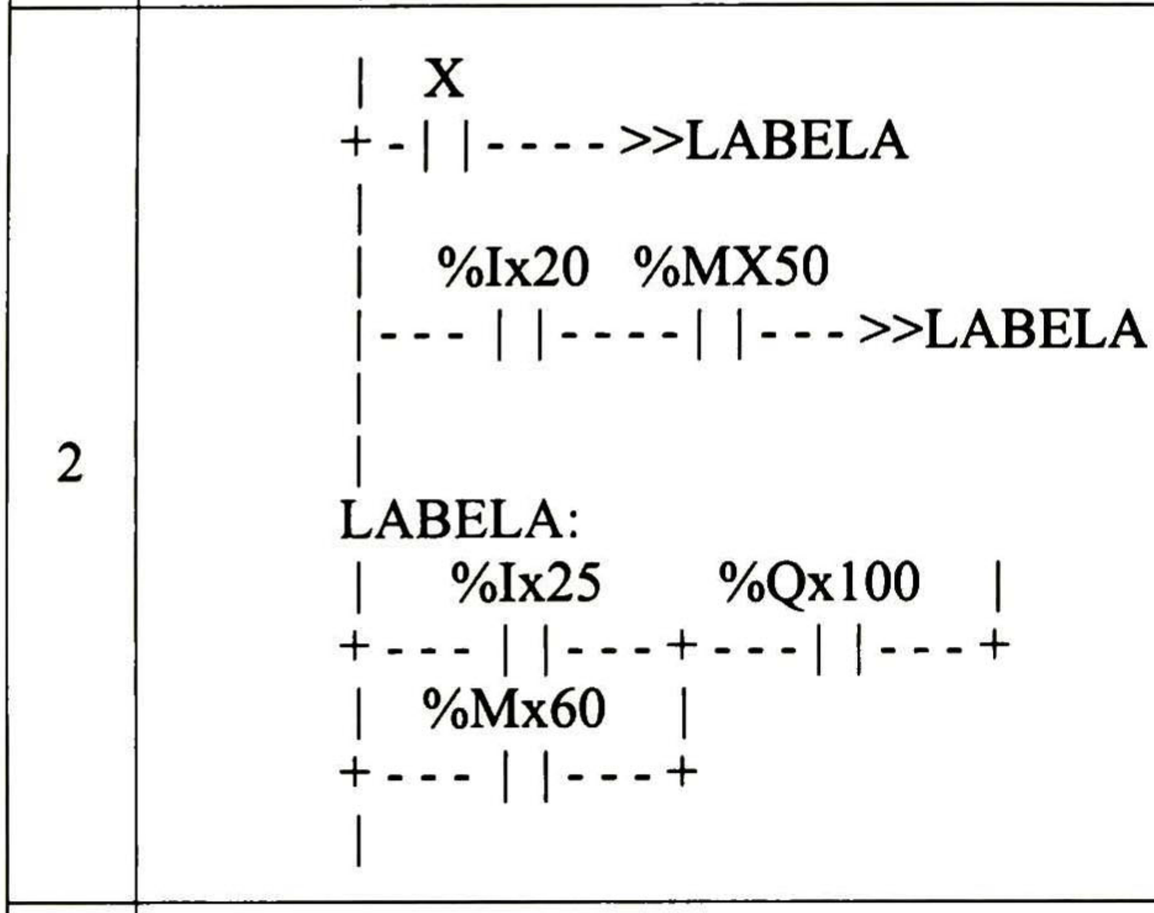
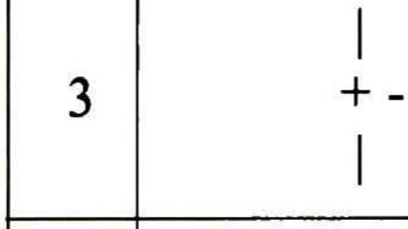
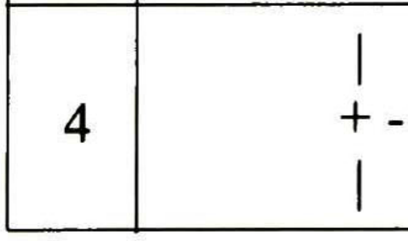
N°	Símbolo/Ejemplo	Explicación
1		Salto incondicional
2		Salto condicional Ejemplo: Condición de salto Objetivo del salto
3		Regreso Condicional
4		Regreso Incondicional

Figura 2.11: Elementos de Control de ejecución

2.5.5. Rieles de Energía

Como se muestra en la figura 2.12, la red del diagrama de escalera estará delimitada por la izquierda por una línea vertical conocida como riel de energía izquierdo y en el lado derecho por una línea vertical conocido como riel de energía derecho. El riel de energía derecho puede ser explícito o implícito.

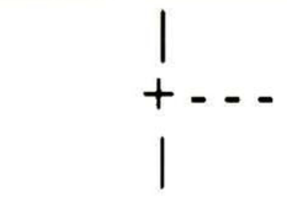
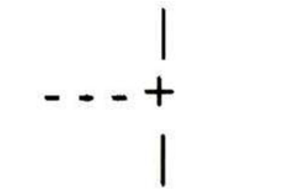
N°	Símbolo	Descripción
1		Riel de energía izquierdo (con enlace horizontal agregado)
2		Riel de energía derecho (con enlace horizontal agregado)

Figura 2.12: Rieles de energía.

2.5.6. Elementos Enlazados y Estados

Como se muestra en la figura 2.13 , los elementos de enlace pueden ser horizontales o verticales. El estado del elemento enlazado deberá ser denotado *activado*(On) o *desactivado* (Off), correspondiente a valor booleano 1 o 0 respectivamente.

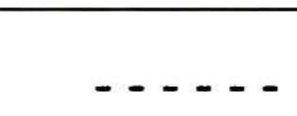
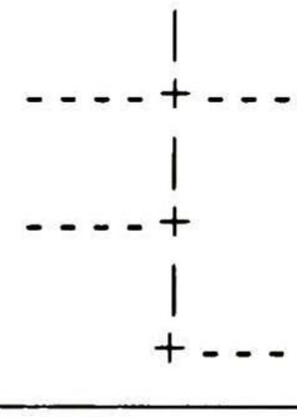
N°	Símbolo	Descripción
1		Enlace horizontal
2		Enlace vertical (con enlace horizontal agregado)

Figura 2.13: Elementos de Enlace

El estado del riel izquierdo deberá ser considerado energizado todas las veces. Ningún estado es definido para el riel derecho.

Un elemento enlazado horizontal deberá ser indicado por una línea horizontal. Un elemento enlazado horizontal transmite el estado del elemento a su izquierda inmediata al elemento sobre su inmediata derecha.

El elemento enlazado vertical deberá consistir de una línea vertical intersecada con uno o más elementos enlazados horizontales sobre cada lado. El estado del enlace vertical deberá representar la operación *o inclusiva* de los estados On de los enlaces horizontales sobre su lado izquierdo, esto es, el estado del enlace vertical deberá ser:

- Off si los estados de todos los enlaces horizontales unidos a su izquierda son Off.
- On si el estado de uno o más de los enlaces horizontales unidos a su izquierda es On.

El estado del enlace vertical deberá ser copiado a todos los enlaces horizontales unidos sobre su derecha. El estado del enlace vertical no deberá ser copiado a cualquiera de los elementos horizontales unidos sobre su izquierda.

2.5.7 Contactos

Un contacto es un elemento el cual imparte un estado al enlace horizontal sobre su lado derecho el cual es igual a la operación booleana y del estado del enlace horizontal a su lado izquierdo con una función apropiada de su entrada, salida o variable booleana asociada. Los símbolos estándar de contactos se muestran el la figura 2.14.

2.5.8. Bobinas

Una bobina copia el estado del enlace a su izquierda al enlace a su derecha sin modificaciones y almacena una función apropiada al estado de transición del enlace izquierdo a la variable booleana asociada. Los símbolos estándar de la bobinas son descritos en la figura 2.15.

2.6. Autómatas de Büchi Generalizados Etiquetados

Un autómata de Büchi generalizado etiquetado es una estructura de estados y transiciones semejante a los autómatas finitos etiquetados (AFE)[11], con la diferencia de que estos últimos tienen estados de aceptación (o finales) y los primeros tienen una *colección de conjuntos de estados de aceptación.* Además, la interpretación que actúa

N°	Símbolo	Descripción
1	**** --| |-- Ó **** --! !--	Contacto normalmente Abierto El estado del enlace izquierdo es copiado al enlace derecho si el estado de su variable booleana asociada (indicada por ****) esta activada (On) En otro caso, el estado del enlace derecho es desactivado (Off).
2	**** --|/|-- Ó **** --!/!--	Contacto normalmente cerrado El estado del enlace izquierdo es copiado al enlace derecho si el estado de su variable booleana asociada esta desactivada (Off) En otro caso, el estado del enlace derecho es activado (On).

Figura 2.14: Contactos

sobre los AFE está basada en palabras finitas, mientras que los ABGE se basan en palabras infinitas. A continuación definiremos con mayor precisión esos conceptos.

Sea AP un conjunto finito y no vacío de proposiciones atómicas. Denotemos por Σ a la potencia de AP, esto es $\Sigma := 2^{AP}$ Una *palabra infinita* (ω-*palabra*) v sobre Σ es una función del tipo $\sigma : \mathbb{N}_0 \to \Sigma$; esto es, σ es una sucesión infinita de elementos de Σ, que se denota por:

$$\sigma = a_0 a_1 a_2 \ldots a_n \ldots, \text{ donde } a_n = \sigma(n) \text{ para cada } n \in \mathbb{N}_0$$

El *conjunto de todas las secuencias infinitas* u ω-*palabras* sobre Σ se denota por Σ^ω A cualquier subconjunto de Σ^ω se le llama un *lenguaje* sobre Σ.

Definición 2.6.1. Autómata de Büchi Generalizado Etiquetado

Un Autómata de Büchi Generalizado Etiquetado sobre Σ es una quíntupla de la forma

$$\mathcal{A} := (Q,\ Q_0,\ \rho,\ \mathcal{F},\ l)$$

donde:

- Q es un conjunto finito y no vacío de *estados* de $\mathcal{A}$

N°	Símbolo	Descripción
1	**** --( )--	Bobina El estado del enlace izquierdo es copiado a la variable booleana asociada y al enlace derecho.
2	**** --(/)--	Bobina negada El estado del enlace izquierdo es copiado al enlace derecho. El inverso del estado del enlace es copiado a la variable booleana asociada, esto es, si el estado del enlace izquierdo es desactivado (Off), entonces el estado de la variable asociada es activado (On) y vice versa

Figura 2.15: Bobinas

- $Q_0 \subseteq Q$ es un conjunto de *estados iniciales* de $\mathcal{A}$
- $\rho : Q \to 2^Q$ es una función de transición de $\mathcal{A}$, de modo que si $q \in Q$, entonces $\rho(q)$ es llamado el conjunto de *estados sucesores* de q en $\mathcal{A}$
- $\mathcal{F} := \{F_1, F_2, \ldots, F_k\} \subseteq 2^Q$ es una *colección* (finita) *de conjuntos de estados de aceptación* (o *conjuntos legales*) de $\mathcal{A}$
- $l : Q \to 2^\Sigma$ es una *función de etiquetado* de $\mathcal{A}$ que asigna a cada estado $q \in Q$ una colección de subconjuntos de AP, denominada la etiqueta de q

Definición 2.6.2. Lenguaje aceptado por un ABGE

Sea $\mathcal{A} = (Q,\ Q_0,\ \rho,\ \mathcal{F},\ l)$, con $\mathcal{F} = \{F_1, F_2, \ldots, F_k\}$, un ABGE sobre Σ.

i) Una *corrida* (o *cómputo*) de $\mathcal{A}$ es una secuencia infinita de estados de $\mathcal{A}$, $\pi = q_0, q_1, \ldots, q_n, \ldots$, tal que $q_0 \in Q_0$ y para todo $n \geq 0$, $q_{n+1} \in \rho(q_n)$.

ii) Sea $inf(\pi)$ el conjunto de todos los estados que ocurren un número infinito de veces en la corrida π; esto es $inf(\pi) := \{q \in Q \mid q = q_i$ para un número infinito de $i\}$.

Decimos que la corrida π es *legal* si para cada $F_j \in \mathcal{F}$ la intersección de F_j e $inf(\pi)$ no es vacía, $inf(\pi) \cap F_i \neq \emptyset$. Toda corrida de $\mathcal{A}$ cumple esta condición

por definición. Toda corrida legal es una *corrida de aceptación.* Es decir, que al menos un estado de cada F_i aparezca un número infinito de veces sobre π.

Observe que si $\mathcal{F} = \emptyset$, entonces toda corrida de $\mathcal{A}$ es de aceptación, puesto que se satisface vacuamente la condición de aceptación. Por otro lado, si algún conjunto $F_i = \emptyset$, entonces ninguna corrida de $\mathcal{A}$ es de aceptación puesto que la condición de aceptación no se satisface.

iii) Una *ω-palabra* $\sigma = a_0 a_1 a_2 \ldots a_n \ldots$ de Σ^ω es *aceptada* por $\mathcal{A}$ si y sólo si existe una corrida de aceptación $\pi = q_0, q_1, \ldots, q_n, \ldots$ de $\mathcal{A}$ tal que $a_n \in l(q_n)$, para cada $n \in \mathbb{N}_0$.

iv) Al conjunto de las palabras aceptadas por $\mathcal{A}$ se le llama *lenguaje* (aceptado) de $\mathcal{A}$ y se denota por $\mathcal{L}_\omega(\mathcal{A})$:

$$\mathcal{L}_\omega(\mathcal{A}) := \{\sigma \in \Sigma^\omega \mid \sigma \text{ es una } \omega\text{-palabra aceptada por } \mathcal{A}\}$$

Ejemplo 2.6.3. Sea el ABGE $\mathcal{A}_1 := (Q_1,\ Q_{01},\ \rho_1,\ \mathcal{F}_1,\ l_1)$ sobre $\Sigma = 2^{AP}$, con $AP = \{a, b\}$, donde:

$$\Sigma = 2^{AP} = \{\emptyset,\ \{a\},\ \{b\},\ \{a,\ b\}\}$$
$$Q_1 = \{q_0,\ q_1\}$$
$$Q_{01} = \{q_0,\ q_1\}$$
$$\rho_1 = \{(q_0,\ q_0),\ (q_0,\ q_1),\ (q_1,\ q_1)\}$$
$$\mathcal{F}_1 = \{\{q_0,\ q_1\}\}$$
$$l_1 \ = \{(q_0,\ \{a\}),\ (q_1,\ \{b\})\}$$

El ABGE $\mathcal{A}_1$ se puede representar mediante un grafo dirigido donde cada nodo representa un estado y cada arista la transición de un estado a otro. Los nodos se pueden etiquetar de acuerdo con la función de etiquetado y, adicionalmente, los estados iniciales se pueden representar con una flecha de entrada. La colección de aceptación no es conveniente representarla gráficamente; por lo tanto, se mostrará como tal en un extremo del digrafo. La figura 2.16 muestra la representación gráfica de $\mathcal{A}_1$. Para simplificar, escribimos a en lugar de $\{\{a\}\}$ y b en lugar de $\{\{b\}\}$.

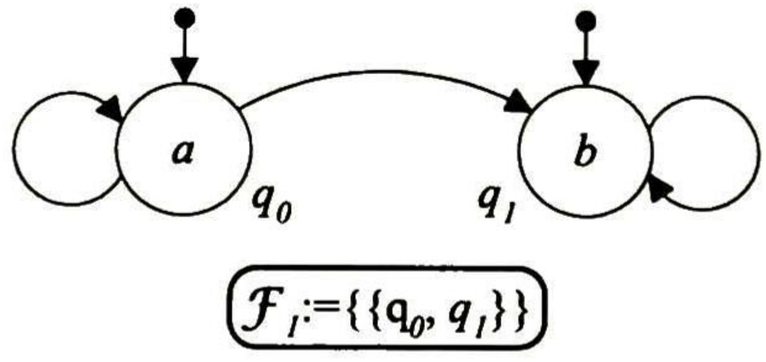


Figura 2.16: Autómata de Büchi generalizado etiquetado $\mathcal{A}_1$

Las corridas de aceptación en $\mathcal{A}_1$ son aquellas que pasan un número infinito de veces por alguno de los dos estados q_0 y q_1. Luego, el lenguaje de $\mathcal{A}_1$ es:

$$\mathcal{L}_\omega(\mathcal{A}_1) = \{\sigma \in \Sigma^\omega \mid \sigma \text{ se ajusta a la expresión } a^\omega + a^*b^\omega\}$$

Es importante mencionar que el lenguaje aceptado por un ABGE puede cambiar radicalmente si se cambia su colección de conjuntos de aceptación $\mathcal{F}$ Por ejemplo, observe lo que sucede si para el autómata del ejemplo anterior dicha colección se define como $\mathcal{F}_1 = \{\{q_0\}, \{q_1\}\}$. Ahora, las corridas de aceptación en $\mathcal{A}_1$ son aquellas que pasan un número infinito de veces por los estados q_0 y q_1, pero debido a la estructura del autómata cualquier corrida de aceptación que pase por el estado q_1 sólo pasará un número finito de veces por q_0. Por lo tanto, el lenguaje de $\mathcal{A}_1$ es vacío.

2.7 Lógica Temporal Lineal Proposicional

La lógica temporal lineal proposicional (LTLP) es una lógica en donde los valores de verdad de las proposiciones varían con respecto al tiempo. El tiempo es considerado como una línea que se extiende en una dirección (hacia el infinito) [36].

2.7.1. Sintaxis

El alfabeto de la LTLP consiste de los siguientes símbolos:

- Un conjunto contable de proposiciones atómicas $\mathcal{P} = \{p_0, p_1, \ldots\}$
- Conectivos booleanos: $\vee$ (disyunción), $\neg$ (negación).

- Conectivos temporales: $\bigcirc$ (siguiente), $\mathcal{U}$ (hasta que).
- Constante lógica: $\top$ (verdad).
- Símbolos de agrupación: (,) (paréntesis).

Con estos símbolos se forman cadenas finitas llamadas expresiones y sólo se toma un subconjunto, el de las fórmulas, que se define de manera inductiva a continuación.

Definición 2.7.1. Fórmulas de la LTLP y Conectivos Derivados

El conjunto de las *fórmulas* de la LTLP es el menor de los conjuntos de expresiones $\mathcal{X}$, sobre el alfabeto dado, que cumple con las condiciones siguientes:

- Todo elemento de $\mathcal{P}$ y la constante $\top$ pertenecen a $\mathcal{X}$
- Si $\varphi, \psi \in \mathcal{X}$, entonces también $(\neg\varphi), (\varphi \vee \psi), (\bigcirc\varphi), (\varphi \mathcal{U} \psi) \in \mathcal{X}$

Este conjunto mínimo se denota por Ψ.

A lo largo de este trabajo, por simplicidad, se omitirán los paréntesis externos de las fórmulas.

Como los conectivos booleanos $\neg$ y $\vee$ constituyen un conjunto funcionalmente completo para la lógica proposicional, podemos definir los demás conectivos en términos de éstos:

- $p \rightarrow q := \neg p \vee q$
- $p \wedge q := \neg(\neg p \vee \neg q)$
- $p \leftrightarrow q := (p \rightarrow q) \wedge (q \rightarrow p)$

También definimos las modalidades $\Box$, $\Diamond$ y $\mathcal{V}$, basadas en $\mathcal{U}$:

- $\Diamond\varphi := \top \mathcal{U} \varphi$
- $\Box\varphi := \neg\Diamond\neg\varphi$
- $\varphi \mathcal{V} \upsilon := \neg(\neg\varphi \mathcal{U} \neg\psi)$

Las fórmulas que son proposiciones atómicas o negaciones de proposiciones atómicas se llaman *literales*[3] A las fórmulas de la forma $\bigcirc\varphi$, donde $\varphi \in \Psi$, se les conoce como *fórmulas de estado siguiente* y las fórmulas que no poseen operadores temporales se conocen como *fórmulas de estado.*

2.7.2. Semántica

La LTLP es vista como una línea seccionada en instantes de tiempo separados por intervalos constantes. Una fórmula de la LTLP puede tomar el valor de verdadera o falsa dependiendo del conjunto de proposiciones atómicas existentes en el instante de tiempo en el que se evalúe.

Definición 2.7.2. Modelo

Un Modelo de la LTLP es una función $\mathcal{M} : \mathbb{N}_0 \rightarrow 2^{\mathcal{P}}$ Esto es, $\mathcal{M}$ es una sucesión infinita de la forma $P_0, P_1, P_2, \ldots, P_k, \ldots$, donde cada P_k es un subconjunto de $\mathcal{P}$. Intuitivamente, un modelo $\mathcal{M}$ describe la manera en que cambia el valor de verdad de un grupo de proposiciones atómicas con el avance lineal del tiempo. Por eso a las proposiciones atómicas que pertenezcan al conjunto P_k del modelo $\mathcal{M}$ se les considera verdaderas en el instante k y como falsas, en ese mismo instante, a aquellas que no pertenezcan a P_k.

Definición 2.7.3. Satisfacción de una Fórmula de la LTLP

Dados un modelo $\mathcal{M}$ y una fórmula φ de la LTLP, la expresión

$$\mathcal{M}, i \models \varphi$$

afirma que "φ es verdadera en el instante de tiempo i en el modelo $\mathcal{M}$" Esta noción se define de acuerdo con la estructura de la fórmula φ, como sigue:

- $\mathcal{M}, i \models \varphi$ si y sólo si $\varphi \in \mathcal{M}(i)$, para cada $\varphi \in \mathcal{P}$
- $\mathcal{M}, i \models \neg\varphi$ si y sólo si $\mathcal{M}, i \not\models \varphi$
- $\mathcal{M}, i \models \varphi \vee \psi$ si y sólo si $\mathcal{M}, i \models \varphi$ ó $\mathcal{M}, i \models \psi$

[3] También se consideran como literales a $\top$ y $\neg\top$

- $\mathcal{M}, i \models \bigcirc \varphi$ si y sólo si $\mathcal{M}, i+1 \models \varphi$
- $\mathcal{M}, i \models \varphi \mathcal{U} \psi$ si y sólo si existe $k \geq i$ tal que $\mathcal{M}, k \models \psi$, y para toda j tal que $i \leq j < k$ se cumple $\mathcal{M}, j \models \varphi$
- $\mathcal{M}, i \models \Diamond \varphi$ si y sólo si existe $k \geq i$ tal que $\mathcal{M}, k \models \varphi$
- $\mathcal{M}, i \models \Box \varphi$ si y sólo si para todo $k \geq i$ se cumple $\mathcal{M}, k \models \varphi$
- $\mathcal{M}, i \models \varphi \mathcal{V} \psi$ si y sólo si para todo $k \geq i$ tal que $\mathcal{M}, k \not\models \psi$ existe $i \leq j < k$ tal que $\mathcal{M}, j \models \varphi$

Definición 2.7.4. Satisfacibilidad y Validez

Sean $\mathcal{M}$ un modelo y φ una fórmula de la LTLP Se dice que:

- $\mathcal{M}$ *satisface* a φ si y sólo si existe un $i \in \mathbb{N}_0$ tal que $\mathcal{M}, i \models \varphi$
- φ es *satisfacible* si y sólo si existe un modelo $\mathcal{M}$ que satisfaga a φ
- φ es *válida*, y se escribe $\models \varphi$, si y sólo si φ es satisfacible en todo modelo de la LTLP

Una consecuencia sencilla pero importante es que una fórmula φ es válida si y sólo si la fórmula $\neg\varphi$ no es satisfacible.

Por otro lado, observe que si una fórmula φ es satisfacible, entonces existe un modelo $\mathcal{M}$ con un instante i donde $\mathcal{M}, i \models \varphi$, y por lo tanto, también existe otro modelo $\mathcal{M}'$ que comienza en el instante i, donde $\mathcal{M}', 0 \models \varphi$. Por esta razón se puede concluir que *una fórmula es satisfacible en algún modelo si y sólo si es satisfacible en el instante inicial 0 de otro modelo.* De aquí que se puede omitir la representación de los instantes y escribir únicamente $\mathcal{M} \models \varphi$ para indicar que φ es satisfacible en el modelo $\mathcal{M}$.

Capítulo 3

Modelado de Sistemas PLC Descritos Mediante LE.

3.1. Contenido del Capítulo

En este capítulo se discute la importancia de tener un modelo correcto del sistema y se explica el método de traducción propuesto para generar un modelo del sistema, así como las diversas partes que soporta. En la sección 3.2 recordamos las partes de la verificación y resaltamos la importancia de haber hecho un modelo correcto del diseño del sistema. Definimos en la sección 3.3 la estructura del método propuesto de traducción para una sintaxis restringida del LE. En esa misma sección explicamos cómo se realiza construcción incremental del modelo. En la sección 3.4 definimos las diversas secciones que soporta el método de traducción, en la subsección 3.4.2 mostramos la importancia de definir un estado inicial del sistema y explicamos la manera en cómo se aplica en el algoritmo de construcción incremental. En la subsección 3.4.3 aplicamos una técnica para acercar más el modelo del diseño con el sistema real, lo cual nos permite obtener sistemas más fieles a la realidad. En la subsección 3.4.5 señalamos que en los ABGEs es necesario especificar estados marcados si queremos asegurar un cierto comportamiento a verificar. En la subsección 3.4.6 mostramos la utilidad de restringir la existencia de ciertos estados que no deben aparecer en nuestro sistema ya sea por motivos de seguridad o funcionamiento en sí. Por último en la sección 6

mostramos algunos detalles observados en el modelado de sistemas.

3.2. Modelado de Sistemas

En la verificación formal por comprobación de modelos descrita en [43] el proceso de verificación lo podemos esquematizar como se muestra en la figura 3.1. En general

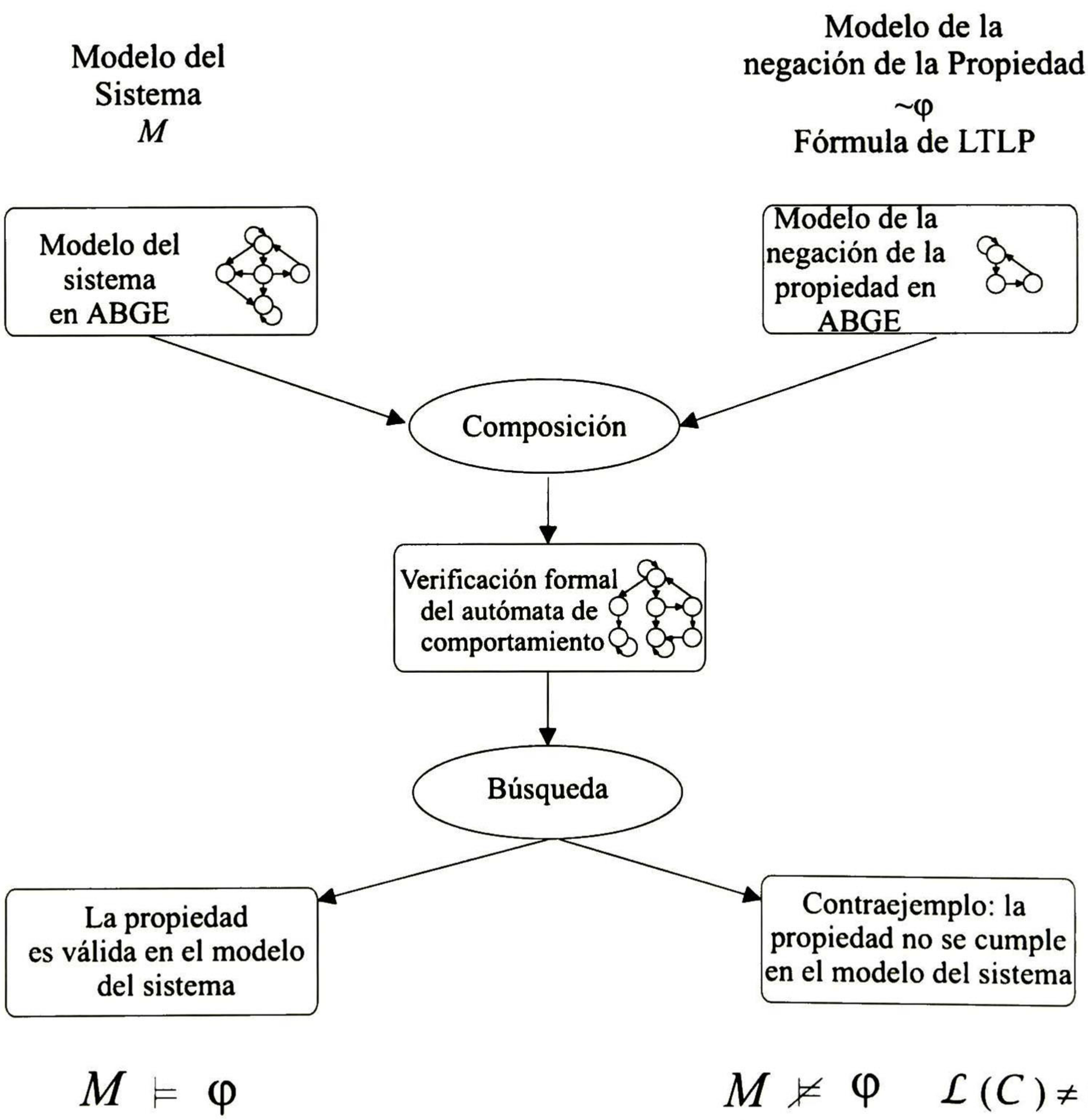


Figura 3.1: Componentes de la verificación por comprobación de modelos.

la verificación formal por comprobación de modelos se divide en tres partes:

- Modelado
- Especificación
- Prueba

En caso de que la propiedad con la que se ha hecho la prueba no se cumple en el modelo del sistema se obtiene un contraejemplo. Ese contraejemplo pudo haberse generado debido a alguna de las siguientes razones:

1. Diseño incorrecto
2. Modelo incorrecto
3. Especificación incorrecta

Durante el proceso de verificación lo que nos interesa es saber si nuestro diseño es correcto funcionalmente y no preocuparnos por si el modelo del diseño está bien hecho o no. Es fundamental que el modelo del sistema realmente refleje el comportamiento del diseño. Modelar el sistema es una tarea que en muchas ocasiones se hace de manera automática simplemente sincronizando los elementos que lo componen o mediante algún método de síntesis. Pero en ocasiones esto no es suficiente y se deben hacer correcciones manuales debido a que algunos comportamientos que no tiene el sistema se han agregado.

En este trabajo nos enfocamos principalmente en la tarea de generar un modelo de un sistema a través de un proceso de traducción. El tipo de sistemas a los que nos enfrentamos son los controladores lógicos descritos mediante una sintaxis restringida del lenguaje de escalera.

3.3. Método de Traducción y Generación del Modelo

La sintaxis restringida que utilizamos para modelar, tomada del lenguaje de escalera es:

- Contactos normalmente abiertos.
- Contactos normalmente cerrados.
- Bobinas.
- Bobinas negadas.
- Contactos en paralelo.
- Contactos en serie.
- Asignación a múltiples bobinas.
- Asignación dinámica de variables. Las variables pueden ser asignadas varias veces dentro del programa.
- Retroalimentación de trayectoria.
- Se trabaja únicamente con variables booleanas.

En general, el ciclo de un programa PLC se realiza en tres pasos, como se muestra en la figura 3.2, en el primer paso se hace una lectura de los valores de las señales de entrada y se asignan a sus variables correspondientes, en el segundo paso se ejecuta la lógica de control (que constituye el programa) descrita por la manera en que los elementos fueron enlazados en el diagrama de escalera y por último se actualizan los valores de las variables que se modificaron en el programa y se hace la activación de las salidas correspondientes.

Un programa PCL puede verse como una cuádrupla de la forma $\mathcal{P} = (I,\ O,\ B,\ \Gamma)$ donde I es el conjunto de variables de entrada, $I = \{i_1,\ i_2, \ldots,\ i_j\}$, O es el conjunto de variables de salida, $O = \{o_1,\ o_2, \ldots,\ o_k\}$, B es el conjunto de variables internas del programa, $B = \{b_1,\ b_2, \ldots,\ b_l\}$ y Γ es el conjunto de escalones que componen el programa, $\Gamma = \{\gamma_1,\ \gamma_2, \ldots,\ \gamma_i\}$, para $j,\ k,\ l \in \mathbb{N}$.

Las variables de entrada se asocian a los contactos, las variables de salida y las internas se asocian a bobinas (pueden ser utilizadas también en contactos pero su referencia es a una bobina).

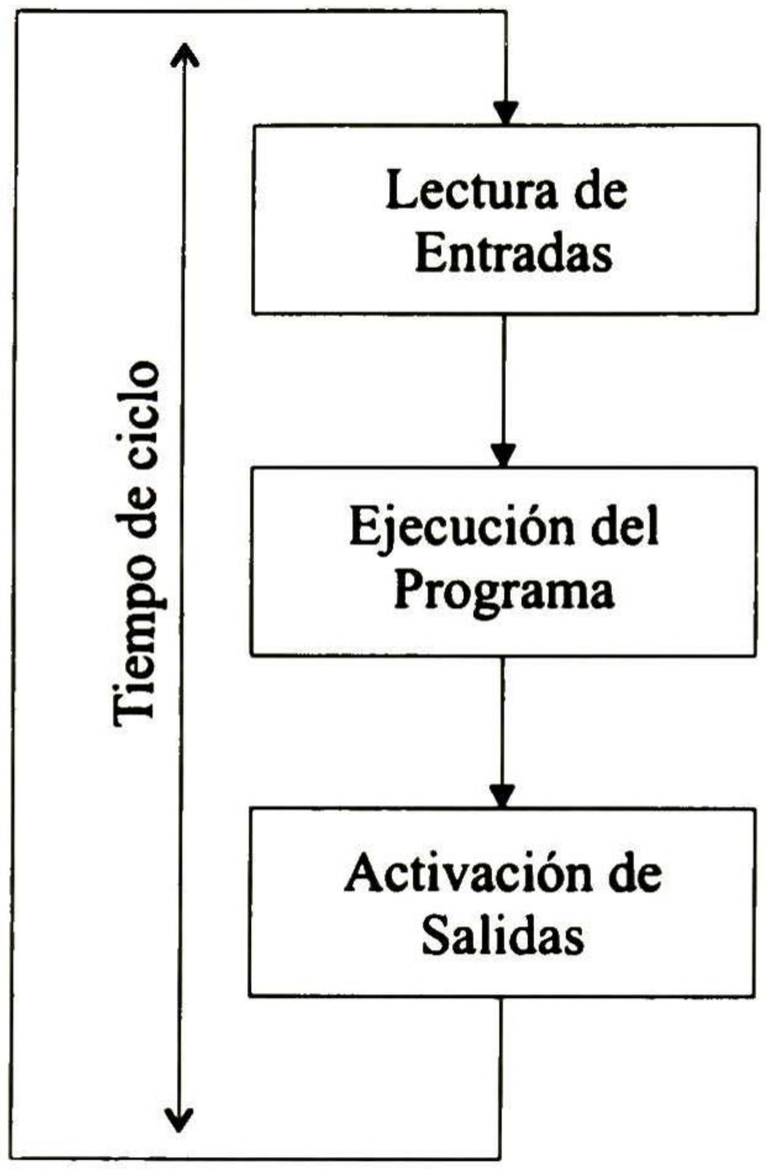


Figura 3.2: Ciclo de ejecución de un programa PLC.

Cada contacto que se encuentra conectado en serie con otro corresponde a una operación lógica de conjunción entre sus variables asociadas y cada contacto que se encuentra conectado en paralelo a otro corresponde a una operación lógica de disyunción entre sus variables asociadas. Las variables correspondientes a cada uno de los contactos serán negadas si el contacto corresponde a uno normalmente cerrado y serán sin negar si corresponde a uno normalmente abierto. La manera en que los elementos de cada escalón del diagrama de escalera son enlazados le llamaremos *lógica del escalón.*

Cada variable de entrada toma su valor booleano de las entradas externas del sistema suministradas por elementos como botones, sensores, etc. Cada variable de salida o interna toma su valor booleano de la ejecución de la lógica del escalón con los valores de las variables de entrada, internas y de salida que intervengan en dicho escalón

$$o_k \Leftrightarrow \gamma_i(I,\ O,\ B) \text{ ó } b_l \Leftrightarrow \gamma_i(I,\ O,\ B).$$

Cada γ_i puede ser vista como una función que puede tomar los valores del conjunto $\mathbb{B} = \{0,\ 1\}$.

3.3.1. Traducción del Lenguaje de Escalera a un ABGE

La generación del modelo consiste en una traducción del diagrama de escalera asociado al controlador del sistema a un sistema de *ecuaciones de estado siguiente* y de allí a un ABGE. Una vez obtenido el ABGE correspondiente podremos realizar la verificación formal del modelo. Introducimos algunas hipótesis para la generación del modelo. La primera es que el funcionamiento del PLC es como un circuito síncrono, en donde se hace una lectura de las entradas y ellas se mantienen a lo largo de la ejecución del programa, una vez terminada la ejecución se vuelven a leer las entradas y se comienza un nuevo el ciclo de ejecución del programa.

Modelamos un sistema como un ABGE $\mathcal{A} = (Q,\ Q_0,\ \rho,\ \mathcal{F},\ l)$ sobre un conjunto $\Sigma = 2^{AP}$, donde $AP = \{p_1,\ p_2, \ldots,\ p_n\}$ es un conjunto de proposiciones atómicas (variables que conforman $I \cup O \cup B$). Cada estado caracteriza un ciclo de ejecución del programa en donde las variables de entrada se leen al inicio y no cambian durante la ejecución del programa, al final de la ejecución de se actualizan los valores de las variables de salida que definen el estado siguiente. Cada estado q del modelo $\mathcal{A}$ es visto como una asignación de valores de verdad tomados del conjunto $\mathbb{B} = \{0,\ 1\}$; esto es, q es una función de AP en $\mathbb{B}$. De este modo, interpretamos $q(p_i) = 1$ como que p_i es una afirmación verdadera (o que se cumple) en el estado q. Esto nos permite representar cada estado q de $\mathcal{A}$ como un vector booleano de dimensión igual a la cardinalidad de AP, digamos que $q := (v_1,\ v_2, \ldots,\ v_n)$, donde

$$v_i = \begin{cases} 1 & \text{si y sólo si } p_i \text{ se cumple en } q \text{ para cada } i = 1,\ 2, \ldots,\ n. \\ 0 & \text{de otro modo.} \end{cases}$$

Consecuentemente $Q = \mathbb{B}^n$ o un subconjunto de $\mathbb{B}^n$, dependiendo de si sólo queremos tomar en cuenta los estados alcanzables desde algún estado inicial. El subconjunto de los estados iniciales Q_0 de $\mathcal{A}$ se caracteriza mediante una condición expresada como una fórmula proposicional (ver 3.4.2). La función ρ asocia a cada estado el conjunto de sus estados sucesores. La función de etiquetado $l : Q \longrightarrow 2^{\Sigma}$ sirve para hacer

explícita las posibles colecciones de proposiciones atómicas que se cumplen en cada estado de $\mathcal{A}$ y por tanto se define mediante la regla siguiente:

$$l(q) := \{\{p_j \in AP \mid q(p_j) = 1, \text{ para } j = 1,\ 2, \ldots,\ n\}\}$$

Finalmente, $\mathcal{F}$ es una colección de conjuntos de estados con la cual decidimos cuáles corridas inicializadas describen comportamientos legales del sistema modelado por $\mathcal{A}$ (ver 3.4.5). Cada estado es caracterizado

Los pasos de la traducción son los siguientes:

1. Por cada γ_i del diagrama de escalera generar su *ecuación lógica de estado siguiente* correspondiente.

   Cada una de las γ_i se puede representar en forma de una ecuación lógica de estado siguiente, en donde cada variable de estado siguiente $v\_est\_s_i^+ \in \{O \cup B\}$[1] Recordemos que es posible hacer asignaciones a múltiples bobinas. Entonces la forma de la ecuación de estado siguiente es:

$$v\_est\_s^+_{1k+1} = \ldots = v\_est\_s^+_{ik+1} =$$
$$v_{1k+1}\ operador_{1k+1}\ v_{2k+1}\ operador_{2k+1} \ldots\ operador_{n-2k+1},\ v_{n-1k+1}$$
$$operador_{jk+1}\ v_{nk+1}$$

   Donde cada $v_n \in I \cup O \cup B$ y cada $operador_j$ corresponde a una operación de conjunción o disyunción, para $n,\ k \in \mathbb{N}$ y $j \in \{0,\ 1, \ldots,\ n-1\}$. Cada i corresponde al número de variables de estado siguiente asignadas simultáneamente en el escalón k. A la parte izquierda de la igualdad le llamaremos en lo sucesivo variable de estado siguiente y a la parte derecha le llamaremos ecuación de estado siguiente asociada, $e\_est\_s\_a$.

   La ecuación más básica que podemos generar está determinada por el diagrama que se compone por un contacto conectado directamente a una bobina. Otros diagramas básicos son cuando dos contactos se encuentran en serie ó en paralelo como se muestra en la figura 3.3. A partir de estos diagramas básicos podemos generar ecuaciones más complejas.

[1] Las variables de estado siguiente que estén asociadas a bobinas negadas estarán también negadas.

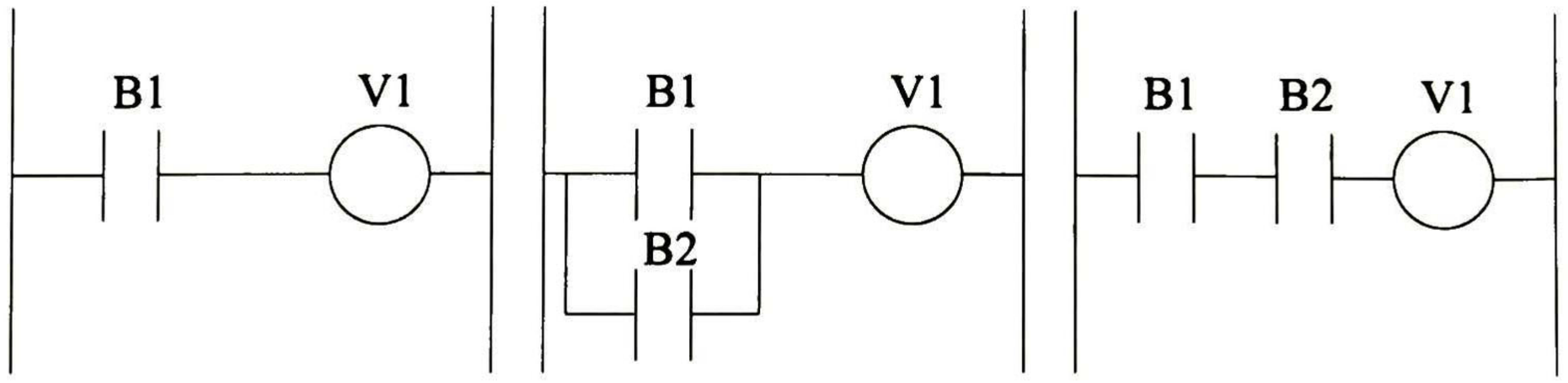


Figura 3.3: Las ecuaciones lógicas que representan los diagramas son: $V1^+ = B1$, $V1^+ = B1 \vee B2$, $V1^+ = B1 \wedge B2$, donde V1 es la variable asociada a la bobina, B1 y B2 son las variables asociadas a los contactos.

2. Generar los estados iniciales. El conjunto de estados iniciales Q_0 corresponde al inicio de operación del sistema y lo determinamos mediante la asignación de valores a un subconjunto de variables de AP (ver sección 3.4.2). Cada una de las diferentes asignaciones de valores a las variables de AP corresponden a un estado inicial del ABGE.

3. Generar los sucesores de los estados iniciales. A partir de los estados iniciales definidos en el ABGE debemos obtener sus sucesores mediante la evaluación de las ecuaciones de estado siguiente.

Para cada estado q de $\mathcal{A}$, el conjunto $\rho(q)$ consiste de los estados adyacentes desde q en $\mathcal{A}$ y les llamamos *sucesores* de q en $\mathcal{A}$. Si q' es un sucesor de q, entonces se escribe $q \longrightarrow q'$ ó $q' \in \rho(q)$.

La función de transición ρ está determinada por la siguiente regla:

- Para cualquier par de estados q, $q' \in Q$ tenemos que $q \longrightarrow q'$. si y sólo si

$$v\_est\_s^+_{1q'} \Leftrightarrow e\_est\_s\_a_{1q} \wedge v\_est\_s^+_{2q'} \Leftrightarrow e\_est\_s\_a_{2q} \wedge \ldots \wedge$$
$$v\_est\_s^+_{iq'} \Leftrightarrow e\_est\_s\_a_{iq}$$

donde cada $e\_est\_s\_a_{iq}$, para $i \in \mathbb{N}$, es la ecuación de estado siguiente (evaluada en el estado q) asociada a la variable de estado siguiente $v\_est\_s_{iq'}$ (en el estado q').

> La interpretación de la regla anterior es la siguiente: un estado q' es sucesor de un estado q si y sólo si al evaluar las ecuaciones de estado siguiente en q, el valor de cada ecuación es igual al valor de su variable asociada en q'

Una vez obtenidos los estados sucesores de los estados iniciales debemos obtener los sucesores de los sucesores obtenidos y así sucesivamente hasta completar el ABGE y llegar a un punto fijo en donde ya no agregamos más estados. Cabe señalar que el algoritmo para crear los estados sucesores hace una construcción incremental y cada estado que se adiciona pertenece a los estados que son alcanzables a partir del estado o estados iniciales definidos.

El algoritmo mostrado en la figura 3.4 fue implementado en la herramienta de construcción de modelos y representa el paso 3 de traducción: creación de los estados sucesores.

Al comienzo del algoritmo tenemos la lista de estados sucesores directos de los estados iniciales, *lista*, (estos fueron obtenidos mediante la aplicación del algoritmo que se muestra en la figura 3.7), el conjunto de ecuaciones lógicas de estado siguiente, *ecuaciones* y la colección de conjuntos de estados de aceptación, *marcados*. El resultado es el ABGE $\mathcal{A}$.

Por conveniencia en los ABGEs mostraremos en cada estado los vectores de bits de los valores de las variables en lugar de sus etiquetas.

El algoritmo termina una vez que se ha vaciado la *lista*. La complejidad del algoritmo es de orden exponencial. El peor de los casos es cuando tiene que generar todo el espacio de estados posible, que es $2^{|AP|}$ estados.

3.3.2. Ejemplo 1: Detector de Secuencias

Apliquemos el método de traducción al siguiente ejemplo sencillo que muestra un detector de secuencias, el cual está representado por el diagrama de la figura 3.5. La señal de entrada X recibe una secuencia de bits y la secuencia 101 es de aceptación.

Paso 1. Al aplicar la regla 1 generamos las siguientes ecuaciones que representan el diagrama de escalera:

- $V1^{+} = V2 \wedge \neg X$;

crearEstadosSucesores
1: **Entrada:** *lista*, *ecuaciones*, *marcados*
2: **Salida:** ABGE $\mathcal{A}$ correspondiente
3: **Mientras** *lista* no sea vacía **Hacer**
4: Tomar el primer elemento de *lista* y asignarlo al estado s
5: Agregar s al ABGE $\mathcal{A}$
6: Verificar si s es un estado marcado
7: **Si** s es un estado marcado
8: **Entonces**
9: agregar s a la lista de estados *marcados*
10: Evaluar *ecuaciones* con los valores de las variables del estado actual s.
11: Generar la plantilla de sucesores de s mediante la siguiente regla:
12: Para cualquier par de estados s, $s' \in S$ tenemos que $s \longrightarrow s'$ si y sólo si
$v\_est\_s_1^+ \Leftrightarrow e\_est\_s\_a_1 \wedge var\_estado_2^+ \Leftrightarrow e\_est\_s\_a_2 \wedge ... \wedge$
$v\_est\_s_n^+ \Leftrightarrow e\_est\_s\_a_n$
13: **Para cada** $i = 0$ hasta $i < 2^I$ **Hacer** //donde I es el número de variables de entrada
14: Completar s' //mediante la plantilla obtenida y el valor de i
15: **Si** la transición a s' es permitida //cumple precedencias y no es prohibido
16: **Entonces**
17: **Si** el estado s' se encuentra en *lista*
18: **Entonces**
19: generar una transición hacia s'
20: **Si no**
21: agregarlo a *lista* y generar una transición hacia s'
22: Borrar s de *lista*
23: Escribir los estados *marcados*
24: **Regresa** $\mathcal{A}$

Figura 3.4: Algoritmo para crear incrementalmente los estados sucesores del ABGE $\mathcal{A}$ a partir de los estados iniciales definidos en el paso 2 de traducción.

- $V2^+ = X$;

Paso 2. Definamos los estados iniciales como aquellos que contengan a las variables de estado siguiente desactivadas (valor 0 booleano). Como son dos variables de estado siguiente y una variable de estado, los estados iniciales que se generan son dos. Los estados iniciales se muestran en la tabla 3.1.

Paso 3. A partir de las ecuaciones lógicas de estado siguiente generamos los estados sucesores de los estados iniciales. Entonces, para aplicar el paso tres necesitamos las ecuaciones de estado siguiente y los estados iniciales:

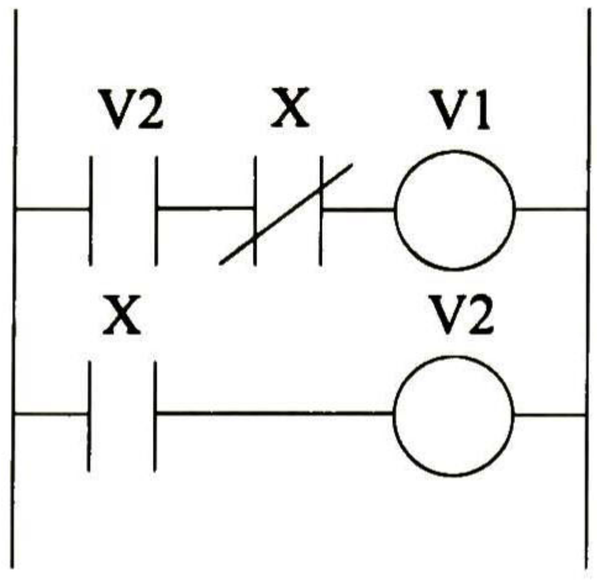


Figura 3.5: Diagrama de escalera de un detector de secuencia.

V2	V1	X
0	0	0
0	0	1

Tabla 3.1: Estados iniciales definidos para el detector de secuencias.

- $I = \{$ 000, 001$\}$.
- La expresión que define la función de transición ρ es:

$$(V1^{+} \Leftrightarrow (V2 \wedge \neg X) \wedge V2^{+} \Leftrightarrow X).$$

Al evaluar las ecuaciones de estado siguiente generamos las transiciones para cada estado, en donde el primer estado es el estado actual y el segundo es el estado sucesor:

{(000,000), (000, 001), (001,100), (001,101), (100, 010), (100, 011), (010, 001),
(010, 000), (011, 100), (011, 101),(101,010),(101,101)}

Para completar el ABGE tenemos:

- El espacio de estados generado es: $S = \{000, 001, 100, 010, 011, 110\}$
- La colección de conjuntos de aceptación en este ejemplo se forma por un sólo conjunto que contiene al estado en donde se completa la secuencia de bits 101 requerida por el detector de secuencias. La colección es: $\mathcal{F} = \{\{011\}\}$

- La función de etiquetado es: $l = \{ (000, \{\emptyset\}), (001, \{X\}), (101, \{V2, X\}), (100, \{V2\}), (011, \{V1, X\}), (010, \{V1\}) \}$

Nótese que no se pueden alcanzar los estados 111, 110 partiendo de los estados iniciales señalados.

El ABGE que representa el diagrama de escalera se muestra en la figura 3.6.

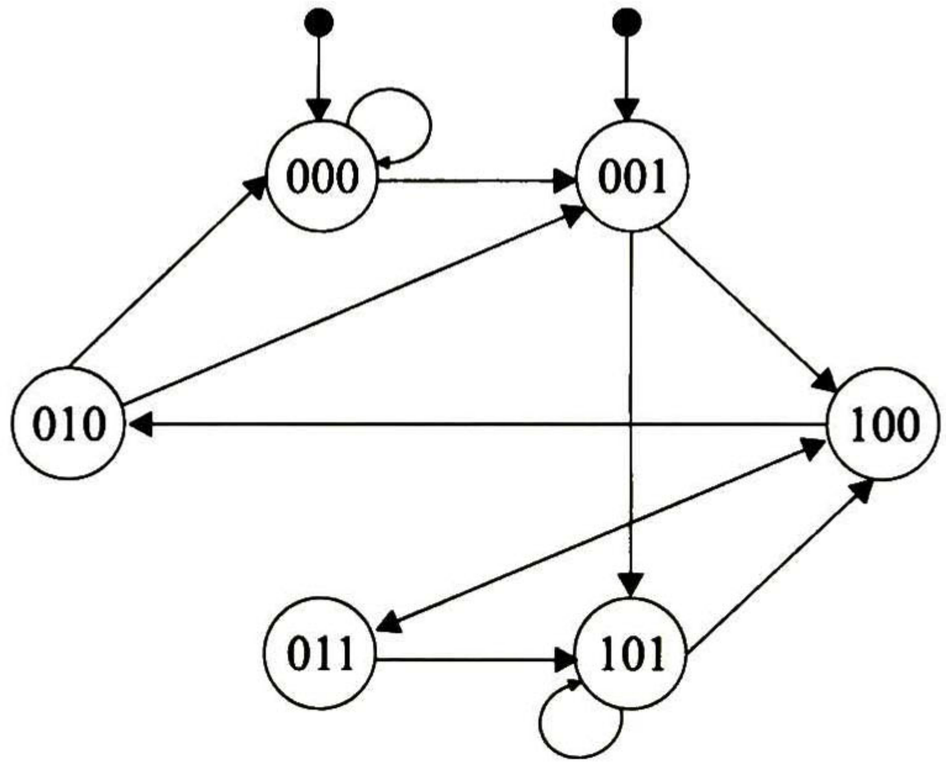


Figura 3.6: ABGE asociado al detector de secuencias.

En las secciones posteriores explicaremos brevemente las partes que componen el método de modelado y explicaremos como es que cada parte ayuda en la construcción del modelo del controlador dado en lenguaje de escalera.

Mostraremos el algoritmo básico que implementa cada sección de la herramienta. Para un conocimiento más detallado de la herramienta de modelado véase el Apéndice A.

3.4. Secciones del Método de Traducción

El método básico de traducción se compone del algoritmo para generar los estados iniciales y del algoritmo para crear los estados sucesores. Adicionales a esos algoritmos

se han incorporado algunas secciones para tomar en cuenta cierto conocimiento útil del sistema que nos permitirá construir modelos más apegados a su funcionamiento real.

La herramienta para generar los modelos se compone de las siguientes secciones, que explicaremos con detalle más adelante:

1. **Ecuaciones.** Se determinan en esta sección las ecuaciones de estado siguiente que conforman los escalones del diagrama de escalera.
2. **Estados Iniciales.** Se definen en esta sección los estados iniciales del sistema.
3. **Estados Marcados.** Se definen en esta sección la colección de conjuntos de estados considerados como marcados o de aceptación.
4. **Estados Prohibidos.** Se definen en esta sección los estados prohibidos del sistema.
5. **Precedencias.** Se definen en esta sección las precedencias de variables dentro del sistema.

Todas las secciones son opcionales a excepción de la sección **ecuaciones** la cual representa los escalones del diagrama de escalera. Si sólo se especifica esta sección se aplicarán únicamente los algoritmos para generar los estados iniciales y para crear sus sucesores.

3.4.1. Ecuaciones

La sección ecuaciones es obligatoria y en ella se escriben las ecuaciones de estado siguiente que representan el diagrama de escalera asociado al controlador del sistema. En la herramienta de modelado se implementó un analizador sintáctico y semántico con las herramientas [33] para realizar la lectura del archivo de entrada. La función de esta sección es evaluar las ecuaciones lógicas en el estado actual, con los valores asociados a cada variable en dicho estado y obtener los valores de las variables en el estado siguiente. Se efectúa este proceso de la misma manera que se realiza el ciclo del PLC.

En la figura 3.14 mostramos un ejemplo típico de la sección ecuaciones. En este caso las ecuaciones representan el funcionamiento de dos bombas alternantes que se detallará más adelante. Se utiliza la palabra reservada **ecuaciones** para delimitar la sección y las ecuaciones se delimitan por medio de llaves. Cada ecuación termina en un punto y coma. Los conectivos lógicos que se utilizan para formar las ecuaciones son la negación, la conjunción y la disyunción.

3.4.2. Estados Iniciales

En los sistemas es necesario especificar uno o varios modos iniciales de operación. En esta configuración inicial de operación los componentes se encuentran en determinado estado para poder comenzar con la operación del sistema. Por ejemplo, las válvulas están cerradas, el motor apagado y el botón de mando desactivado.

La determinación de los estados iniciales en la parte de modelado debe ser fiel al estado inicial de operación del sistema. La conformación de un estado inicial está dada por el valor booleano de las variables de los componentes del sistema que forman parte del diagrama de escalera y de las variables internas.

A partir de un estado inicial podemos saber cuáles son las rutas de operación que podemos seguir mediante las diversas entradas y la ejecución de la lógica del programa. Si determinamos de manera errónea los estados iniciales podemos obtener un modelo que no corresponda con la operación real del sistema.

En los sistemas se pueden especificar varios estados iniciales, dependiendo de la aplicación que se está manejando o del modo de operación que estemos modelando. En la herramienta desarrollada es posible determinar uno o varios estados iniciales mediante la asignación de valores a las variables involucradas. En caso de que deseemos modelar el estado inicial como aquél en donde todas las variables se encuentran desactivadas, la herramienta asignará el valor booleano 0 a todas las variables sin necesidad de hacerlo de forma explícita, es decir se puede omitir en este caso la sección de estados iniciales. Si por el contrario, deseamos señalar que el estado inicial está determinado por una señal de entrada, y ésta puede tomar el valor de 0 ó 1 en dicho estado, entonces necesitamos especificar el valor inicial de las demás variables y dejar

sin especificar el valor de esa señal de entrada. Esa señal asumirá automáticamente los valores de 1 y 0 junto con los valores especificados para las demás variables. En ese caso obtendremos dos estados iniciales. Asimismo podemos dejar sin especificar las variables que deseemos y se generarán tantos estados iniciales como 2^k siendo k el número de variables sin especificar.

En la figura 3.7 mostramos el algoritmo que se encarga de la generación de los estados iniciales. A las variables que se les asigna explícitamente un valor en la definición del estado inicial les llamamos variables iniciales. Al resto llamamos variables libres. El algoritmo fue implementado en la herramienta de construcción de modelos elaborada y representa el paso 2 de traducción: creación de los estados iniciales.

Al comienzo del algoritmo tenemos únicamente el número de variables iniciales $nIniciales$, el número de variables totales $nVar$ y el conjunto de ecuaciones lógicas de estado siguiente *ecuaciones*. El resultado son los estados iniciales del ABGE $\mathcal{A}$ y la lista de estados sucesores directos, *lista*, de los estados iniciales.

Una vez que se ha aplicado el algoritmo para construir los estados iniciales se aplica el algoritmo para crear los estados sucesores descrito en la figura 3.4.

Los algoritmos de las secciones restantes del método de modelado están incorporados como funciones en los algoritmos de crear estados iniciales y crear estados sucesores, tales funciones preguntan si es marcado un estado o si la transición para crear un sucesor es permitida.

Por conveniencia en la siguiente subsección explicaremos la parte correspondiente a las precedencias.

3.4.3. Precedencias

En muchas ocasiones los componentes del sistema están de cierta forma enlazados, ya sea que un componente realiza su función justamente después que otro y no antes. O que necesitamos que algunas señales de entrada tengan un valor específico antes que otras. Estas situaciones muchas veces no son fácilmente modeladas de forma automática y se necesita de cierto esfuerzo extra para poder incorporar este conocimiento del funcionamiento del sistema. En el método de modelado propuesto

```
crearEstadosIniciales
1:  Entrada: nIniciales, nVar, ecuaciones
2:  Salida: A, lista
3:  Si nIniciales == 0 ó nVar − nIniciales == 0
4:  Entonces
5:      Asignar el valor 0 a las variables del estado s
6:      Agregar el estado s a A
7:      Verificar si s es un estado marcado
8:      Si s es un estado marcado
9:      Entonces
10:         agregar s a la lista de estados marcados
11:     Evaluar las ecuaciones en el estado inicial
12:     Agregar los sucesores de s a lista//mediante la regla de estados sucesores
13:     Regresa A, lista
14: Si no
15:     j = 2^(nVar−nIniciales)
16:     Para cada h = 0 hasta h < j Hacer
17:         Generar s asignando el valor de las variables iniciales y de las variables libres
18:         Agregar el estado s a A
19:         Verificar si s es un estado marcado
20:         Si s es un estado marcado
21:         Entonces
22:             agregar s a la lista de estados marcados
23:         Evaluar las ecuaciones en el estado s
24:         Agregar los sucesores de s a lista
25: Regresa A, lista
```

Figura 3.7: Algoritmo para crear los estados iniciales del ABGE $\mathcal{A}$ y la lista de estados sucesores directos *lista* de los iniciales

tratamos que parte de dicho conocimiento sea incorporado de manera automática a los modelos generados por la herramienta. A este conocimiento le llamaremos precedencias de variables y la idea básica es tomada de [13].

Una segunda hipótesis que introducimos en la generación del modelo, específicamente en la parte de precedencias, es que suponemos que las condiciones de operación del sistema son las ideales y que no se esperan fallas en el sistema, de este modo podemos suponer que los componentes del sistema trabajan adecuadamente.

Por medio de éstas relaciones entre variables podemos obtener un modelo del sistema más fiel al funcionamiento del sistema. En consecuencia tenemos un modelo que es más exacto funcionalmente. La incorporación de las precedencias de variables nos

permite obtener modelos más pequeños porque evitamos modelar comportamientos que realmente nunca aparecen en el sistema.

Podemos identificar dos tipos de precedencias a las que llamaremos:

1. **Precedencia Débil.**

2. **Precedencia Fuerte.**

Las precedencias son condiciones que se deben cumplir antes de que suceda un evento. Las precedencias operan al nivel de los valores de las variables del sistema.

Definimos dos tipos de precedencias de la siguiente forma:

$$var_1 = valor\_booleano\ tipo\_de\_precedencia\ var_2 = valor\_booleno$$

La variable var_1 corresponde a la variable cuyo valor booleano debe ser cumplido antes que el valor booleano de la variable var_2. Se dice que var_1 precede en la relación de precedencia. El término *valor_booleano* puede ser obtenido por medio de la evaluación de una ecuación lógica o por una asignación directa del valor booleano 0 ó 1 a var_1 y var_2.

La precedencia débil es aquella que obliga a que se cumpla el valor asignado a la variable var_1 antes de que pueda suceder el valor asignado a la variable var_2. Este comportamiento lo podemos observar en sistemas donde necesitamos que una señal se active antes de que se active otra. Por ejemplo, no debemos modelar la activación de la señal para desactivar un motor sin que antes haya habido una señal que lo activara.

En el método propuesto antes de saber si un estado es sucesor de otro debemos revisar las listas de precedencias definidas en el sistema.

El algoritmo para incorporar las precedencias al método de modelado se muestra en la figura 3.8. Las entradas de este algoritmo son: las listas de precedencias (tanto la débil como la fuerte) *lprecedencias*, el estado actual *EstadoActual* y el estado siguiente *EstadoSiguiente*. La salida es la determinación de si se cumple o no la precedencia. En caso de que sí se cumplan las relaciones de precedencias establecidas, se permite que el *EstadoSiguiente* sea sucesor del *EstadoActual*. En caso contrario, no se permite la transición y se evalúa el siguiente estado candidato, si queda alguno.

```
Transición
1:  Entrada: lprecedencias, EstadoActual, EstadoSiguiente
2:  Salida: hay_tranicion, no_hay_transicion
3:  aux=lprecedencias
4:  Mientras aux no sea vacía
5:      Si aux ∈ precfuerte
6:      Entonces
7:          v1 = aux->Nombre1 //variable que precede
8:          v2 = aux->Nombre2 //variable precedida
9:          Si valor de v2 = valor de variable correspondiente del EstadoSiguiente
10:         Entonces
11:             Si valor de v1 = valor de variable correspondiente del EstadoActual
                y valor de v1 = valor de variable correspondiente del EstadoSiguiente
12:             Entonces
13:                 aux=aux− >Siguiente //se evalúa la siguiente precedencia
14:             Si no
15:                 Si valor de variable correspondiente del EstadoActual = valor
                    de variable correspondiente del EstadoSiguiente
16:                 Entonces
17:                     aux=aux->Siguiente //se evalúa la siguiente precedencia
18:                 Si no
19:                     Regresa no_hay_transicion
20:         Si no
21:             aux=aux->Siguiente //se evalúa la siguiente precedencia
22:     Si no
23:         v1 = aux->Nombre1 //variable que precede
24:         v2 = aux->Nombre2 //variable precedida
25:         Si valor de v2 = valor de variable correspondiente del EstadoSiguiente
26:         Entonces
27:             Si valor de v1 = valor de variable correspondiente del EstadoActual
28:             Entonces
29:                 aux=aux->Siguiente //se evalúa la siguiente precedencia
30:             Si no
31:                 Si valor de variable correspondiente de EstadoActual = valor
                    de variable correspondiente del EstadoSiguiente
32:                 Entonces
33:                     aux=aux->Siguiente //se evalúa la siguiente precedencia
34:                 Si no
35:                     Regresa no_hay_transicion
36: Regresa hay_transicion //las precedencias se cumplieron
```

Figura 3.8: Algoritmo para verificar las precedencias definidas en el sistema.

La precedencia fuerte por su parte obliga a que se cumpla el valor asignado a la variable var_1 antes de que pueda suceder el valor asignado a la variable var_2 y además el valor de var_1 debe permanecer. Este comportamiento lo podemos observar en sistemas que tienen sensores en serie. Por ejemplo, si tenemos un sistema donde hay un tanque y hay tres sensores de nivel: bajo, medio, alto, el comportamiento esperado es que la activación del sensor de nivel bajo sea antes que la activación del nivel medio y que además el sensor de nivel bajo siga activado, y la misma relación sucede con el sensor de nivel medio y el sensor de nivel alto. De la misma forma podemos esperar que la desactivación del sensor de nivel alto sea antes que la desactivación del sensor de nivel medio, y lo mismo con el sensor de nivel medio y el sensor de nivel bajo.

Con base en resultados obtenidos, hemos visto que definir las precedencias del sistema nos reduce en gran medida el espacio de estados alcanzable.

En la siguiente subsección incluiremos un ejemplo para hacer notar la importancia de las precedencias, la elección de los estados iniciales y el método de construcción incremental.

3.4.4. Ejemplo 2: Bombas Alternantes

En esta aplicación, se desarrollará el programa en lógica de escalera para alternar bombas en procesos como el vaciar pozos, depósitos y tanques. En este tipo de aplicación, dos bombas pequeñas son frecuentemente utilizadas en lugar de una grande para reducir el costo de operación. El diagrama de tuberías e instrumentos se muestra en la figura 3.9.

La operación de bombeo alternante (inicialmente la bomba 1 como la primaria, después la bomba 2 como la primaria) reduce el mantenimiento requerido en las bombas individuales y provee una operación más confiable. En esta aplicación, la bomba secundaria o de reserva estará disponible si la tasa de agua entrante al tanque es mayor que la que la bomba primaria puede manejar. Si esta situación ocurre, la bomba secundaria también se encenderá para asistir a la primaria y juntas drenar el tanque. Los disparadores para estos eventos podrían ser señales analógicas o entradas discretas simples (interruptores de nivel, etc).

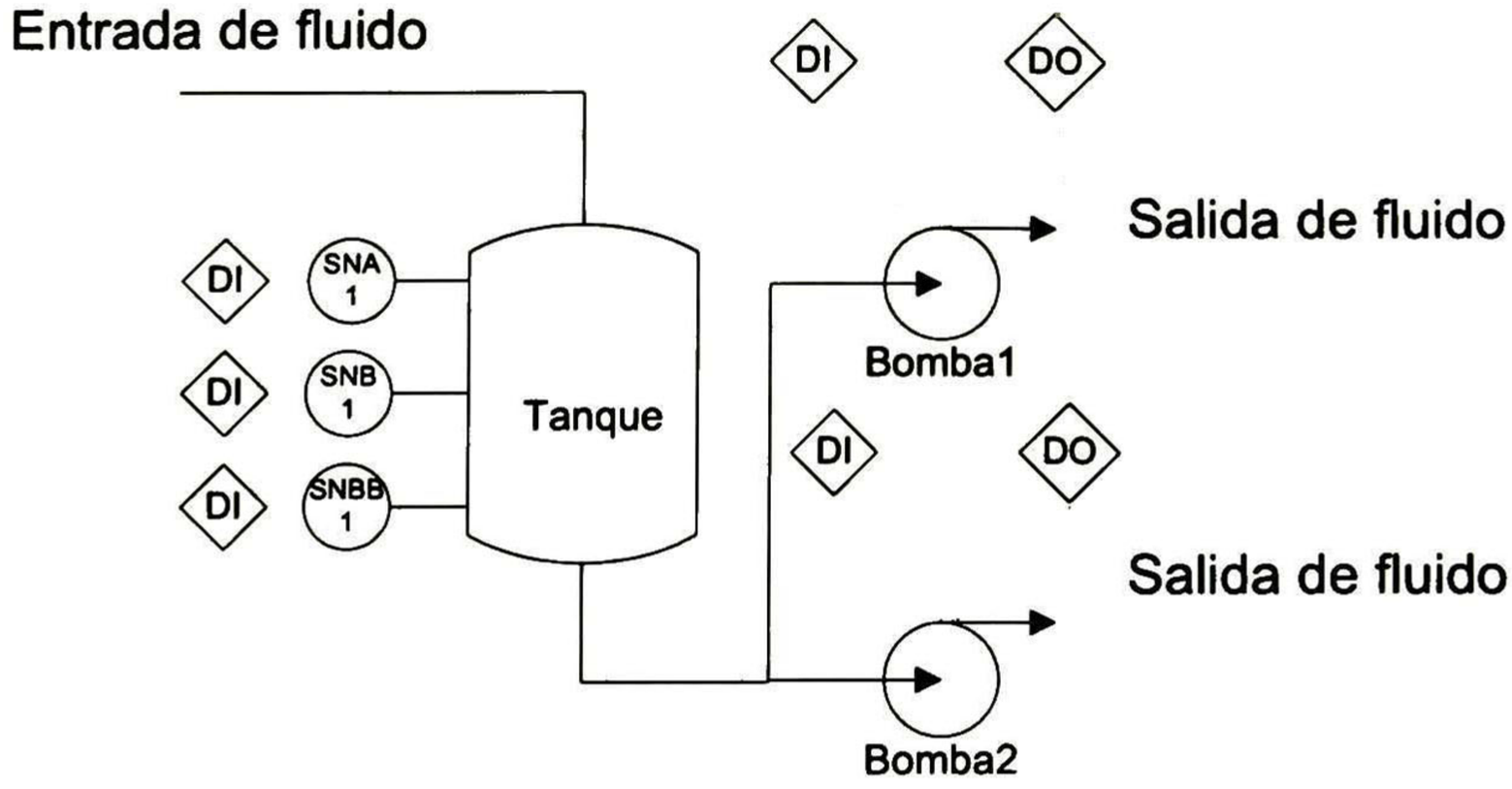


Figura 3.9: Diagrama de Tuberías e Instrumentos para el sistema de dos bombas alternantes

La lógica de escalera usada en esta aplicación consiste de 4 escalones. Los escalones 0 y 1 forman un circuito alternante, así que cada vez que el nivel del tanque es bajo, el interruptor SNBB-1 (sensor de nivel bajo-bajo) se encuentra cerrado. Éste pone el bit I:0/2 a 1 y el bit alternador en el escalón 1 B3:0/2 cambia de estado. El estado de este bit determina cual bomba encenderá primero. La instrucción OSR[2] en el escalón 0 es una instrucción especializada que es sólo energizada para un ciclo del procesador. Esto causa que el bit interno B3:0/1 sea energizado para un ciclo del procesador si el interruptor de nivel bajo-bajo, bit I:0/2, se encuentra cerrado como se muestra en la figura 3.10.

Cabe señalar que la instrucción especializada OSR no forma parte de la sintaxis restringida del lenguaje de escalera que maneja el método, pero se logró su implementación agregando una ecuación de estado siguiente y el comportamiento fue exactamente el mismo (se implementó en un PLC Siemmens S7 y se verificó que trabajara igual a como se describe). La figura 3.11 muestra como se implementó la lógica de escalera para sustituir la instrucción especializada OSR.

El escalón 2 controla la operación de la bomba 1, utilizando el bit de salida O:0/0

[2]Siglas de one shot rising

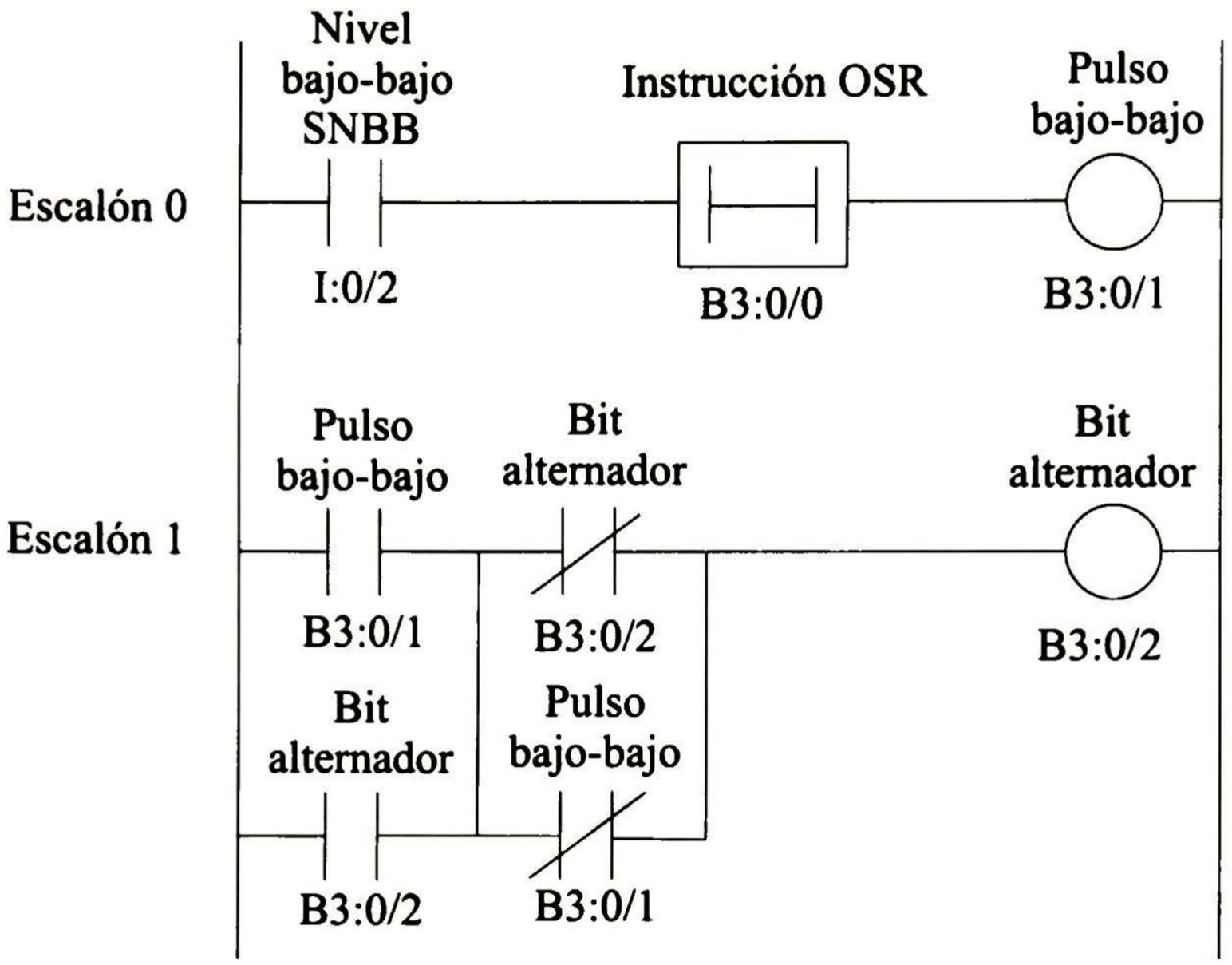


Figura 3.10: Escalón 0 y 1 que forman el circuito alternante de las bombas y que además indica cual bomba será la que inicie la operación en primer lugar.

en el PLC. Si el interruptor de nivel bajo-bajo está cerrado, éste pone el bit I:0/2 a 1. Si el bit interno alternador B3:0/2 está apagado y si el nivel del tanque ha alcanzado el interruptor de nivel SNB-1 (sensor de nivel bajo), poniendo el bit I:0/0 a 1, entonces la bomba 1 será la primer bomba en ser encendida. Si el bit alternador B3:0/2 está encendido, entonces la bomba 1 será la segunda en ser encendida. En la figura 3.12 se muestra el diagrama de escalera para controlar la operación de la bomba 1.

El escalón 3 controla la operación de la bomba 2 utilizando el bit de salida O:0/1. Si el interruptor de nivel bajo-bajo está encendido (bit I:0/2 está puesto en 1), el bit alternador B3:0/2 está encendido, y el nivel del tanque ha alcanzado el interruptor de nivel bajo (el bit I:0/1 está puesto en 1), entonces la bomba 2 será la primera en ser encendida. Si B3:0/2 está apagado, la bomba 2 será la segunda en ser encendida. El diagrama de escalera de los escalón 3 se muestra en la figura 3.13.

Las variables involucrados en el proceso se detallan en la tabla 3.2. Podemos observar que en la operación intervienen 8 variables, de las cuales 3 son variables

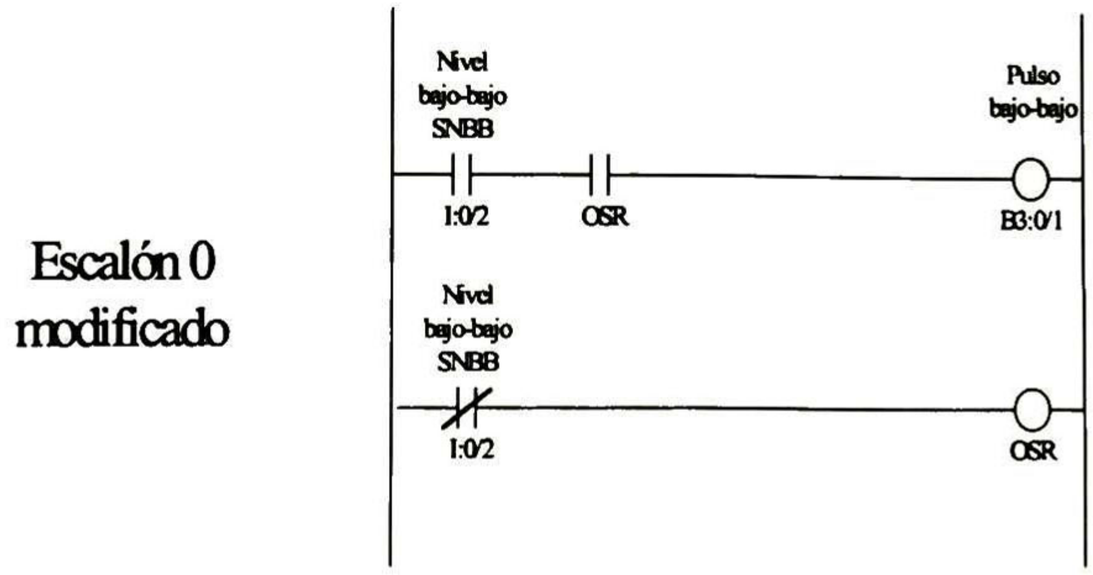


Figura 3.11: La instrucción especializada OSR se logró implementar agregando un escalón al diagrama y modificando el escalón 0.

internas, 3 son variables de entrada y 2 son salidas.

Una vez que se tiene la información del funcionamiento del sistema podemos inferir cierto comportamiento que en el diagrama de escalera no se muestra, tales como el estado inicial del sistema ó cuáles son las precedencias de variables que podemos establecer.

El primer paso en el método de modelado es traducir los escalones del diagrama de escalera a ecuaciones lógicas de estado siguiente. El segundo paso es definir el estado o estados iniciales del sistema y el tercer paso mostrado en este ejemplo es agregar algunas precedencias de variables.

La configuración del archivo de entrada para generar el modelo de las bombas alternantes en la herramienta desarrollada se muestra en la figura 3.14. En la primer sección se observan las ecuaciones lógicas de estado siguiente que representan los escalones del diagrama de escalera.

La sección estados iniciales muestra el estado inicial del sistema, en donde se detalla que todas las variables se encuentran desactivadas menos la variable que representa el sensor bajo-bajo, la cual queda como una variable libre. De acuerdo al algoritmo nos resultan dos estados iniciales. En la sección precedencias observamos la aplicación práctica de la técnica de precedencias. Las precedencias que utilizamos son del tipo fuerte, debido a la naturaleza del problema. Tenemos que los sensores de nivel se encuentra en serie y debemos representar su comportamiento de la misma forma en la

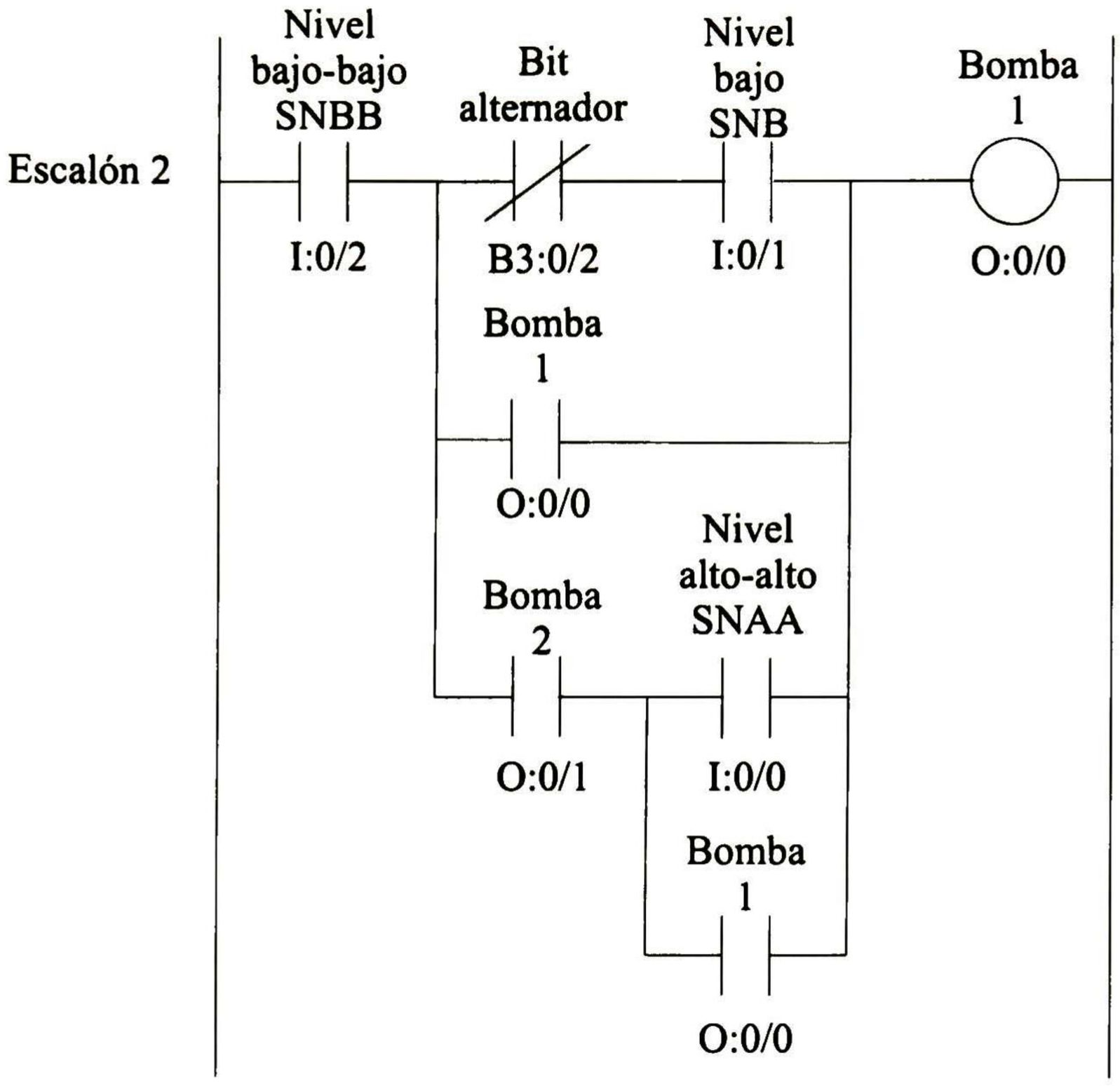


Figura 3.12: El escalón 2 se encargan de la operación de la bomba 1.

que trabajan en el sistema. El sensor de nivel bajo-bajo debe activarse y mantenerse activo antes que el sensor de nivel bajo se active y lo mismo sucede con los sensores de nivel bajo y alto. En el mismo sentido el sensor de nivel alto debe desactivarse antes que lo haga el sensor de nivel bajo y así mismo debe permanecer desactivado, y lo mismo ocurre para los sensores bajo y bajo-bajo.

Una vez definido el archivo de entrada para la herramienta de modelado el proceso de construcción es automático. Aplicando los algoritmos de construcción junto con él que se encarga de verificar las de precedencias generamos un modelo que tiene 32 estados y se muestra en la figura 3.15.

Un algoritmo ingenuo de construcción podría construir todo el espacio generado por las 8 variables involucradas, lo que genera 256 estados y a partir de cada estado

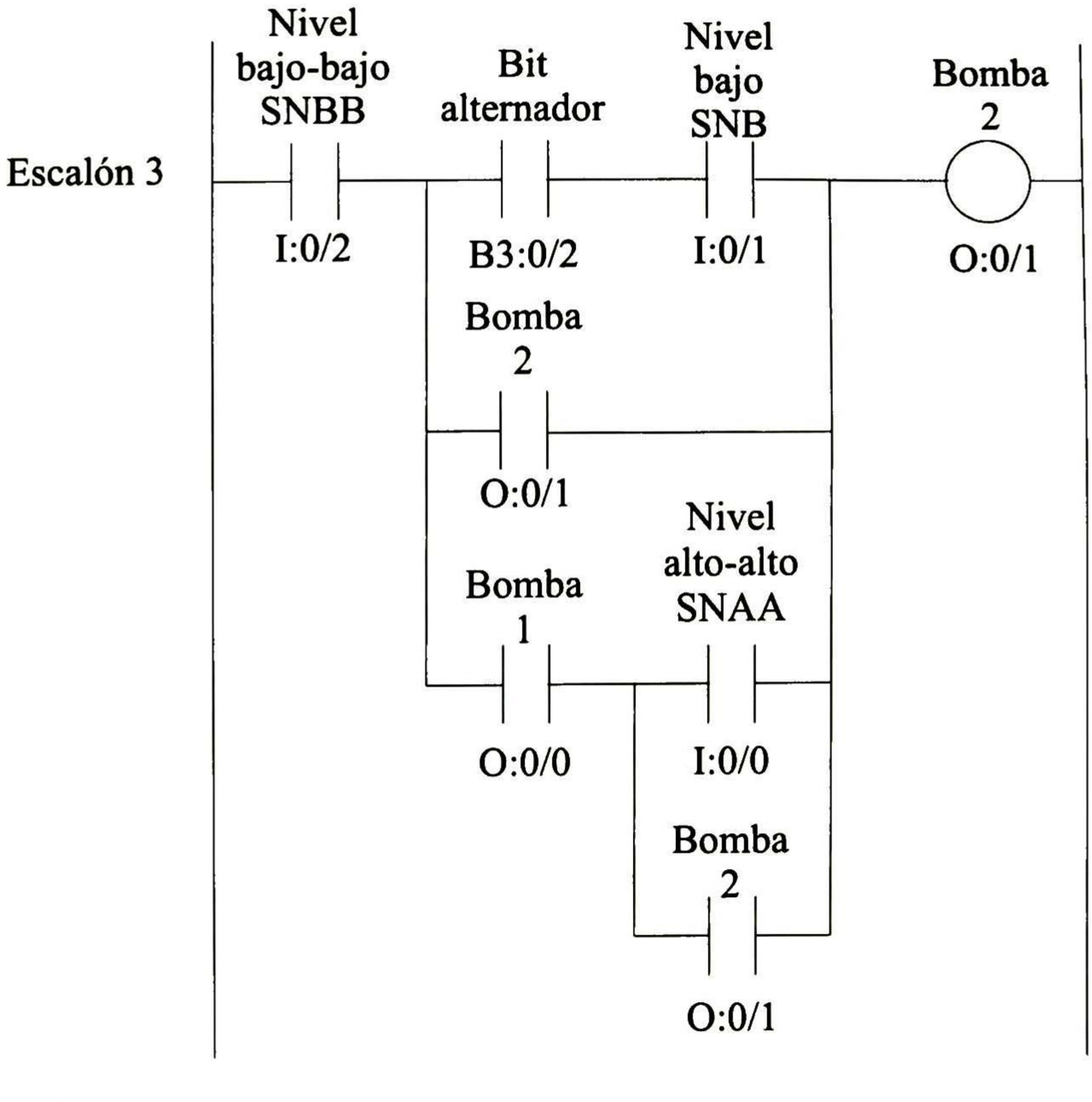


Figura 3.13: El escalón 3 se encargan de la operación de la bomba 2.

comenzar a buscar los sucesores y lanzar transiciones hacia ellos. Como son 3 variables de entrada tendríamos por cada estado 8 sucesores y en total serían 2048 transiciones. Y una vez generado ese modelo se debe hacer un análisis de alcanzabilidad para determinar cuales son realmente los estados a los que se puede llegar a partir de los estados iniciales.

Por otra parte si utilizamos nuestro algoritmo de construcción incremental sin especificar un estado inicial correcto y sin precedencias, por ejemplo decir que los estados iniciales van a ser todos aquellos donde el sensor de nivel bajo-bajo se encuentra activado entonces obtenemos 200 estados y 1600 transiciones, se obtiene una reducción del 22 % en el número de estados y transiciones.

Pero si generamos nuestro modelo en base al estado de operación en donde todas

Componente	Descripción	Señal	Nom. Etiq.
Var. Interna	Bit de pulso activado	B3:0/1	$B301$
Var. Interna	Bit de pulso desactivado	$\overline{B3:0/1}$	$\neg B301$
Var. Interna	Instrucción OSR	B3:0/0	OSR
Var. Interna	Instrucción OSR desactivada	$\overline{B3:00}$	$\neg OSR$
Var. Interna	Bit alternador activado	B3:0/2	B302
Var. Interna	Bit alternador desactivado	$\overline{B3:0/2}$	$\neg B302$
Var. de entrada	Sensor bajo-bajo activado SNBB	I:0/2	LL
Var. de entrada	Sensor bajo-bajo desactivado SNBB	$\overline{I:0/2}$	$\neg LL$
Var. de entrada	Sensor nivel bajo activado SNB	I:0/1	L
Var. de entrada	Sensor nivel bajo desactivado SNB	$\overline{I:0/1}$	$\neg L$
Var. de entrada	Sensor nivel alto-alto activado SNAA	I:0/0	H
Var. de entrada	Sensor nivel alto-alto desactivado SNAA	$\overline{I:0/0}$	$\neg H$
Var. de salida	Bomba 1 Activada	O:0/0	B1
Var. de salida	Bomba 1 Desactivada	$\overline{O:0/0}$	$\neg B1$
Var. de salida	Bomba 2 Activada	O:0/1	B2
Var. de salida	Bomba 2 Desactivada	$\overline{O:0/1}$	$\neg B2$

Tabla 3.2: Variables involucradas en el sistema de las bombas alternantes

las variables se encuentran desactivadas, el cual corresponde a un modo de operación correcto obtenemos 104 estados y 832 transiciones. Podemos observar que eligiendo un estado correcto hemos conseguido reducir el espacio de estados casi a la mitad (con respecto al modelo incremental con estados iniciales incorrectos y un 60 % al modelo ingenuo) y lo mismo sucede con las transiciones, este hecho se logra debido al algoritmo de construcción incremental y a la correcta elección de los estados iniciales, lo que nos permite únicamente tomar en consideración aquellos estados que son sucesores de los estados iniciales correctos. A pesar de esta reducción el modelo aún es grande.

Si utilizamos la información referente a las precedencias de variables y al estado de operación inicial correcto logramos el modelo mostramos en la figura 3.15 el cual tiene 32 estados y 82 transiciones, lo cual constituye una reducción del 87.5 % con respecto al modelo ingenuo en lo que respecta a los estados y un 95.99 % en lo que respecta a las transiciones. Los resultados obtenidos se detallan en la tabla 3.3.

Podemos observar que la reducción de estados es considerable aplicando las precedencias y el estado o estados iniciales correctos. Utilizar el conocimiento del sistema

ecuaciones
{
$B301 = LL \wedge OSR;$
$OSR = \neg LL;$
$B302 = (B302 \vee B301) \wedge (\neg B302 \vee \neg B301);$
$B1 = LL \wedge ((\neg B302 \wedge L) \vee B1 \vee (B2 \wedge (H \vee B1)));$
$B2 = LL \wedge ((B302 \wedge L) \vee B2 \vee (B1 \wedge (H \vee B2)));$
}

iniciales
{
$B301 = 0;$
$OSR = 0;$
$B302 = 0;$
$B1 = 0;$
$B2 = 0;$
$H = 0;$
$L = 0;$
}

precedencias
{
$LL = 1$ precfuerte $L = 1;$
$L = 1$ precfuerte $H = 1;$

$H = 0$ precfuerte $L = 0;$
$L = 0$ precfuerte $LL = 0;$
}

Figura 3.14: Ejemplo de algunas de las secciones que soporta el método de modelado.

para generar modelos más exactos es una buena forma de modelar.

3.4.5. Estados Marcados

Los estados marcados caracterizan aquellos estados que son importantes en el sistema. Por ejemplo, si nuestro sistema debe realizar cierta tarea, un estado marcado puede ser cuando ésta ya ha sido terminada.

En términos de ABGEs la colección de conjuntos de estados de aceptación estará formada por aquellos conjuntos de estados que cumplan con las condiciones de marcado. Se puede determinar un único estado marcado o un conjunto de estados que

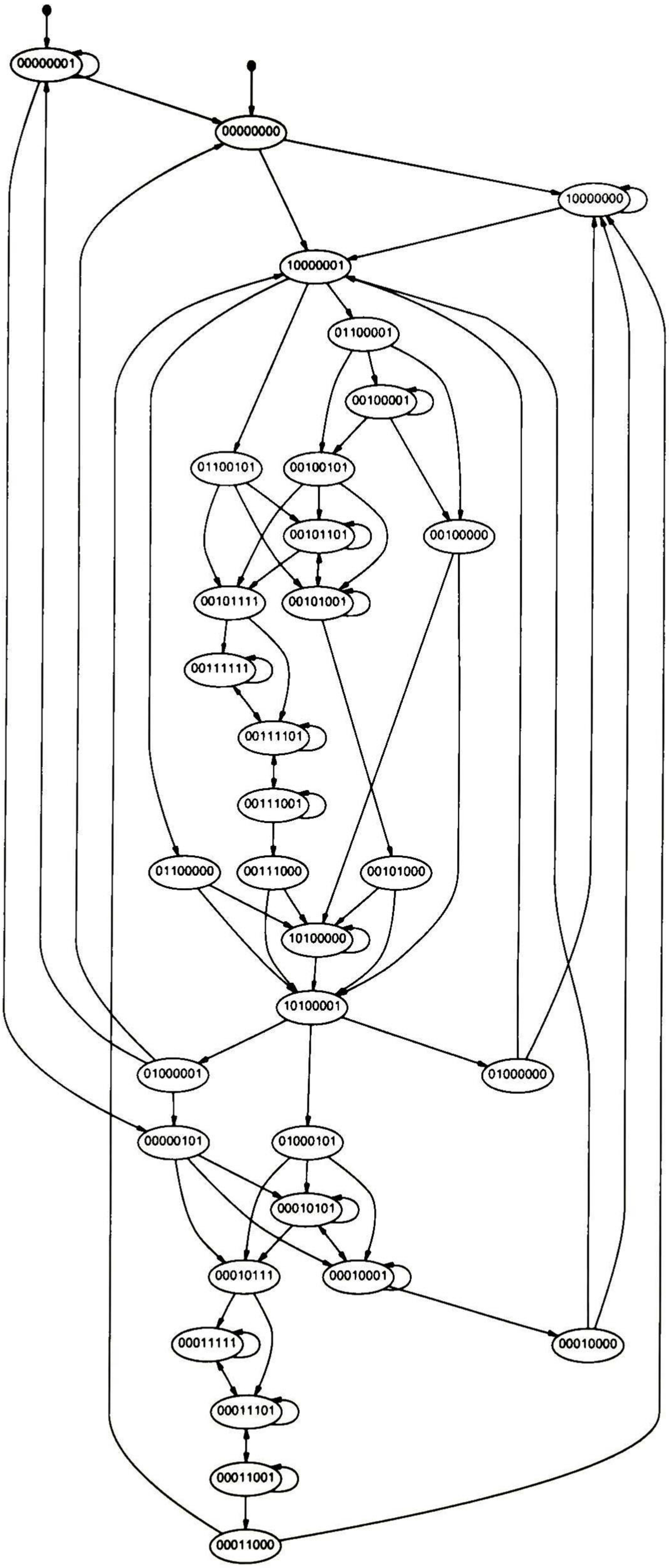


Figura 3.15: ABGE que representa el modelo generado por la herramienta desarrollada para el sistema de las bombas alternantes. El orden de variables es (OSR, B301, B302, B1, B2, L, H, LL)

Algoritmo	#Estados	#Transiciones	%Redu. edos.	%Redu. trans.
Ingenuo	256	2048	0	0
Incr. ini. incorrectos	200	1600	24	24
Incr. ini. correctos	104	832	59.3	59.3
Incr. ini. preced.	32	82	87.5	95.99

Tabla 3.3: Comparación entre los diversos modelos obtenidos con el ejemplo de las bombas alternantes pero en circunstancias diferentes.

cumplan ciertas condiciones de marcado, los cuales forman la colección de conjuntos de estados de marcados.

En los algoritmos de construcción justo después de agregar un nuevo estado al ABGE preguntamos si ese estado es marcado. Cada conjunto de estados marcados en la colección lo caracterizamos por medio de los valores de las variables que deseamos que tengan en dicho conjunto. Podemos decidir que un conjunto de estados marcados será cuando cada una de las variables tengan cierto valor y mediante esa configuración obtenemos un único estado marcado para ese conjunto. Podemos también decidir que un conjunto de estados marcados es aquel en el que un grupo de variables tienen un determinado valor y todos los estados que visitemos que cumplan con esa condición pertenecerán a ese conjunto de estados marcados. Si tomamos en cuenta todos los conjuntos de estados definidos como marcados obtenemos una colección de conjuntos de estados marcados. Podemos tener tantos conjuntos de estados marcados en la colección como sea necesario.

El algoritmo 3.16 muestra la forma en que se decide si un estado es marcado o no. Las entradas para éste algoritmo son: el estado actual *estado* la lista que contiene la colección de conjuntos de aceptación *laceptacion*. La salida es la decisión de marcar o no el *estado*.

Los estados marcados nos servirán en el proceso de verificación. Una de las condiciones para que un cómputo sea válido en nuestro modelo es que se visite un número infinito de veces por lo menos un estado de cada una de las colecciones de estados marcados.

La sección referente a los estados marcados se ejemplifica en la figura 3.17.

```
marca
1:  Entrada: estado, laceptacion
2:  Salida: se_marca = 1 o se_marca = 0
3:  marca = laceptacion
4:  Si marca es vacía
5:  Entonces
6:          Regresa se_marca = 0
7:  Si no
8:      conjunto_actual = marca- > NumeroConjunto
9:  Mientras marca no sea vacía
10:     Si conjunto_actual = marca- > NumeroConjunto
11:     Entonces
12:         Si marca- > Valor es diferente al valor de la variable correspondiente del estado
13:         Entonces
14:             nconjunto = marca- > NumeroConjunto
15:             se_marca = 0
16:             Mientras nconjunto = marca- > NumeroConjunto
17:                 Si marca- > Siguiente
18:                 Entonces
19:                     marca = marca- > Siguiente
20:                 Si no
21:                     nconjunto = -1
22:             Si nconjunto = -1
23:             Entonces
24:                 Regresa se_marca = 0
25:             Si no
26:                 conjunto_actual = marca- > NumeroConjunto
27:                 se_marca = 1
28:         Si no
29:             marca = marca- > Siguiente
30:     Si no
31:         Si se_marca = 1
32:         Entonces
33:             agregar estado conjunto_actual
34:             conjunto_actual = marca- > NumeroConunto
35:         Si no
36:             conjunto_actual = marca- > NumeroConunto
37:             se_marca = 1
38: Si se_marca = 1
39: Entonces
40:     agregar estado conjunto_actual
41: Regresa se_marca
```

Figura 3.16: Algoritmo para decidir si el estado actual es marcado.

```
marcados
{
    {X=1,V1=1, V2=0},
    {X=0,V1=0, V2=0}
{
```

Figura 3.17: Ejemplo de la sección de estados marcados.

En el primer ejemplo, detector de secuencia, se muestra la aplicación de los estados marcados. Cada vez que se llegue al estado 011 se sabe que se tiene una secuencia de aceptación.

3.4.6. Estados Prohibidos

La contraparte de los estados de aceptación son los estados prohibidos, en los cuales ocurren situaciones indeseables en el sistema. Por ejemplo, si tenemos un motor que hace avanzar una banda en dos direcciones y cada dirección del motor es activada por medio de una señal de un botón, entonces es deseable que únicamente una de las señales de dirección esté activada en un tiempo determinado.

Debido a que en ocasiones el diseño permite que se alcancen los estados prohibidos éstos se modelan inicialmente y mediante otras técnicas se eliminan (por ejemplo el método de síntesis mostrado en [49]). En la implementación se asegura que estos estados nunca se alcancen. Es deseable que el modelo también refleje el hecho de que los estados prohibidos no deben ser alcanzados.

Los estados prohibidos se representan de igual forma que los estados marcados: como una colección de conjuntos de estados prohibidos.

Podemos especificar un estado prohibido mediante la descripción de los valores que deben tener cada una de las variables en ese estado. Especificamos un conjunto de estados mediante la descripción de algunos de los valores de las variables y todos los estados donde esos valores se cumplen conformarán el conjunto de estados prohibidos. Si tomamos en cuenta varios conjuntos de estados prohibidos formamos una colección

de estados prohibidos.

En el algoritmo mostrado en la figura 3.18 se determina si un estado es prohibido. Las entradas de este algoritmo son el estado actual *estado* y la colección de estados prohibidos *lprohibidos*. La salida es la decisión de prohibir un estado o no.

La especificación de estados prohibidos nos permite acercar el modelo al funcionamiento real del sistema. La incorporación de estados prohibidos nos reduce el espacio de estados del modelo debido a que se modelan sólo los estados que son permitidos en el sistema. En la siguiente subsección mostraremos un ejemplo para ver la utilidad de modelar los estados prohibidos.

3.4.7. Ejemplo 3: Deshidratación de Gas Natural

El proceso de deshidratación utiliza torres empacadas para remover la humedad excesiva del gas natural.

En este proceso se utilizan sensores de presión para determinar la presión en las torres de deshidratación. Se utilizan válvulas para controlar el flujo de gas hacia las torres 1 y 2. El diagrama de la figura 3.19 muestra los componentes del proceso.

Descripción del proceso de control de deshidratación

Las dos torres son utilizadas para remover humedad del gas natural. Generalmente una de las torres está en *servicio* (por ejemplo removiendo humedad del gas de proceso) y la otra torre está siendo *secada* ó *regenerada*. Los pasos automáticos del proceso son como siguen:

1. Asuma que el sistema de control ha sido puesto en automático, así que el PLC puede controlar el proceso basado en las entradas de campo.

2. Si la presión de la torre 1 llega al valor alto, el controlador programable pondrá la torre 1 en modo de regeneración abriendo las válvulas FY-3 y FY-5 y cerrando FY-1 y FY-7.

3. Al mismo tiempo, si la presión en la torre 2 es baja, la torre 2 será puesta en servicio abriendo las válvulas de gas FY-2 y FY-8 y el sistema de control cerrará

```
prohibe
1:  Entrada: estado, lprohibidos
2:  Salida: se_prohibe = 1 o se_prohibe = 0
3:  prohibido = lprohibidos
4:  Si prohibido es vacío
5:  Entonces
6:      Regresa se_prohibe = 0
7:  Si no
8:      conjunto_actual = prohibido- > NumeroConjunto
9:  Mientras prohibido no sea vacia
10:     Si conjunto_actual = prohibido- > NumeroConjunto
11:     Entonces
12:         Si prohibido- > Valor es diferente al valor de la variable correspondiente del estado
13:         Entonces
14:             nconjunto = prohibido- > NumeroConjunto
15:             se_prohibe = 0
16:             Mientras nconjunto = prohibido- > NumeroConjunto
17:                 Si prohibido- > Siguiente
18:                 Entonces
19:                     prohibido = prohibido- > Siguiente
20:                 Si no
21:                     nconjunto = -1
22:             Si nconjunto = -1
23:             Entonces
24:                 Regresa se_prohibe = 0
25:             Si no
26:                 conjunto_actual = prohibe- > NumeroConjunto
27:                 se_prohibe = 1
28:         Si no
29:             prohibido = prohibido- > Siguiente
30:     Si no
31:         Si se_prohibe = 1
32:         Entonces
33:             Regresa se_prohibe = 1
34:         Si no
35:             conjunto_actual = prohibe- > NumeroConunto
36:             se_prohibe = 1
37: Si se_prohibe = 1
38: Entonces
39:     Regresa se_prohibe
```

Figura 3.18: Algoritmo para determinar si se prohibe el estado actual.

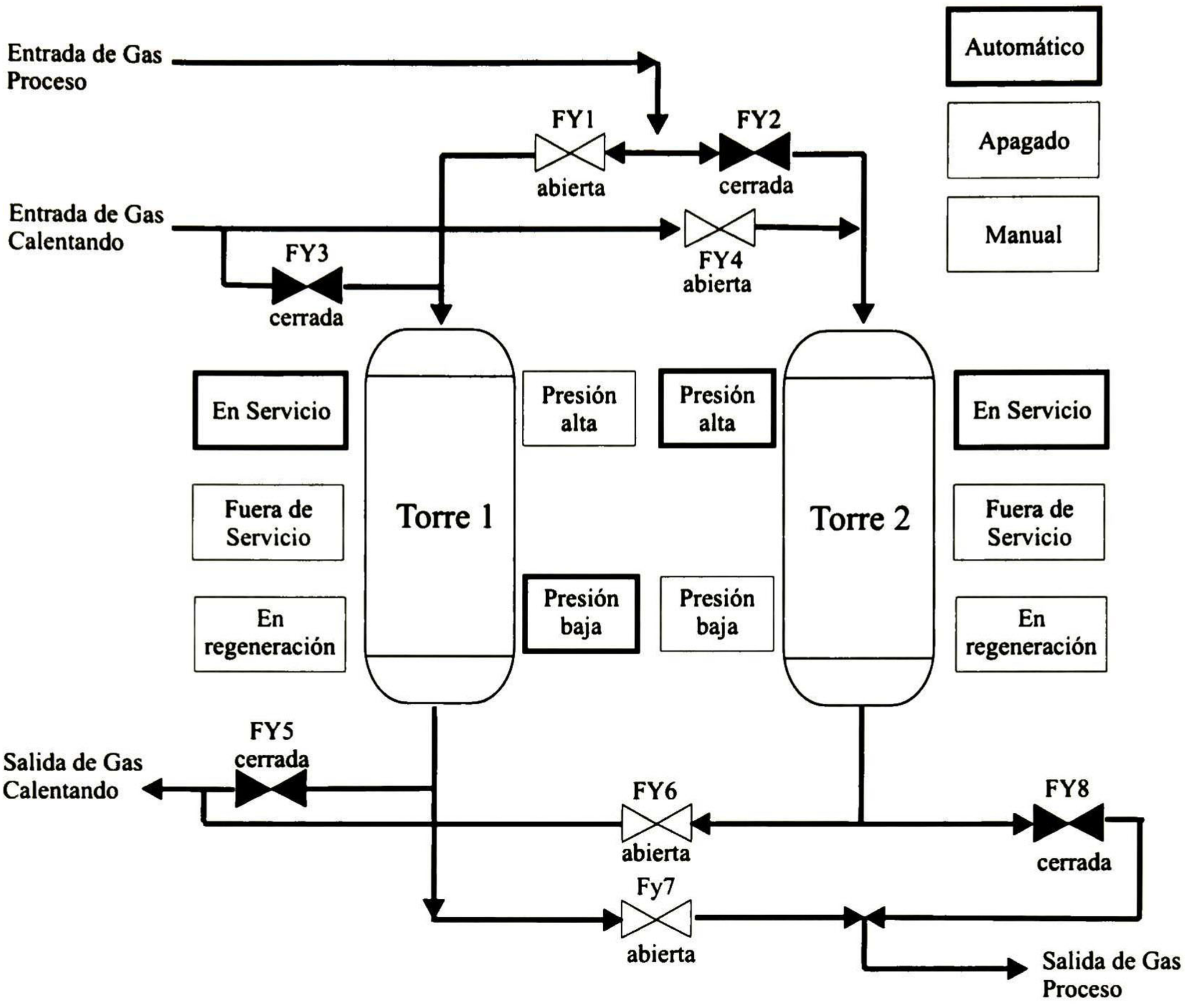


Figura 3.19: Diagrama que muestra los componentes del sistema para la deshidratación de gas natural.

las válvulas del ciclo de calentamiento FY-4 y FY-6 para la torre 2.

4. Despues, si la presión en la torre 2 llega al nivel alto y el sensor de presión de la torre 1 regresa a bajo, el sistema de control pondrá la torre 1 en servicio y la torre 2 en regeneración. Este ciclo continuará hasta que el operador elija parar el proceso.

La tabla 3.4 muestra las variables implicadas en la lógica de control.

El controlador utiliza interruptores de límite para determinar el estado o estados de las válvulas. En algunas aplicaciones, únicamente un interruptor de límite es

Componente	Descripción	Señal	Nombre Etiqueta
Var. interna	Modo Automático	B3:0/00	B300
Var. interna	Desactivado	B3:0/01	B301
Var. interna	Modo Manual	B3:0/02	B302
Var. interna	T1 en serv.	B3:0/03	B303
Var. interna	T1 fuera de serv.	B3:0/04	B304
Var. interna	T1 en reg.	B3:0/05	B305
Var. interna	T2 en servicio	B3:0/06	B306
Var. interna	T2 fuera de serv.	B3:0/07	B307
Var. interna	T2 en reg.	B3:0/08	B308
Var. de entrada	Presión baja T1	I:000/00	I00
Var. de entrada	Presión alta T1	I:000/01	I01
Var. de entrada	Presión baja T2	I:000/02	I02
Var. de entrada	Presión alta T2	I:000/03	I03
Var. de salida	Válvula FY-1	O:001/10	O10
Var. de salida	Válvula FY-2	O:001/11	O11
Var. de salida	Válvula FY-3	O:001/12	O12
Var. de salida	Válvula FY-4	O:001/13	O13
Var. de salida	Válvula FY-5	O:001/14	O14
Var. de salida	Válvula FY-6	O:001/15	O15
Var. de salida	Válvula FY-7	O:001/16	O16
Var. de salida	Válvula FY-8	O:001/17	O17

Tabla 3.4: Componentes del sistema de deshidratación de gas natural.

utilizado y el controlador puede asumir el estado opuesto (cerrado o abierto) si un interruptor sencillo es utilizado. Sin embargo, un control más confiable se obtiene si dos interruptores son usados. Por ejemplo, si una válvula falla al abrirse o cerrarse completamente en alguna operación, el controlador es capaz de detectar esta falla y señalarla al operador.

La lógica de escalera para el modo automático del proceso puede ser escrita como se muestra en la figura 3.20. La torre 1 es puesta en servicio si la presión en la torre 1 es baja, la función de operación es automática y la presión no es alta. La torre 1 permanecerá en servicio hasta que el sensor de presión llega al valor alto.

Si el sensor de presión alta se activa entonces el bit de salida I:000/00 es puesto en 1 y su contacto normalmente cerrado es abierto. Así que la bobina de salida de la

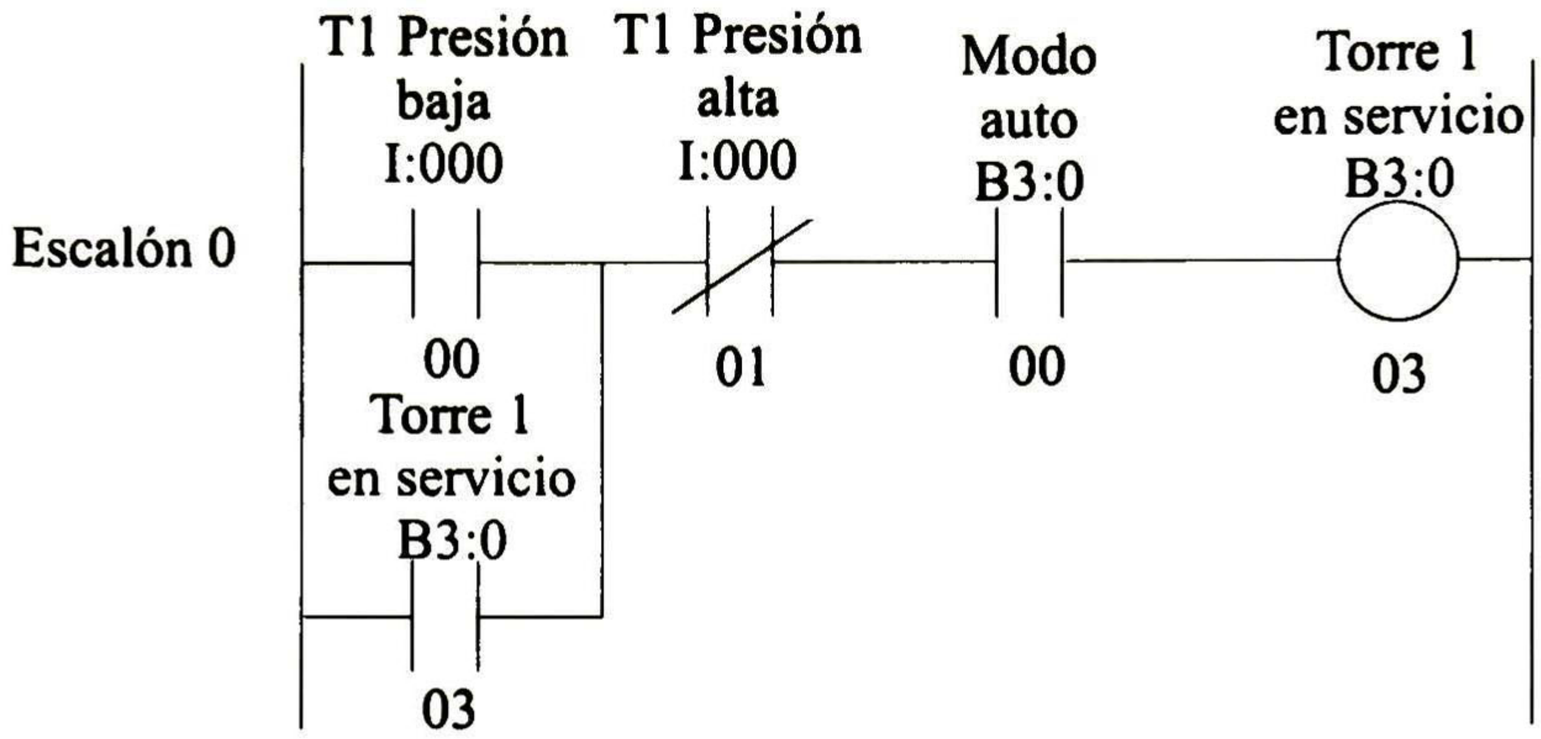


Figura 3.20: Escalón 0 del proceso de deshidratación de gas natural.

torre 1 en servicio es desactivada.

Utilizando la lógica de escalera de la figura 3.21 la torre 1 es puesta en regeneración por el sistema de control para remover la humedad que se ha acumulado durante el ciclo de servicio.

El programa en lógica de escalera de la figura 3.22 muestra el control para las válvulas FY-1, FY-7, FY-3 y FY-5. El mismo procedimiento de diseño puede ser utilizado para escribir la lógica de control para la torre 2.

El archivo de entrada para la herramienta se genera de la misma manera que en el ejemplo de las bombas alternantes. Ahora nos vamos a aprovechar del modo de operación para integrar ese conocimiento del funcionamiento del sistema. Lo haremos de forma tal que algunos estados serán prohibidos.

El archivo de entrada para la herramienta de modelado y la sección de los estados prohibidos se muestra en la figura 3.23 y se justifica como sigue. El modo de operación se encuentra en automático, por lo tanto consideraremos que es un estado prohibido cuando el modo de operación automático esté desactivado. Ambas torres sabemos que se encuentran en funcionamiento, ya sea en servicio o en regeneración, es por eso que está prohibido que se encuentren en el modo fuera de servicio. Los indicadores de presión son independientes entre sí, pero se activan de manera opuesta, ya que si se indica que la presión es alta no puede suceder que al mismo tiempo se indique que

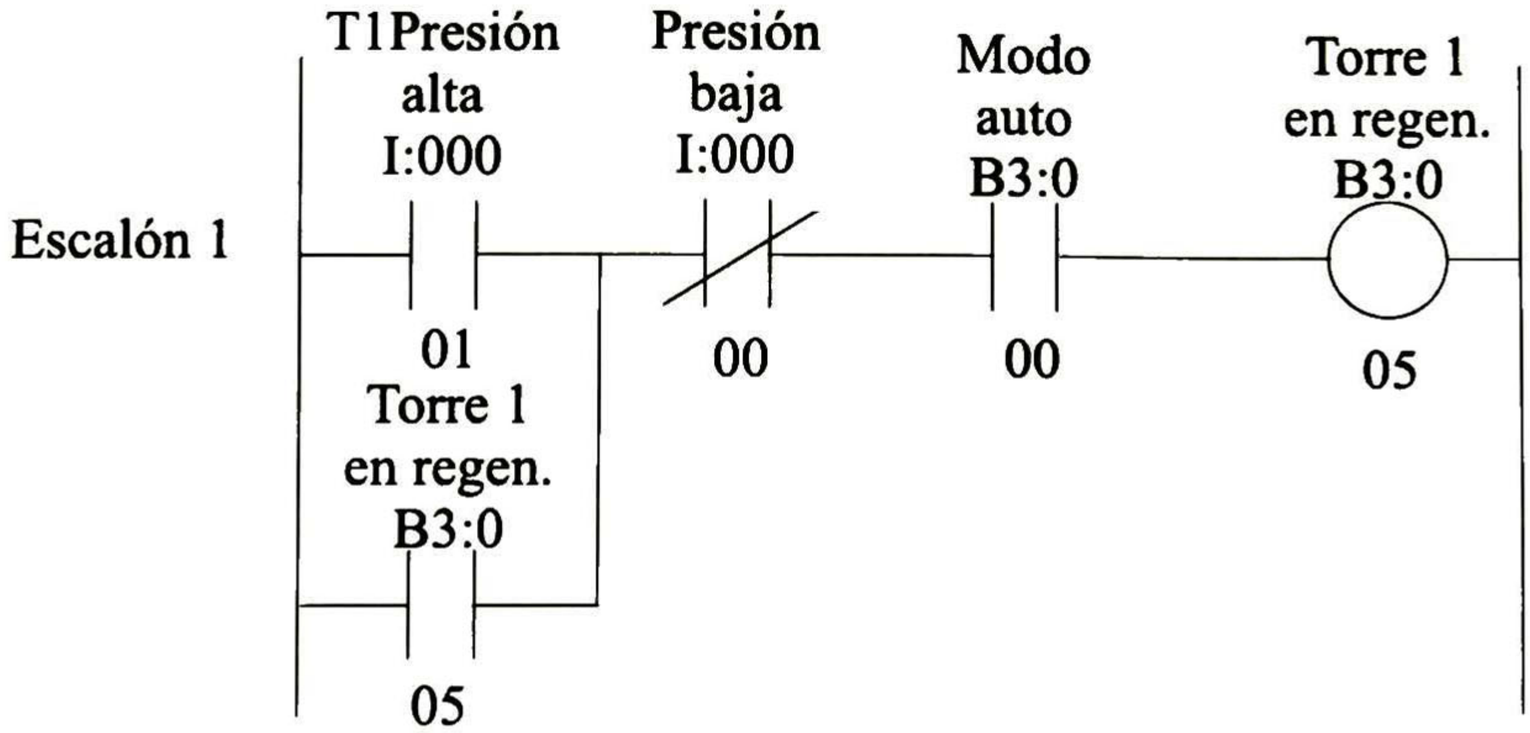


Figura 3.21: Escalón 1 del proceso de deshidratación de gas natural.

la presión es baja y viceversa. Este comportamiento ocurre para los indicadores de presión de las dos torres.

Se realizó el mismo análisis comparativo que el que se hizo con el sistema de las bombas alternantes, y los resultados se muestran en la tabla 3.5. La construcción in-

Algoritmo	#Edos.	#Transiciones	%Reduc. edos.	%Reduc. trans.
Ingenuo	2^{19}	$2^{19} * 7$	0	0
Inc. ini. incorrectos	>5000	>600000	99	83
Inc. ini. correctos	3456	442368	99.99	87.94
Inc. ini. y edos. proh.	16	64	99.99	99.99

Tabla 3.5: Comparación entre los diversos modelos obtenidos con el ejemplo del sistema de deshidratación de gas natural en circunstancias diferentes.

genua es muy ineficiente en este caso, incluso el modelo con estados iniciales correctos es mucho mayor que el modelo que contiene los estados prohibidos. El autómata que corresponde al modelo con estados prohibidos se muestra en la figura 3.24.

Los ejemplos de las bombas alternantes y del sistema de deshidrataciòn de gas natural fueron tomados de [28].

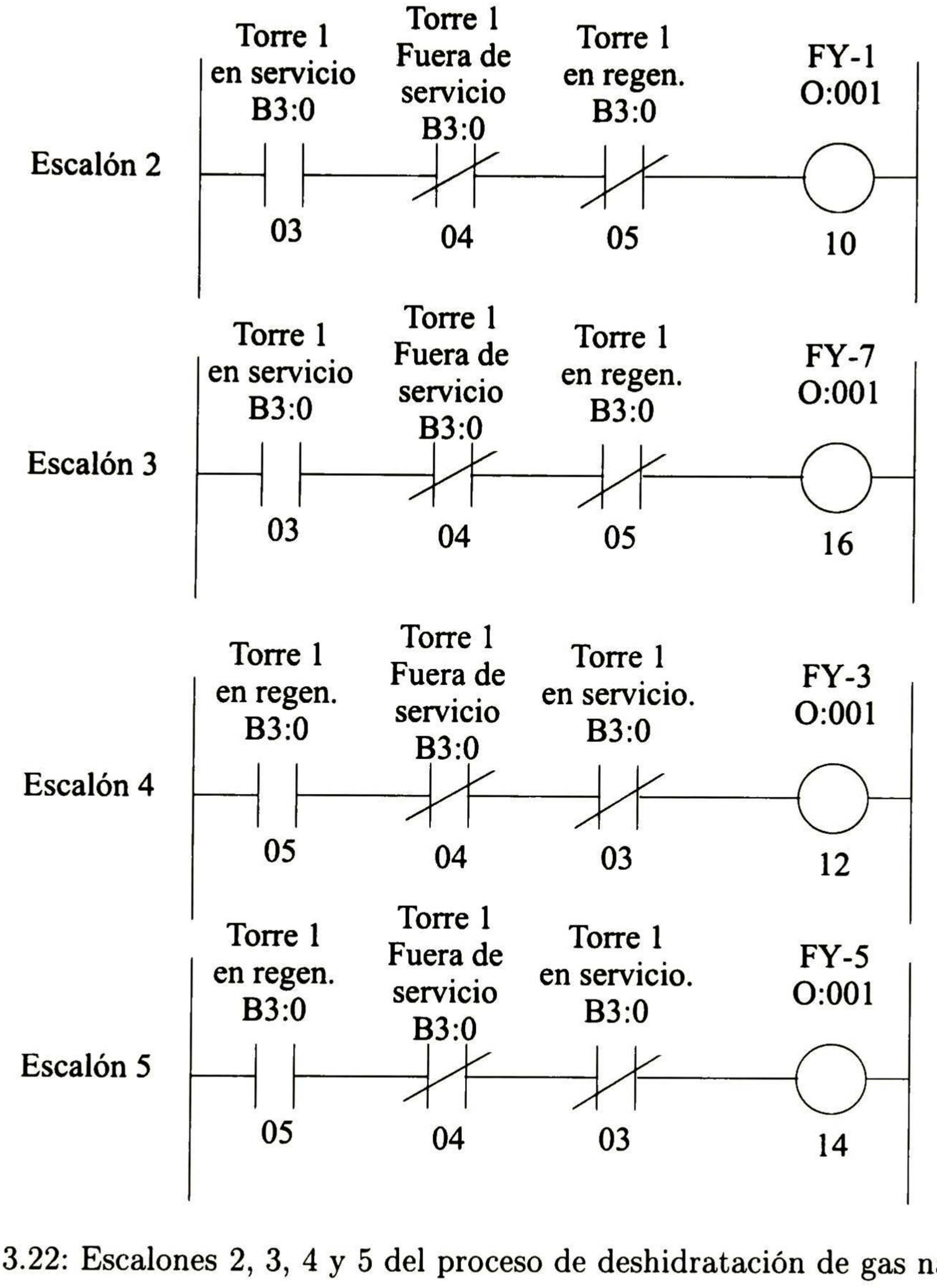


Figura 3.22: Escalones 2, 3, 4 y 5 del proceso de deshidratación de gas natural.

3.5. Conclusiones

Mediante la utilización de la información referente al funcionamiento de los sistemas podemos obtener modelos más fieles al funcionamiento de los mismos. Aplicar la técnica de prohibición de estados nos puede dar resultados buenos al reducir el comportamiento del sistema que sabemos nunca sucederá. Aunque se aplique el conocimiento del sistema los algoritmos siguen siendo exponenciales en el peor de los casos y eso ocasiona que muchas veces la verificación mediante cálculo explícito no sea muy adecuada. Aplicar el conocimiento del sistema es una táctica buena para generar modelos apegados al funcionamiento del sistema. Pero en ocasiones los sistemas no tienen la interrelación de variables necesaria para aplicar una precedencia o no podemos simplemente prohibir estados. y los sistemas que se generan son demasiado grandes para verificarlos directamente por medio de cálculo explícito. Es por eso que se deben introducir algunas técnicas más eficientes para reducir el espacio de estados. tales como abstracciones, simulaciones, bisimulaciones, etc.

En el siguiente capítulo se muestran algunas técnicas que se han venido utilizando y se detalla la técnica de abstracción que se implementó en la herramienta de modelado.

Ecuaciones
{
$B303 = (I00 \vee B303) \wedge I01 \wedge B300$;//Torre1 en servivio
$B305 = (I01 \vee B305) \wedge I00 \wedge B300$;//Torre1 en regeneración
$O10 = B303 \vee B304 \wedge B305$;//válvula FY1
$O16 = B303 \vee B304 \wedge B305$;//válvula FY7
$O12 = B305 \wedge B304 \wedge B303$;//válvula FY3
$O14 = B305 \wedge B304 \wedge B303$;//válvula FY5
$B306 = (I02 \vee B306) \wedge I03 \wedge B300$;//Torre2 en servicio
$B308 = (I03 \vee B308) \wedge I02 \wedge B300$;//Torre2 en regeneración
$O11 = B306 \wedge B304 \wedge B308$;//válvula FY2
$O17 = B306 \wedge B307 \wedge B308$;//válvula FY8
$O13 = B308 \wedge B307 \wedge B306$;//válvula FY4
$O15 = B308 \wedge B307 \wedge B306$;//válvula FY6
}

iniciales
{
B300=1; //Modo de operacion automático
B303=1; //Torre 1 en servicio
B304=0; //Torre 1 ¬fuera de servicio
B305=0; //Torre 1 en ¬regeneración
I00=1; //indicador presión Torre1 bajo
I01=0; //indicador presión Torre1 ¬alto
O10=1; //válvula FY1 abierta
O12=0; //válvula FY3 cerrada
O14=0; //válvula FY5 cerrada
O16=1; //válvula FY7 abierta
B306=0; //Torre 2 en ¬servicio
B307=0; //Torre 2 en ¬fuera de servicio
B308=1; //Torre 2 en regeneración
I02=0; //indicador presión Torre2 ¬bajo
I03=1; //indicador presión Torre2 alto
O11=0; //válvula FY2 cerrada
O13=1; //válvula FY4 abierta
O15=1; //válvula FY6 abierta
O17=0; //válvula FY8 cerrada
}

prohibidos
{
{B300=0}, //modo automático solamente permitido
{B304=1}, //la torre 1 debe estar funcionando
{B307=1}, //la torre 2 debe estar funcionando
{I00=1, I01=1},//no puede estar la presion baja y alta al mismo tiempo
{I00=0, I01=0},
{I02=1, I03=1},
{I02=0, I03=0}
}

Figura 3.23: Archivo para el sistema de deshidratación de gas natural.

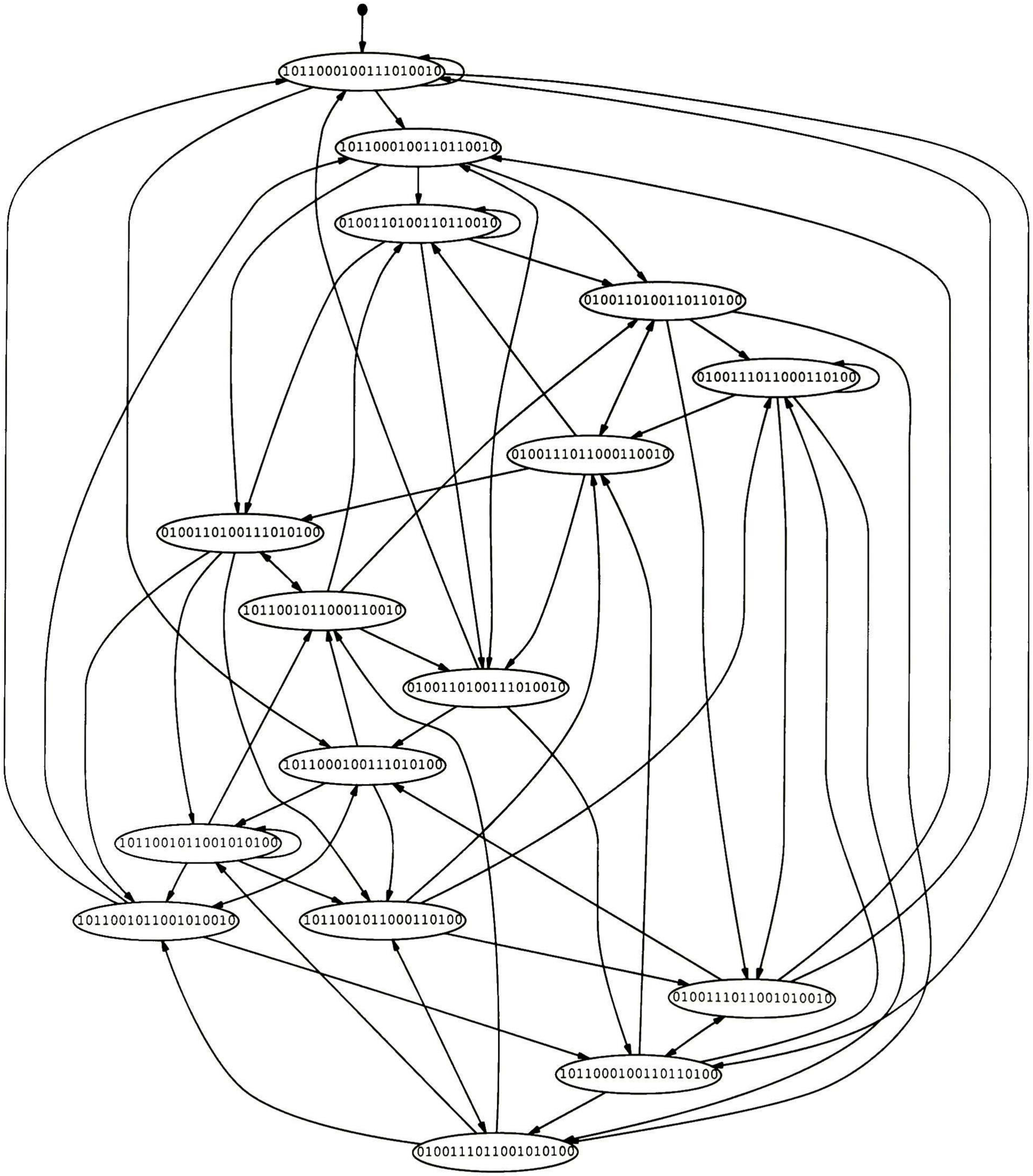


Figura 3.24: ABGE que representa el modelo generado por la herramienta desarrollada para el sistema de deshidratación de gas natural. El orden de variables es el siguiente (B303, B305, O10, O16, O12, O14, B306, B308, O11, O17, O13, O15, I00, I01, B300, B304, I02, I03, B307)

Capítulo 4

Técnicas de Reducción del Espacio de Estados

4.1. Contenido del Capítulo

En este capítulo se hace un breve recuento de las técnicas más comúnmente utilizadas para tratar con el problema de explosión de estados y se muestra la técnica implementada en la herramienta de construcción de modelos, la cual consiste en una simulación por la proyección del conjunto de las variables que aparecen en la propiedad a verificar. En la sección 4.2 discutimos porqué surge el problema de explosión de estados en el modelado. En la sección 4.3 hacemos una breve recapitulación de las técnicas más comunes utilizadas para tratar con el problema de explosión de estados. La técnica de simulación por proyección con respecto a un conjunto de variables la detallamos en la sección 4.4. En la sección 4.5 explicamos la forma de calcular las proyecciones de una manera eficiente. En la sección 4.6 mostramos los cambios necesarios efectuados a los algoritmos de construcción mostrados en el capítulo 3, así como a los algoritmos para definir las precedencias, los estados marcados y los estados prohibidos. En la sección 4.7 ejemplificamos la técnica seleccionada. En la sección 6 hacemos una reflexión sobre la utilidad de emplear técnicas de reducción y los problemas encontrados al aplicarlas.

4.2. Problema de Explosión de Estados

El método de verificación por comprobación de modelos necesita para su aplicación dos modelos: un modelo del sistema y un modelo de la propiedad a probar, ambos expresados en el mismo formalismo. Si utilizamos la verificación por comprobación de modelos mediante cálculo explícito, el modelo es una estructura de estados y transiciones, en donde cada estado del modelo representa un momento en la operación del sistema donde suceden acciones específicas. Utilizando el modelo explícito del sistema podemos seguir la secuencia de operaciones realizadas en el sistema. Por ejemplo, si tenemos una válvula solenoide cuyas acciones son abrirse y cerrarse, un modelo para esa válvula se representaría con una estructura de dos estados, uno de los cuales es el estado donde la válvula se encuentra cerrada y otro el estado donde se encuentra abierta. Podemos seguir la secuencia de operación fácilmente, del estado válvula cerrada al estado válvula abierta y del estado válvula abierta al estado válvula cerrada. Si agregamos otra válvula, el comportamiento que podemos encontrar ahora está representado por 4 estados, dada por la combinación de válvulas abiertas y cerradas. Si continuamos agregando válvulas (u otros componentes), el comportamiento se duplica por cada válvula agregada. Éste es un crecimiento exponencial y fácilmente obtenemos un espacio de estados muy grande aún con pocos componentes en el sistema.

Cuando los modelos son creados a partir de una descripción dada en lenguaje de escalera es probable que no se genere todo el espacio posible de estados por la manera en que se construye el modelo. Pero el sistema tiene un espacio de estados potencial de 2^n donde n es el número de variables booleanas. Entonces un sistema con tan sólo 20 variables boolenas tiene un espacio potencial de $1'048,576$ estados y un número mucho mayor de transiciones. Guardar ese autómata en memoria es muy caro y realizar operaciones sobre él es costoso en tiempo (si acaso la operación puede realizarse).

4.3. Técnicas de Reducción del Espacio de Estados

Las diversas técnicas para reducir el espacio de estados se incorporan con el fin de obtener modelos que sean manejables y que el modelo reducido del sistema no pierda sus propiedades originales. A continuación se describen brevemente algunas de las técnicas comunes aplicadas para reducir el espacio de estados.

- **Reducción de Orden Parcial**

  El método consiste en la construcción de un grafo de estados reducido. El grafo completo nunca es construido. Los comportamientos del grafo reducido forman un subconjunto del de los comportamientos del grafo de estados completo. La justificación del método de reducción muestra que los comportamientos que no están presentes no añaden información. Es posible definir una relación de equivalencia entre los comportamientos tales que la propiedad a ser verificada no sea pueda distinguir entre comportamientos equivalentes. Si un comportamiento no está presente en el grafo de estados reducido, se debe incluir un comportamiento equivalente [16, 27, 42].

  La reducción de orden parcial está basada en la dependencia que existe entre las transiciones de un sistema. Éste método de reducción especifica cuáles transiciones deben ser incluidas en el modelo reducido y cuáles no.

- **Reducción al Cono de Influencia**

  La reducción al cono de influencia intenta reducir el tamaño del grafo de estados y transiciones enfocándose en las variables del sistema que ocurren en la especificación. La reducción se obtiene eliminando variables que no influyen sobre las variables en la especificación. De esta manera las propiedades verificadas son preservadas, pero el tamaño del modelo que necesita ser verificado es más pequeño [16].

- **Reducción por Explotación de Simetrías**

  La técnica de reducción por explotación de simetrías se basa en la observación

de que muchos sistemas consisten de varios componentes que exhiben considerable simetría en su estructura. Por ejemplo, es posible encontrar simetría en memorias, caches, protocolos, etc.

Las técnicas de reducción por simetría están basadas en el hecho de que tener simetría en el sistema implica la existencia de grupos de permutaciones no triviales que preservan la relación de transición y la función de etiquetado. Tales grupos pueden ser usados para definir una relación de equivalencia sobre el espacio de estados del sistema. El modelo cociente del sistema inducido por esta relación suele ser más pequeño que el modelo original. Además es equivalente por bisimulación al modelo original [29, 21, 16].

- **Reducción por Bisimulación**

  Los algoritmos de minimización por bisimulación particionan un espacio de estados en clases de equivalencia tal que los estados en la misma clase son observacionalmente equivalentes con respecto al comportamiento del sistema.

  Los estados en cada clase de equivalencia bajo la bisimulación coinciden en los valores de las proposiciones atómicas y sobre sus transiciones de estado siguiente a otras clases [9, 16, 20, 22, 34].

  Si una especificación es válida en el modelo reducido, es así mismo verdadera en el modelo concreto, y si una especificación es falsa en el modelo reducido esa misma especificación es falsa en el modelo concreto.

- **Reducción por Simulación**

  La simulación garantiza que cada comportamiento de una estructura es también un comportamiento de su abstracción. Sin embargo, la abstracción podría tener comportamientos que no son posibles en la estructura original.

  Cuando una especificación es verdadera en el modelo abstracto, será verdadera en el modelo concreto.

  Si la especificación es falsa en el modelo abstracto, el contraejemplo puede ser el resultado de algún comportamiento en la aproximación el cual no está presente en el modelo original [9, 16, 34].

4.4. Simulación Por Abstracción de Variables

La técnica de reducción del espacio de estados que seleccionamos para incorporar en la herramienta de modelado desarrollada, es una técnica de simulación, dada por una función de abstracción sobre el conjunto de las variables del sistema que se encuentran en la propiedad a probar.

En los sucesivo consideraremos únicamente ABGEs definidos sobre subconjuntos de Σ de la forma $\Sigma' := 2^{AP'}$ donde AP' es alguno de los subconjuntos del conjunto de proposiciones atómicas AP

Sean $\mathcal{A}' = (Q',\ Q'_0,\ \rho',\ \mathcal{F}',\ l')$ y $\mathcal{A}'' = (Q'',\ Q''_0,\ \rho'',\ \mathcal{F}'',\ l'')$ ABGEs sobre $\Sigma' := 2^{AP'}$ y $\Sigma'' := 2^{AP''}$, respectivamente. Sea $H \subseteq Q' \times Q''$. Decimos que H es una *relación de simulación* de $\mathcal{A}'$ en $\mathcal{A}''$ si y sólo si para todos los estados $q' \in Q'$ y $q'' \in Q''$, si $(q',\ q'') \in H$, entonces

1. $l''(q'') = \{C \cap AP'' \mid C \in l'(q')\}$

2. Para cada corrida legal $\pi' = q'_0,\ q'_1,\ ,q'_2, \ldots$ en $\mathcal{A}'$ que parte de q' $(q' = q'_0)$, existe una corrida legal $\pi'' = q''_0,\ q''_1,\ ,q''_2, \ldots$ en $\mathcal{A}''$ que parte de $q'' = q''_0$ tal que $(q'_i, q''_i) \in H$, para cada $i \geq 0$. En este caso decimos que las *corridas legales* π' y π'' se corresponden.

Decimos que $\mathcal{A}''$ *simula* a $\mathcal{A}'$ (o que $\mathcal{A}''$ es *una abstracción* de $\mathcal{A}'$) y escribimos $\mathcal{A}' \preceq \mathcal{A}''$, si $AP'' \subseteq AP'$ y existe una relación de simulación H de $\mathcal{A}'$ en $\mathcal{A}''$ tal que para cada $q'_0 \in Q'$ hay al menos un $q''_0 \in Q''$ con $(q'_0,\ q''_0) \in H$.

Lema 4.4.1. *La relación $\preceq$ es un preorden sobre la colección de todos los ABGEs definidos sobre los subconjuntos de Σ de la forma $2^{AP'}$*

Prueba:

($\preceq$ es reflexiva) Sea $\mathcal{A} = (Q,\ Q_0,\ \rho,\ \mathcal{F},\ l)$ un ABGE sobre un conjunto $\Sigma := 2^{AP'}$ Entonces, la identidad $I := \{(q,\ q) \mid q \in Q\}$ es una relación de simulación de $\mathcal{A}$ en $\mathcal{A}$, por lo cual $\mathcal{A} \preceq \mathcal{A}$.

($\preceq$ es transitiva) Sean $\mathcal{A}'$, $\mathcal{A}''$ y $\mathcal{A}'''$ ABGEs sobre tres conjuntos Σ', Σ'' y Σ''', respectivamente, tales que $\Sigma''' = 2^{AP'''}$, $\Sigma'' = 2^{AP''}$ y $\Sigma' = 2^{AP'}$ Supongamos que

$\mathcal{A}' \preceq \mathcal{A}''$ mediante H_1 y $\mathcal{A} \preceq \mathcal{A}'''$ mediante H_2. Sea $H_3 := H_1 \circ H_2$, la composición de H_1 y H_2. Luego, $H_3 \subseteq Q' \times Q'''$ Supongamos ahora que $(q',\ q''') \in H_3$. Entonces existe $q'' \in Q$ tal que $(q',\ q'') \in H_1$ y $(q'',\ q''') \in H_2$. Esto implica que

- $l''(q'') = \{C \cap AP'' \mid C \in l'(q')\}$ y $l'''(q''') = \{D \cap AP''' \mid D \in l''(q'')\}$ de donde se obtiene fácilmente que $l'''(q''') = \{C \cap AP''' \mid C \in l'(q')\}$, usando el hecho $AP''' \subseteq AP''$; además,
- si $\pi' = q'_0,\ q'_1,\ q'_2, \ldots$ es una corrida legal en $\mathcal{A}'$ que parte de q', entonces existe una corrida legal $\pi'' = q''_0,\ q''_1,\ q''_2, \ldots$ en $\mathcal{A}''$ que parte de q'' tal que $(q'_i,\ q''_i) \in H_1$, para cada $i \geq 0$, y también una corrida legal $\pi''' = q'''_0,\ q'''_1,\ q'''_2, \ldots$ en $\mathcal{A}'''$ que parte de q''' tal que $(q''_i,\ q'''_i) \in H_2$, para cada $i \geq 0$. Entonces $(q'_i,\ q'''_i) \in H_3$, para cada $i \geq 0$, por definición de H_3.

Por lo tanto, $\mathcal{A}' \preceq \mathcal{A}'''$ □

Sean $\mathcal{A}' = (Q',\ Q'_0,\ \rho'.\ \mathcal{F}',\ l')$ y $\mathcal{A}'' = (Q'',\ Q''_0,\ \rho'',\ \mathcal{F}'',\ l'')$ ABGEs sobre $\Sigma' := 2^{AP''}$ y $\Sigma'' := 2^{AP''}$, respectivamente, tales que $\mathcal{A}' \preceq \mathcal{A}''$ mediante la relación de simulación H. Sea $\sigma' = a'_0 a'_1 a'_2 \ldots$ una ω-palabra aceptada por $\mathcal{A}''$. Entonces se cumple el lema siguiente.

Lema 4.4.2. *Existe una ω-palabra σ'' aceptada por $\mathcal{A}''$ tal que para toda formula φ de LTLP con $Voc(\varphi) \subseteq AP''$ se cumple la propiedad:*

$$\textit{Si hay un } i \in \mathbb{N}_0 \textit{ tal que } \sigma'',\ i \models \varphi,\ \textit{entonces } \sigma',\ i \models \varphi$$

Prueba:

Como $\sigma = a'_0 a'_1, a'_2, \ldots$ es una ω-palabra aceptada por $\mathcal{A}'$, existe una corrida de aceptación de $\mathcal{A}'$, digamos $\pi' = q'_0,\ q'_1,\ q'_2, \ldots$, donde q'_0 es un estado inicial y $a_i \in l'(q'_i)$, para cada $i \geq 0$.

Ahora, como $q'_0 \in Q'_0$ y H es una relación de simulación de $\mathcal{A}'$ en $\mathcal{A}''$, existe un estado inicial $q''_0 \in Q''_0$ tal que $(q'_0,\ q''_0) \in H$. Esto implica que existe una corrida de aceptación $\pi'' = q''_0,\ q''_1,\ q''_2, \ldots$ en $\mathcal{A}''$ que parte de q''_0 tal que π' y π'' se corresponden.

Sea $\sigma'' = a''_0 a''_1 a''_2 \ldots$ con $a''_i := a'_i \cap AP''$ para cada $i \geq 0$. Entonces $a''_i \in l''(q''_i)$, para cada $i \geq 0$. Luego, σ'' es una ω-palabra asociada a π'' y por tanto es aceptada por $\mathcal{A}''$.

Apliquemos inducción sobre la estructura de la fórmula φ para probar la propiedad enunciada en este lema.

(Base) Sea $\varphi = \text{p}$ una fórmula atómica en AP y supongamos que σ'', $i \models \text{p}$ para $i \geq 0$. Entonces $\text{p} \in a_i''$. Pero $a_i'' \subseteq a_i'$, con lo cual $\text{p} \in a_i'$ y por tanto σ', $i \models \text{p}$.

Sea $\varphi = \neg\, \text{p}$ y supongamos que σ'', $i \models \neg\, \text{p}$, para un $i \geq 0$. Entonces, σ'', $i \not\models \text{p}$ con lo cual $\text{p} \notin a_i'$. Pero $\text{p} \in AP''$ (ya que $\{\text{p}\} = Voc(\varphi) \subseteq AP''$), por lo cual $\text{p} \notin a_i'$. Consecuentemente, σ', $i \not\models \text{p}$, lo que equivale a σ', $i \models \neg\, \text{p}$.

(Inducción) Supongamos que se cumple la propiedad enunciada en este lema para las subfórmulas de una fórmula φ dada.

Sea $\varphi = \neg\neg\psi$. Supongamos que σ'', $i \models \neg\neg\psi$ para un $i \geq 0$. Como $\neg\neg\psi \equiv \psi$, tenemos que σ'', $i \models \psi$. Por la hipótesis de inducción esto implica que σ', $i \models \psi$, lo que es equivalente a σ', $i \models \neg\neg\psi$

Sea $\varphi = \psi_1 \vee \psi_2$ y supongamos que σ'', $i \models \psi_1 \vee \psi_2$ para un $i \geq 0$. Entonces, σ'', $i \models \psi_1$ ó σ'', $i \models \psi_2$. Por la hipótesis de inducción esto implica que σ', $i \models \psi_1$ ó σ', $i \models \psi_2$, por lo que σ', $i \models \psi_1 \vee \psi_2$.

La prueba es muy semejante a la anterior para el caso en que $\varphi = \psi_1 \wedge \psi_2$.

Sea $\varphi = \bigcirc\psi$ y supongamos que σ'', $i \models \bigcirc\psi$ para un $i \geq 0$. Entonces σ'', $i+1 \models \psi$, de donde σ', $i+1 \models \psi$, por la hipótesis de inducción. Por lo tanto σ', $i \models \bigcirc\psi$.

Sea $\varphi = \psi_1 \mathcal{U} \psi_2$ y supongamos que σ'', $i \models \psi_1 \mathcal{U} \psi_2$ para un $i \geq 0$. Entonces, existe $k \geq i$ tal que σ'', $k \models \psi_2$ y para todo j con $i \leq j < k$ se cumple que σ'', $j \models \psi_1$.

Por la hipótesis de inducción tenemos que σ', $k \models \psi_2$ y σ', $j \models \psi_1$ para cada j con $0 \leq j < k$, de las que obtenemos σ', $i \models \psi_1 \mathcal{U} \psi_2$.

Para el caso $\varphi = \psi_1 \mathcal{V} \psi_2$ la prueba es muy semejante a la anterior.

Por lo tanto, para toda fórmula φ de la LTLP, si hay un $i \geq 0$ tal que σ'', $i \models \varphi$, entonces σ', $i \models \varphi$, por el principio de inducción estructural. □

Como consecuencia del lema 4.4.2, tenemos que bajo el preorden de simulación se preserva (débilmente) el cumplimiento de propiedades expresadas como fórmulas de la LTLP en ABGEs relacionados por el preorden de simulación.

Corolario 4.4.3. *Sean $\mathcal{A}'$ y $\mathcal{A}''$ ABGEs sobre Σ' y Σ'', respectivamente, tales que $\mathcal{A}' \preceq \mathcal{A}''$ mediante una relación de simulación H. Sea φ una fórmula de la LTLP tal que $Voc(\varphi) \subseteq AP'$*

Si $\mathcal{A}' \models \varphi$, *entonces* $A \models \varphi$.

Prueba:

Supongamos que $\mathcal{A}' \models \varphi$. Sea σ una ω-palabra aceptada por $\mathcal{A}$. Entonces, por el lema 4.4.2, existe una ω-palabra σ' aceptada por $\mathcal{A}'$ tal que si σ', $i \models \varphi$, para algún $i \geq 0$, entonces σ, $i \models \varphi$. Pero como $\mathcal{A}' \models \varphi$, tenemos, en particular, que σ', $0 \models \varphi$. Luego, σ, $0 \models \varphi$ para cualquier ω-palabra σ aceptada por A. Por lo tanto, $A \models \varphi$ □

Sea $AP' \subseteq AP$ Supongamos que $AP' = \{p_1, p_2, \ldots, p_k\}$ para algún $k \leq n$. Sea $q = (v_1, v_2, \ldots, v_n)$ un estado de $\mathcal{A}$.

Definición 4.4.4. Proyección de q con Respecto a AP'

1. La *proyección de q con respecto a* AP' es la k-tupla ordenada

$$proy(q, AP') := (v_1, v_2, \ldots, v_k)$$

Extendiendo esta definición a conjuntos de estados, si $S \subseteq Q$, entonces la proyección del conjunto S con respecto a AP' es el conjunto

$$proy(S, AP') := \{proy(q, AP') \mid q \in S)$$

Vista como una asignación, la proyección de q con respecto al conjunto AP' es simplemente la restricción de q a AP'; esto es, $proy(q, AP) = q \mid_{AP'}$, por lo cual $proy(q, AP')(p_j) = q(p_j)$, para todo $p_j \in AP'$

La *proyección de* $\mathcal{A}$ *con respecto a* AP' es el ABGE sobre $\Sigma := 2^{AP'}$

$$\mathcal{A} := (Q', Q'_0, \rho'. \mathcal{F}', l')$$

donde

- $Q' := proy(Q, AP')$
- $Q'_0 := proy(Q_0, AP')$

- $\rho' : Q' \longrightarrow 2^{Q'}$ se define por la regla siguiente:

  $q'_2 \in \rho'(q'_1)$ si y sólo si existen $q_1,\ q_2 \in Q$ tales que

  $q'_1 = proy(q_1,\ AP'),\ q'_2 = proy(q_2,\ AP')$ y $q_2 \in \rho(q_1)$

- $l' : Q' \longrightarrow 2^{\Sigma}$ se define por la regla:

  $$l'(q') := \{\{p_j \in AP' \mid q'(p_j) = 1\}\}$$

  - Si $\mathcal{F} = \{F_1,\ F_2, \ldots,\ F_m\}$, entonces $\mathcal{F}' = \{F'_1,\ F'_2, \ldots,\ F'_m\}$, donde $F'_j := proy(F_j,\ AP')$ para cada $j = 1,\ 2, \ldots,\ m$.

Denotamos a este ABGE de manera natural por $proy(\mathcal{A},\ AP')$.

Teorema 4.4.5. $\mathcal{A} \preceq proy(\mathcal{A}, AP')$.

Prueba:

Sea $H := \{(q,\ proy(q,\ AP')) \mid q \in Q\}$. Probemos que H es una relación de simulación de $\mathcal{A}$ en $proy(\mathcal{A},\ AP')$.

Sea $(q,\ proy(q,\ AP')) \in H$. Entonces, $l'(proy(q,\ AP')) = \{C \cap AP' \mid C \in l(q)\}$, por definición de l'

Ahora, sea $\pi : q_0,\ q_1,\ q_2, \ldots$ una corrida legal de $\mathcal{A}$ que parte de $q = q_0$. Definamos la secuencia $proy(\pi, AP') : q'_0,\ q'_1,\ q'_2, \ldots$, donde $q'_i := proy(q_i,\ AP')$ para todo $i \geq 0$.

Afirmamos que $proy(\pi, AP')$ es una corrida legal de $proy(\mathcal{A},\ AP')$ que parte de $q'_0 = proy(q,\ AP')$.

Sea $i \in \mathbb{N}_0$. Como π es una corrida de $\mathcal{A}$, $q_{i+1} \in \rho(q_i)$. Entonces, por definición de ρ', $q'_{i+1} \in \rho'(q'_i)$, puesto que $q'_i := proy(q_i, AP')$ y $q'_{i+1} := proy(q_{i+1},\ AP')$.

Por tanto, $proy(\pi,\ AP')$ es una corrida de $proy(\mathcal{A},\ AP')$. Además, como $q = q_0$, esta corrida parte de $proy(q,\ AP')$.

Para probar que también es una corrida legal, sea $j \in \{1,\ 2, \ldots,\ m\}$. Como π es una corrida legal de $\mathcal{A}$, $inf(\pi) \cap F_j \neq \emptyset$. Tomemos un elemento s de $inf(\pi) \cap F_j$. Entonces, hay un número infinito de valores del subíndice $i \in \mathbb{N}_0$ tales que $q_i = s$ y $s \in F_j$. De aquí, $q'_i = proy(s,\ AP')$ para esos mismos valores de i y $proy(s,\ AP') \in F'_j$, por definición de F_j. Por lo tanto, $proy(s,\ AP') \in inf(proy(\pi,\ AP')) \cap F'_j$, lo que

muestra que este conjunto no es vacío. Concluimos así que $proy(\pi,\ AP')$ es una corrida legal de $proy(\mathcal{A},\ AP')$ que parte de $proy(q,\ AP')$.

Finalmente, el que π y $proy(\pi,\ AP')$ se corresponden es inmediato de la definición de H.

Por lo tanto H es una relación de simulación de $\mathcal{A}$ en $proy(\mathcal{A},\ AP')$.

Para terminar, sea $q_0 \in Q_0$. Entonces $proy(q_0,\ AP') \in Q'_0$, por definición de Q'_0, con lo cual $(q_0, proy(q_0,\ AP')) \in H$.

Por lo tanto, $\mathcal{A} \preceq proy(\mathcal{A},\ AP')$, como se quería probar. □

Aplicando el corolario 4.4.3 y gracias al teorema 4.4.5 que acabamos de probar, obtenemos de inmediato el resultado más importante de este capítulo.

Corolario 4.4.6. *Dadas las asunciones del teorema 4.4.5, se tiene para toda fórmula φ de la LTLP con $Voc(\varphi) \subseteq AP'$ que*

$$si\ proy(\mathcal{A},\ AP') \models \varphi,\ entonces\ \mathcal{A} \models \varphi.$$

Corolario 4.4.7. *Si $AP'' \subseteq AP' \subseteq AP$, entonces $proy(\mathcal{A},\ AP') \preceq proy(\mathcal{A},\ AP'')$.*

Prueba:

Es tedioso pero sencillo probar que $proy(\mathcal{A},\ AP'') = proy(proy(\mathcal{A},\ AP'),\ AP'')$. Aplicando ahora el teorema 4.4.5, obtenemos que $proy(\mathcal{A},\ AP') \preceq proy(proy(\mathcal{A},\ AP'),\ AP'') = proy(\mathcal{A},\ AP'')$ □

Este corolario nos hace ver que por cada torre ascendente de subconjuntos de AP

$$\emptyset = AP_0 \subseteq AP_1 \subseteq \ldots \subseteq AP_r = AP$$

hay una torre ascendente de proyecciones de $\mathcal{A}$

$$proy(\mathcal{A}, AP) = proy(\mathcal{A}, AP_r) \preceq \ldots \preceq proy(\mathcal{A}, AP_1) \preceq proy(\mathcal{A}, AP_0) = proy(\mathcal{A}, \emptyset)$$

donde $\mathcal{A} = proy(\mathcal{A}, AP)$ y $proy(\mathcal{A}, \emptyset)$ es un ABGE trivial.

El preorden de simulación resulta ser un orden parcial sobre la colección de las proyecciones de un ABGE dado $\mathcal{A}$ con respecto a los subconjuntos de la forma $\Sigma = 2^{AP'}$, que denotamos por $Proy(\mathcal{A})$.

Más aún, el álgebra booleana $(\Sigma, \cup, \cap, -.\ AP, \emptyset)$ de los subconjuntos de AP induce un álgebra booleana sobre la colección de las proyecciones de $\mathcal{A}$, $Proy(\mathcal{A})$.

Cada proyección determina una función sobreyectiva $proy_j : Q \longrightarrow proy(Q, AP_j)$ (conocida como una *función de abstracción*), que naturalmente asocia a cada estado q en Q su proyección con respecto a AP_j, $proy(q, AP_j)$.

A su vez, la función $proy_j$ define una relación de equivalencia $\equiv_j$ sobre Q mediante la regla:

$q_1 \equiv_j q_2$ si y sólo si $proy_j(q_1) = proy_j(q_2)$ si y solo si $proy(q_1, AP_j) = proy(q_2, AP_j)$

Las clases de equivalencia determinadas por esta relación claramente corresponden a los estados de $proy(\mathcal{A}, AP_j)$

Ahora bien, si $\pi' = q'_0,\ q'_1,\ q'_2, \ldots$ es una corrida de aceptación de $proy(\mathcal{A}, AP')$, entonces la preimagen de π' es la sucesión de conjuntos de estados de $\mathcal{A}$

$$proy^{-1}(\pi') : proy^{-1}(q'_0),\ proy^{-1}(q'_1),\ proy^{-1}(q'_2), \ldots$$

El problema es encontrar una corrida de aceptación de $\mathcal{A}$. Esto mismo lo podemos plantear respecto de proyecciones sucesivas.

La función de abstracción h tiene por objetivo abstraer cada estado del modelo concreto a un estado en el modelo abstracto que únicamente contiene las variables pertenecientes a $Voc(\varphi)$. Por ejemplo, si estamos modelando un sistema que tiene 4 válvulas $v_1, ...,\ v_4$ y dos botones $b_1,\ b_2$ y la propiedad φ a verificar tiene $Voc(\varphi) = \{v_2,\ v_4,\ b_1\}$, entonces los estados del modelo abstracto tendrán la forma $(v_{2i},\ v_{4i},\ b_{1i})$, para $i \in \mathbb{N}$, donde i es él número de estados abstractos generados.

En la figura 4.1 se muestra como el espacio de estados es particionado en bloques inducidos por la función de abstracción h.

4.5. Idea para Calcular $proy(\mathcal{A}, AP')$

Generalmente AP' es muy pequeño en relación a AP, puesto que en primer instancia $AP' = Voc(\varphi)$, donde φ es una propiedad a comprobar. Podemos generar $proy(\mathcal{A},\ AP')$ de varias maneras directamente a partir de las ecuaciones que se derivan de los diagramas de escalera, utilizando el mismo procedimiento que empleamos para generar explícitamente el espacio de estados de $\mathcal{A}$, pero caemos en un

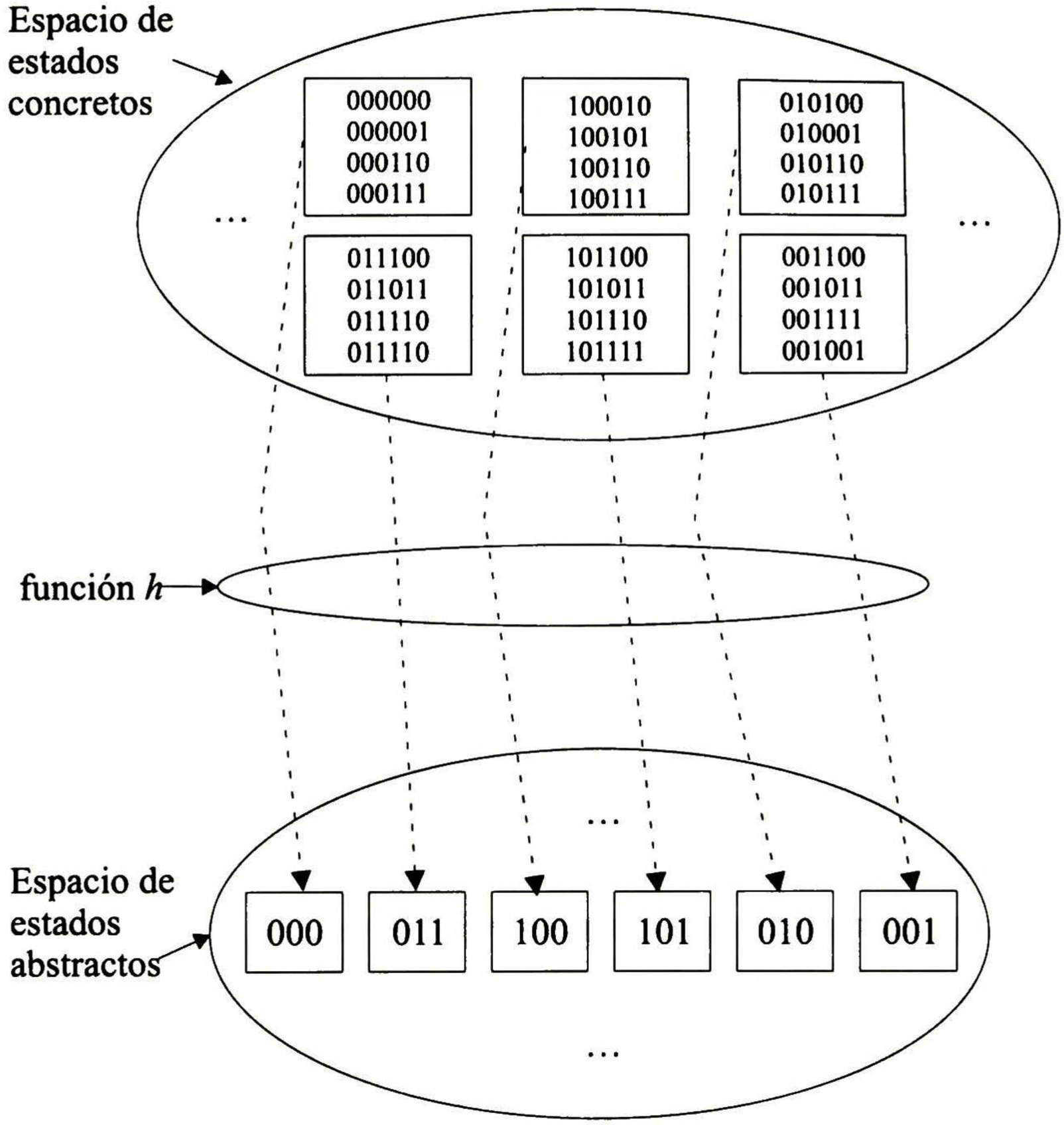


Figura 4.1: Función de abstracción h que actúa sobre un espacio de estados concreto para generar un espacio de estados abstractos.

problema intratable, por que al evaluar las ecuaciones tenemos que considerar todos los valores posibles de las variables de AP que no pertenecen a $Voc(\varphi)$. Si este conjunto de variables es grande ($|AP| > 30$ ya es demasiado), nos toma mucho tiempo generar la abstracción $proy(\mathcal{A},\ AP')$. El mayor ahorro es el consumo de memoria. Debemos pues, intentar construir $proy(\mathcal{A}, AP')$ de otra manera.

El tamaño del espacio de estados de $proy(\mathcal{A}, AP')$ es cuando mucho igual a $2^{|AP'|}$ y al igual que para los estados del modelo concreto cada estado posible se representa mediante un vector booleano de longitud igual a $|AP'|$. Podemos entonces

restringirnos a la consideración de cada uno de los estados abstractos y tratar de determinar el conjunto de sucesores directamente. Para lograr esto tomamos el conjunto de ecuaciones y sustituimos las variables que etiquetan el estado por los valores que toman en él. Entonces lo que tratamos de obtener son los valores posibles de esas mismas variables en un estado siguiente. En caso de que alguna de las variables de AP' no sea una variable de estado siguiente, tendremos que considerar todas las combinaciones de sus valores posibles. Pero esto suele hacerse sólo para un número muy reducido de ellas. En general, determinar los valores posibles de las variables en AP' a partir de las ecuaciones requiere de una cantidad considerable de simplificaciones, esto es, de aplicación de reglas del álgebra booleana que tendrían que programarse. En vez de hacerlo, podemos utilizar los algoritmos para BDDs, de la manera siguiente:

1. Sustituyamos cada variable de estado siguiente $v\_est\_s_i$ que aparezca por primera vez por una nueva variable $v\_est\_s_i'$ (indicando así el valor de $v\_est\_s_i$ en el estado siguiente) y también sustituyamos $v\_est\_s_i$ por $v\_est\_s_i'$ en cada una de las ecuaciones subsiguientes.

2. Formemos la conjunción de todas las ecuaciones obtenidas en el paso anterior, sustituyendo el signo de igualdad por el de doble implicación, y construyamos el BDD de la fórmula resultante. Si hay variables de AP' que no aparezcan en la fórmula, entonces consideraremos todas sus combinaciones posibles una vez obtenidos los resultados del paso 4.

3. Apliquemos cuantificación existencial para deshacernos de todas las variables primadas que no corresponden a variables de AP' y de todas las variables que no son primadas.

4. Apliquemos la operación *allsat*. Como resultado, debemos obtener todos los conjuntos de combinaciones posibles de valores de las variables en un estado siguiente. Esto es, cada solución que arroje el algoritmo *allsat* deberá determinar una colección de vectores booleanos que designan estados sucesores del estado actual en $proy(\mathcal{A},\ AP')$. Estos vectores difieren entre sí por los valores que asignamos a las variables que no fueron consideradas en el BDD y que pertenecen

a AP'

Estos pasos los repetimos para cada uno de los posibles estados abstractos. Los que tengan asociados BDDs triviales iguales a 0 deben ser eliminados, puesto que no debemos tener estados finales; los que tengan asociados BDDs triviales iguales a 1 tienen a todos los estados abstractos como sucesores. Los vectores que no se obtengan en corrida alguna corresponden a estados no alcanzables desde un estado inicial y por tanto no deben ser tomados en cuenta.

4.6. Incorporación de la Técnica de Abstracción en la Herramienta Desarrollada

Para construir el modelo abstracto de un sistema necesitamos como entradas: el archivo con las ecuaciones que representan el diagrama de escalera (con la información opcional de estados iniciales, marcados, prohibidos y precedencias) y la propiedad a verificar escrita en LTLP. El archivo de entrada que representa la información del sistema es el mismo que se utiliza para generar un modelo concreto. Los cambios realizados en la herramienta operan al nivel de los algoritmos de construcción.

Al igual que en la construcción del modelo concreto, la construcción de $proy(\mathcal{A},\ AP')$ es incremental y conforme se construye cada estado con sus transiciones se escribe el archivo que contiene el modelo abstracto. Los algoritmos para construir el modelo a partir de las ecuaciones del sistema sufrieron cambios pero en esencia siguen trabajando de la misma manera. En las siguientes secciones mostraremos brevemente los cambios.

En el capítulo anterior por conveniencia mostramos primero el algoritmo para crear los estados sucesores y después el algoritmo para crear los estados iniciales, ahora mostraremos primero el algoritmo para crear los estados iniciales abstractos.

4.6.1. Estados Iniciales

Antes de aplicar el algoritmo *crearEstadosInicialesBDD* se crea la ecuación del paso 2 de la sección 4.5. Dicha ecuación se crea al hacer la lectura del archivo del

sistema y nos sirve para generar el BDD del sistema de ecuaciones de estado siguiente.

Para la construcción de los estados iniciales necesitamos saber cuales son las variables que están en $Voc(\varphi)$ y de esas variables cuáles están definidas en la sección iniciales y con que valor. En caso que no se hayan definido variables iniciales tomamos el estado inicial como aquél donde las variables de $Voc(\varphi)$ tienen valor booleano 0.

En la figura 4.2 mostramos el algoritmo que se encarga de la generación de los estados iniciales abstractos. Al comienzo del algoritmo tenemos únicamente el número de variables iniciales $nIniciales$, las variables que pertenecen a $Voc(\varphi)$ y la ecuación lógica, *ecuacion*, para crear el BDD del sistema de ecuaciones de estado siguiente. El resultado son los estados iniciales del ABGE abstracto $\mathcal{A}'$ y la lista de estados sucesores abstractos directos, *lista*, de los estados iniciales abstractos.

La diferencia principal entre este algoritmo y el anterior (para generar estados iniciales concretos) es que se ha incorporado el uso de los BDDs y las funciones *existencial* y la función *allsat*. La función existencial nos permite eliminar todas las variables que no son de interés para nosotros, dejando únicamente las variables de estado siguiente que pertenecen a $Voc(\varphi)$. La función allsat nos proporciona los valores que satisfacen el sistema de ecuaciones de estado siguiente.

Con el uso de los BDDs obtenemos de manera eficiente los valores de las variables de estado siguiente que pertenecen a $Voc(\varphi)$ y así la plantilla de estados sucesores del estado actual. El algoritmo termina una vez que se han construido los estados iniciales abstractos definidos y sus sucesores directos. El espacio de estados generado para los estados iniciales es igual a $2^{nIniciales}$

4.6.2. Estados Sucesores

El algoritmo para crear los estados sucesores abstractos de los estados iniciales abstractos tiene la función de terminar la construcción del ABGE $\mathcal{A}'$.

Una vez terminada la construcción de los estados iniciales abstractos estamos en condiciones para aplicar el algoritmo $crearEstadosSucesoresBDD$ el cual se muestra en la figura 4.3. El algoritmo necesita la ecuación lógica, *ecuacion*, la lista de estados iniciales abstractos *lista* y la lista de variables que pertenecen a $Voc(\varphi)$. El cuerpo

crearEstadosInicialesBDD

1: **Entrada:** $nIniciales$, $VarFormula$, $ecuacion$
2: **Salida:** $\mathcal{A}'$, *lista*
3: **Si** $nIniciales == 0$
4: **Entonces**
5: Asignar el valor 0 a las variables del estado s
6: Agregar el estado s a $\mathcal{A}'$
7: Verificar si s es un estado marcado
8: **Si** s es un estado marcado
9: **Entonces**
10: agregar s a la lista de estados marcados
11: Evaluar la *ecuacion* en el estado inicial y obtener el BDD de las ecuaciones.
12: Aplicar la función existencial al BDD //sobre las variables que no son de interés
13: Aplicar la función *allsat* al BDD //para obtener los valores de las variables de interes
14: Agregar los sucesores abstractos de s a *lista*
15: **Regresa** $\mathcal{A}'$, *lista*
16: **Si no**
17: $j = 2^{nIniciales}$
18: **Para cada** $h = 0$ hasta $h < j$ **Hacer**
19: Generar s asignando el valor de las variables iniciales y de las variables libres
20: Agregar el estado s a $\mathcal{A}'$
21: Verificar si s es un estado marcado
22: **Si** s es un estado marcado
23: **Entonces**
24: agregar s a la lista de estados marcados
25: Evaluar la *ecuacion* en el estado s y obtener el BDD
26: Aplicar la función existencial al BDD
27: Aplicar la función *allsat* al BDD
28: Agregar los sucesores abstractos de s a *lista*
29: **Regresa** $\mathcal{A}'$, *lista*

Figura 4.2: Algoritmo para crear los estados iniciales abstractos del ABGE $\mathcal{A}'$ y la lista de estados abstractos sucesores directos de los estados iniciales abstractos

del algoritmo es un ciclo mientras que se ejecuta hasta que la lista de estados por evaluar se vacía. Se toma en cada paso un estado y se evalúan las ecuaciones para determinar sus estados sucesores. Los estados sucesores que no han sido generados se agregan a la lista para su posterior evaluación.

crearEstadosSucesoresBDD

Entrada: *lista*, *ecuacion*
Salida: ABGE $\mathcal{A}'$
Mientras *lista* no sea vacía **Hacer**
    Tomar el primer elemento de *lista* y asignarlo al estado abstracto s
    Agregar s al ABGE $\mathcal{A}'$
    Verificar si s es un estado marcado
    Si s es un estado marcado
    Entonces
        agregar s a la lista de estados marcados
    Evaluar *ecuacion* y obtener el BDD de las ecuaciones.
    Aplicar la función existencial al BDD
        //sobre las variables que no son de interés
    Aplicar la función *allsat* al BDD
        //para obtener los valores de las variables de interés
    Obtener la plantilla de sucesores abstractos de s
    Para cada $i = 0$ hasta $i < 2^I$ **Hacer**
        //donde I es el número de variables de entrada en $Voc(\varphi)$
        Completar s' //mediante la plantilla obtenida y el valor de i
        Si la transición a s' es permitida //cumple precedencias y no es prohibido
        Entonces
            Si el estado s' se encuentra en *lista*
            Entonces
                generar una transición hacia s'
            Si no
                agregarlo a *lista* y generar una transición hacia s'
    Borrar s de *lista*
Escribir los estados marcados
Regresa $\mathcal{A}'$

Figura 4.3: Algoritmo para crear incrementalmente los estados sucesores abstractos del ABGE $\mathcal{A}'$ a partir de los estados iniciales abstractos.

El algoritmo termina una vez que se ha vaciado la *lista*, y su orden es de $2^{|AP'|}$. que es cuando tiene que generar todo el espacio de estados abstractos.

4.6.3. Estados Marcados

El algoritmo para determinar los estados abstractos marcados es igual al algoritmo para determinar los estados marcados en el modelo concreto. Antes de evaluar un estado abstracto debemos preguntar si se debe marcar ese estado o no. La diferencia es que ahora el algoritmo recibe un estado abstracto y se determina en base a las variables y valores determinados para ese estado si cumple con los valores de la variables necesarios para marcarlo. Las variables que se toman en cuenta para marcar un estado son las que están presentes en $Voc(\varphi)$. Un estado concreto marcado se proyecta en un estado marcado abstracto.

4.6.4. Estados Prohibidos

Los estados prohibidos abstractos se determinan de la misma manera que los marcados abstractos. Para prohibir un estado abstracto tomamos en cuenta únicamente los valores de las variables pertenecientes a $Voc(\varphi)$ y en base a los valores de las variables que se definen para prohibir un estado y que también aparecen en $Voc(\varphi)$ decidimos si el estado evaluado se prohíbe o no. Un estado prohibido concreto se proyecta en un estado prohibido abstracto.

4.6.5. Precedencias

Las precedencias de variables toman sentido únicamente para las variables que aparecen en $Voc(\varphi)$. Si alguna de las variables que forman la precedencia no pertenece a $Voc(\varphi)$, entonces la regla de precedencia se anula. El algoritmo es el mismo que el utilizado para definir las precedencias en el modelo concreto. Se pregunta si una transición es posible entre dos estados abstractos en base a los valores de las variables que intervienen en la precedencia y que están presentes en $Voc(\varphi)$. De igual forma la precedencia puede ser fuerte o débil.

En la siguiente sección mostraremos resultados obtenidos al construir modelos utilizando la simulación por proyección de variables y su verificación formal.

4.7. Aplicación de Abstracciones

Utilizaremos el ejemplo de las bombas alternantes del capitulo 3 para ilustrar el uso de la técnica de simulación por proyección de variables.

Algunas de las propiedades que podemos plantear para el sistema de las bombas son:

1. Si la bomba 1 esta encendida y la bomba 2 no, entonces eventualmente si no se detecta el nivel bajo bajo, entonces eventualmente la bomba 1 se apaga y la bomba 2 se enciende.

2. Si la bomba 2 esta encendida y la bomba 1 no, entonces eventualmente si no se detecta el nivel bajo bajo, entonces eventualmente la bomba 2 se apaga y la bomba 1 se enciende.

3. Si la bomba 1 está encendida y no se detecta en nivel alto y en el siguiente instante se detecta el nivel alto, entonces dos instantes después las 2 bombas estarán encendidas.

4. Si la bomba 2 está encendida y no se detecta en nivel alto y en el siguiente instante se detecta el nivel alto, entonces dos instantes después las 2 bombas estarán encendidas.

5. Si no se detecta el bit alternante y el nivel es bajo, entonces en el siguiente instante se enciende la bomba 1.

6. Si se detecta el bit alternante y el nivel es bajo, entonces en el siguiente instante se enciende la bomba 2.

7. Si ninguna bomba está encendida y se detecta el nivel bajo, entonces al siguiente instante se enciende una de las dos bombas.

8. Si se detecta el nivel bajo bajo y la señal OSR, entonces en el siguiente instante no se debe detectar la señal OSR.

9. Si se detecta el bit interno B301 y el bit alternante, entonces el bit alternante cambia de valor en el siguiente instante.

10. Si el nivel es bajo y la bomba 1 esta apagada, entonces en el siguiente instante la bomba 1 se enciende.

11. Si el nivel es bajo y la bomba 2 esta apagada, entonces en el siguiente instante la bomba 2 se enciende.

12. Si el nivel es alto y las 2 bombas están encendidas, entonces eventualmente ya no se detectará el nivel alto.

13. Si el nivel es alto y las 2 bombas están encendidas, entonces eventualmente ya no se detecta el nivel bajo bajo.

14. Si se detecta la señal OSR, entonces en el siguiente instante ya no se debe detectar.

15. Si se detecta el nivel bajo bajo y la señal OSR, entonces en el siguiente instante se enciende el bit interno B301.

Los resultados de la verificación de las propiedades utilizando los modelos abstractos se muestran en la tabla 4.1, así como el tamaño de los modelos. El tiempo de construcción de los modelos fue de .01 s para cada uno de los ellos.

Las propiedades que resultaron válidas en el modelo abstracto son también válidas en el modelo concreto. Por otra parte las propiedades que no se cumplieron en la verificación del modelo abstracto arrojaron un contraejemplo abstracto. Ese contraejemplo abstracto puede ser que no corresponda a un contraejemplo en el modelo concreto. A los contraejemplos abstractos que no existen en el modelo concreto los llamaremos *contraejemplos espurios*. Pero si el contraejemplo abstracto efectivamente corresponde a un contraejemplo en el modelo concreto, entonces decimos que el contraejemplo abstracto es *real*.

Podemos observar que la mayoría de las propiedades no se cumplen en los modelos abstractos, pero no quiere decir que efectivamente no se cumplen en el modelo concreto, en el capitulo siguiente se muestra una técnica para decidir efectivamente si un contraejemplo abstracto es real.

#Prop.	Codificación en LTL	Res. Verif.	#Est.	#Trans.
1	$\Box((B1 \wedge \neg B2) \rightarrow (\Diamond(\neg LL \rightarrow \Diamond(B2 \wedge \neg B1))))$	Válida	8	26
2	$\Box((B2 \wedge \neg B1) \rightarrow (\Diamond(\neg LL \rightarrow \Diamond(B1 \wedge \neg B2))))$	Válida	8	26
3	$\Box((B1 \wedge \neg H \wedge \Diamond H) \rightarrow \bigcirc\bigcirc(B1 \wedge B2))$	No se cumple	8	42
4	$\Box((B2 \wedge \neg H \wedge \Diamond H) \rightarrow \bigcirc\bigcirc(B1 \wedge B2))$	No se cumple	8	42
5	$\Box((\neg B301 \wedge L) \rightarrow \bigcirc B1)$	No se cumple	8	54
6	$\Box((B301 \wedge L) \rightarrow \bigcirc B2)$	No se cumple	8	54
7	$\Box((L \vee \neg B1 \wedge \neg B2) \rightarrow \Diamond(B1 \vee B2))$	No se cumple	8	48
8	$\Box((LL \wedge OSR) \rightarrow \bigcirc\neg OSR)$	Válida	4	8
9	$\Box((B301 \wedge B302) \rightarrow \neg B302))$	No se cumple	4	10
10	$\Box((L \wedge \neg B1) \rightarrow \bigcirc B1)$	No se cumple	4	16
11	$\Box((L \wedge \neg B2) \rightarrow \bigcirc B2)$	No se cumple	4	16
12	$\Box((H \wedge B1 \wedge B2) \rightarrow \Diamond\neg H)$	No se cumple	8	42
13	$\Box((H \wedge B1 \wedge B2) \rightarrow \Diamond\neg LL)$	No se cumple	16	88
14	$\Box(OSR \rightarrow \bigcirc\neg OSR$	No se cumple	2	4
15	$\Box((LL \wedge OSR) \rightarrow \bigcirc B301)$	Válida	6	12

Tabla 4.1: Propiedades establecidas para el sistema de las bombas alternantes.

El siguiente ejercicio que hicimos fue replicar el sistema de las bombas alternantes para ver cuantos módulos independientes era capaz la herramienta de modelado de manejar, construyendo únicamente el modelo para un módulo dado con respecto a ciertas propiedades. La forma de hacerlo es similar a como se aplica en [39] para los módulos de sistemas de alarmas independientes. Los resultados se muestran en la tabla 4.2. Si comparamos los resultados podemos ver que los tiempo son muy pequeños para la construcción del modelo abstracto, y el espacio de estados es el mismo no importando el número de tanques independientes. Con estos resultados podemos suponer que podemos verificar sistemas grandes, únicamente tomando en cuenta las variables implicadas en la especificación, por muy grande que sea la especificación el número de variables será mucho menor que el número de variables totales del sistema.

Posteriormente se combinaron algunas propiedades de distintos módulos y se generó el modelo abstracto para ellos. Los resultados se muestran en la tabla 4.3. El número de variables totales se refiere a las variables de los módulos de tanques

#Propiedad	#Módulos	Tiempo de cons.(s)	#Estados	#Trans.	#Var. Totales
1	15	.03	8	26	120
1	30	.06	8	26	240
1	45	.11	8	26	360
1	60	.18	8	26	480
3	15	.03	8	42	120
3	30	.06	8	42	240
3	45	.12	8	42	360
3	60	.19	8	42	480
8	15	.02	4	8	120
8	30	.03	4	8	240
8	45	.07	4	8	360
8	60	.1	4	8	480
10	15	.02	4	16	120
10	30	.04	4	16	240
10	45	.07	4	16	360
10	60	.1	4	16	480
15	15	.03	6	12	120
15	30	.06	6	12	240
15	45	.09	6	12	360
15	60	.13	6	12	480

Tabla 4.2: Replica de sistemas de bombas construyendo únicamente el modelo abstracto para uno de los tanques independientes.

La diferencia es notable cuando los modelos son hechos con propiedades que señalan módulos independientes, ya que sus variables no tienen interacción entre sí, pero tienen que ser tomadas en cuenta y eso produce una explosión combinatoria. Para el sistema de 5 tanques se generan 32 transiciones por cada estado sucesor diferente en la plantilla de sucesores. El espacio de estados realmente es pequeño, pero el número de transiciones es enorme. Si las variables se encuentran relacionadas o si tenemos en las variables de entrada relaciones de precedencia se puede reducir en buena medida el número de transiciones del sistema.

#Prop.	#Módulos.	T. constr.(s)	#Estados	#Trans.	#Var. Espec.	#Var. Tot.
1	2	.02	64	676	6	16
1	3	.22	512	17576	9	24
1	4	44.49	4096	456976	12	32
1	5					40
3	2	.03	64	1628	6	16
3	3	.68	512	64296	9	24
3	4	384.26	4096	2562800	12	32
3	5					40
8	2	.01	16	64	4	16
8	3	.03	64	512	6	24
8	4	.12	256	4096	8	32
8	5	1	1024	32768	10	40
10	2	.01	16	256	4	16
10	3	.04	64	4096	6	24
10	4	.51	256	65536	8	32
10	5	15.88	1024	1048676	10	40
15	2	.01	36	144	6	16
15	3	.07	216	1728	9	24
15	4	.9	1296	20736	12	32
15	5	152.77	7776	248832	15	40

Tabla 4.3: Replica de sistemas de bombas construyendo el modelo abstracto para los tanques implicados en la especificación.

4.8. Conclusiones

Utilizar las abstracciones nos reduce en gran medida el espacio de estados del modelo del sistema y además si utilizamos BDDs podemos reducir en gran medida el tiempo de cálculo para generar los estados abstractos sucesores de un estado abstracto dado. Al utilizar el cálculo explícito junto con operaciones de cálculo simbólico podemos obtener los beneficios de ambas técnicas.

Sabemos que si la propiedad a verificar se cumple en el modelo abstracto, entonces esa misma propiedad se cumple en el modelo concreto. Pero si por el contrario la propiedad no se cumple, hasta este punto no estamos en condiciones de saber si efectivamente la propiedad no se cumple en el modelo concreto.

En el capítulo siguiente utilizaremos el contraejemplo abstracto obtenido en el

proceso de verificación de un modelo abstracto para saber si efectivamente dicho contraejemplo es un contraejemplo en el modelo concreto.

Capítulo 5

Detección de Contraejemplos Espurios

5.1. Contenido del Capítulo

En este capítulo mostraremos como detectar si un contraejemplo obtenido mediante la verificación de un modelo abstracto corresponde a un contraejemplo en un modelo concreto del sistema. En caso de detectar un contraejemplo espurio señalamos como refinar la estructura abstracta, para posteriormente volver a efectuar el proceso de verificación sobre la estructura refinada. En la sección 5.2 enunciamos las condiciones mediante las cuales se genera un contraejemplo espurio, la forma en que éste puede ser detectado y los elementos que intervienen en su detección. En la sección 5.3 detallamos los algoritmos para la detección de los contraejemplos espurios. En la sección 5.4 explicamos por que un contraejemplo localizado debe cumplir con las condiciones de marcado de los ABGEs. En la sección A.2.4 mostramos una manera de realizar un refinamiento sobre un modelo abstracto de forma automática. En la sección 5.6 utilizamos los contraejemplos abstractos obtenidos en el proceso de verificación de los modelos abstractos y las propiedades definidas en el capítulo anterior, con el fin de saber si pueden corresponder con un contraejemplo concreto, en caso de no serlo se efectúa un refinamiento sobre los modelos abstractos y volvemos a verificarlos. Por último en la sección 6 expresamos algunas conclusiones sobre utilizar la detección de

los contraejemplos espurios y los refinamientos de los modelos. También comentamos sobre el trabajo que puede hacerse para obtener otros contraejemplos abstractos.

5.2. Contraejemplos Reales y Espurios.

Una de las ventajas al utilizar la comprobación de modelos en la verificación formal es que si la propiedad que estamos verificando no se cumple, el método nos arroja como resultado un contraejemplo. Un contraejemplo es una secuencia de estados comenzando por un estado inicial (recordemos que podemos especificar más de un estado inicial) en donde la propiedad especificada no se cumple.

Al analizar un contraejemplo obtenido en la etapa de verificación podemos saber cuales son las condiciones en las que se encontraba el sistema cuando sucedió el incumplimiento de la propiedad. Podemos determinar en que parte el sistema está fallando ó saber si la propiedad que se está pidiendo está mal planteada.

Si un contraejemplo fue obtenido realizando la verificación en un modelo concreto del sistema entonces ese contraejemplo es real, es decir, podemos estar seguros que dicho contraejemplo muestra que en el modelo del sistema no se cumple la propiedad que se especificó.

Por otra parte, si el contraejemplo fue obtenido en la verificación de un modelo abstracto del sistema, **generado mediante la técnica de simulación**, no podemos estar completamente seguros que dicho contraejemplo corresponda a un contraejemplo en el modelo concreto, sabemos que:

$$Si\ \mathcal{M}_{\mathcal{A}} \models \varphi, entonces\ \mathcal{M} \models \varphi$$

pero en general no podemos decir que:

$$Si\ \mathcal{M}_{\mathcal{A}} \not\models \varphi, entonces\ \mathcal{M} \not\models \varphi$$

Entonces debemos encontrar una manera de asegurar que el contraejemplo que obtenemos en la verificación de un modelo abstracto corresponde a un contraejemplo en el modelo concreto. Si el contraejemplo encontrado no corresponde a un contraejemplo real lo llamamos contraejemplo *espurio*.

En [34] se muestra una manera de detectar los contraejemplos espurios. Modificamos de acuerdo a nuestras necesidades esa idea en la herramienta de modelado desarrollada, con el fin de realizar la detección de contraejemplos espurios obtenidos en la verificación de modelos abstractos.

5.3. Comprobación del Contraejemplo Abstracto

Una vez que se ha realizado la verificación del modelo abstracto del sistema en la herramienta VR [43] y se ha generado un contraejemplo, debemos determinar si ese contraejemplo es la imagen de un contraejemplo real en el modelo concreto del sistema.

En la figura 5.1 podemos observar la forma en que un contraejemplo abstracto puede corresponder a un contraejemplo concreto. La simulación por proyección de variables se ha efectuado con respecto a las primeras tres variables, los círculos corresponden a los estados abstractos y los rectángulos corresponden a los conjuntos de estados concretos que se proyectan en el mismo estado abstracto. Las flechas más gruesas forman la trayectoria del contraejemplo concreto.

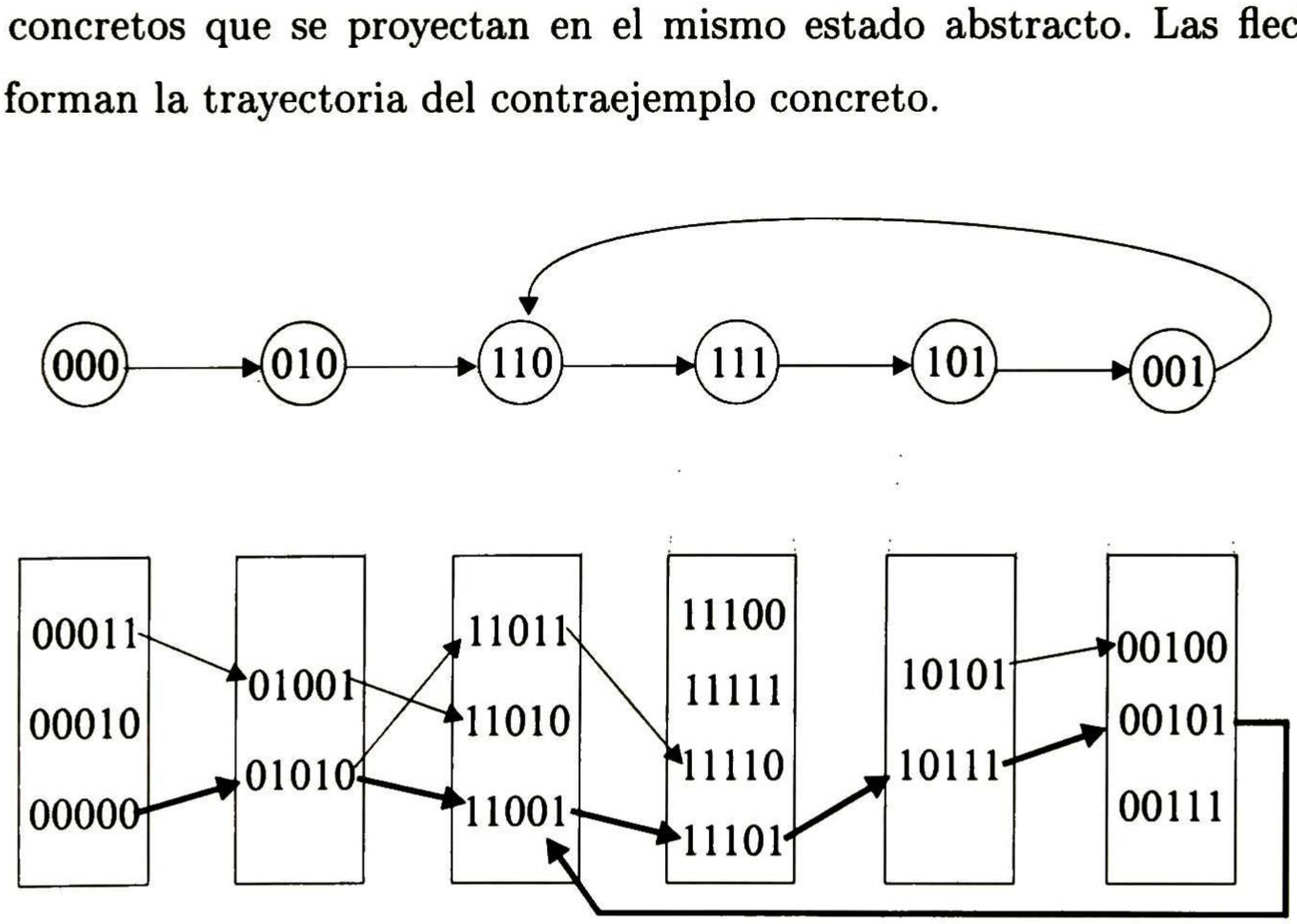


Figura 5.1: Correspondencia entre un contraejemplo abstracto y un contraejemplo concreto.

5.3.1. Preparación del Contraejemplo Abstracto

La condición de aceptación de los ABGE dice que debe haber una secuencia de estados en donde se visite infinitamente por lo menos un estado de cada conjunto de la colección de aceptación. En la herramienta de comprobación de modelos VR si al verificar una propiedad ésta no se cumple, entonces obtenemos como resultado un contraejemplo. El contraejemplo obtenido está formado por dos partes: prefijo y periodo. El prefijo corresponde a los estados que se visitan antes de encontrar un ciclo en donde no se cumple la propiedad. El prefijo puede ser vacío, y se muestra en el mismo orden en el que fueron visitados los estados comenzando por el estado inicial. El periodo por su parte se muestra como un conjunto de estados que inducen una componente fuertemente conexa que conforma el ciclo donde no se cumple la propiedad. El orden de aparición de los estados en el periodo dado por la herramienta VR no nos dice nada acerca de la forma en que fueron visitados los estados.

El primer paso para poder averiguar si un contraejemplo abstracto dado es espurio o no, es formar un *contraejemplo abstracto cumplidor* a partir del contraejemplo obtenido en la herramienta VR. Le llamamos cumplidor porque el periodo deberá *cumplir* con las condiciones de marcado definidas en el modelo del sistema.

El algoritmo que se utiliza para encontrar el contraejemplo cumplidor se muestra en la figura 5.2. El algoritmo necesita como entrada el contraejemplo dado por la herramienta VR y la colección de estados marcados para poder elegir el siguiente estado que formará parte del ciclo. El prefijo que formará parte del contraejemplo cumplidor corresponde directamente con el prefijo del contraejemplo dado por la herramienta VR. Encontrar el periodo es entonces la meta del algoritmo.

La función $copiaPrefijo()$ se encarga de copiar el prefijo del contraejemplo original al contraejemplo cumplidor. La función $addCiclo(estado)$ adiciona el *estado* a la lista de estados del contraejemplo cumplidor que se forma incrementalmente. La función *sucesores(estado)* calcula los estados sucesores del *estado* al evaluar las ecuaciones de estado siguiente con los valores de las variables de $Voc(\varphi)$ en el estado actual.

El algoritmo para elegir un estado entre los sucesores de un estado en particular

```
busca ciclo
1:  Entrada: Contraejemplo
2:  Salida: Contraejemplocumplidor, numEstados
3:  copiaPrefijo()
4:  numEstados=0
5:  estadoObjetivo=Ultimo_del_prefijo
6:  addCiclo(estadoObjetivo)
7:  sucesores(estadoObjetivo)
8:  estadoActual=estadoObjetivo
9:  Mientras no_se_cierra_ciclo
10:     estadoElegido=eligeEstado(estadoActual,estadoObjetivo)
11:     addCiclo(estadoElegido)
12:     numEstados++
13:     sucesores(estadoElegido)
14:     estadoActual=estadoElegido
15: Regresa Contraejemplocumplidor, numEstados
```

Figura 5.2: Algoritmo para crear un ciclo a partir del conjunto de estados del periodo que inducen una componente fuertemente conexa.

se muestra en la figura 5.3. Este algoritmo se encarga de que el estado elegido cumpla con las condiciones de marcado (las cuales se establecieron en el archivo del sistema en la sección estados marcados), en caso de que ningún estado cumpla las condiciones de marcado en una elección determinada, simplemente se elige un estado, y ese estado se marca; para la siguiente elección si hay la opción de elegir ese estado nuevamente se le dará prioridad a algún otro que no haya sido elegido. Si las condiciones de marcado han sido satisfechas entonces el algoritmo buscará cerrar el ciclo.

Las entradas para el algoritmo *elígeEstado* son el estado actual, el estado objetivo (con el que se cierra el ciclo) y la colección de conjuntos de aceptación. La salida es el estado elegido y la determinación de si ya se cerró el ciclo.

La función $buscaEnMarcados(Nombre)$ verifica que el estado a elegir se encuentre en alguna de las colecciones de estados marcados, en caso de ser así lo elige y pone una marca a esa colección. La función $menorEnCiclo(estado)$ elige un estado de los sucesores del estado actual que no haya sido elegido anteriormente, al estado elegido se le pone una marca para saber cuantas veces ha sido elegido. La función $cerrarCiclo(estadoActual, estadoObjetivo)$ se encarga de verificar que el estado a

```
eligeEstado
1:  Entrada: Marcados, estadoActual, estadoObjetivo
2:  Salida: se_cierra_ciclo, estadoElegido
3:  transicion = estadoActual->Transi
4:  Si HayaColecciones por marcar
5:  Entonces
6:      Mientras transicion
7:          Si buscaEnMarcados(transicion->Nombre)
8:          Entonces
9:              Regresa transicion->Nombre, se_cierra_ciclo=falso;
10:         Si no
11:             transicion=transicion->Siguiente
12:     Regresa menorEnCiclo(estadoActual)
13: Si no
14:     Si cerrarCiclo(estadoActual,estadoObjetivo)
15:     Entonces
16:         Regresa estadoObjetivo, se_cierra_ciclo=verdadero;
17:     Si no
18:         Regresa menorEnCiclo(estadoActual)
```

Figura 5.3: Algoritmo para elegir un estado sucesor del estado actual.

elegir pueda cerrar el ciclo, siempre y cuando las colecciones de aceptación hayan sido visitadas en alguno de sus estados por el ciclo que se ha formado.

Una vez obtenido el contraejemplo cumplidor estamos en condiciones para aplicar los algoritmos para saber si el contraejemplo obtenido en la verificación de un modelo abstracto es espurio.

5.3.2. Algoritmos para Detectar si el Contraejemplo es Espurio

Los algoritmos para detectar si un contraejemplo abstracto es espurio se basan en el hecho de que dicho contraejemplo puede ser la imagen de un contraejemplo concreto.

El algoritmo para realizar la comprobación de un contraejemplo abstracto se muestra en la figura 5.4. El primer paso es verificar que el contraejemplo tenga el mismo número de variables que la propiedad utilizada. Después, debemos generar la lista de

los estados marcados abstractos a partir de las condiciones de marcado del archivo del sistema. Posteriormente, debemos probar la parte del prefijo abstracto. Si pasa la prueba debemos probar la parte del periodo.

```
generaContraejemplo
1:  Entrada: Contraejemplo
2:  Salida: contrajemplo_valido
3:  verificaNumvars()
4:  generaMarcados()
5:  Si pruebaPrefijo()=falso
5:  Entonces
6:      Desplegar el estado falla
7:      Desplegar el conjunto falla
8:  Si no
9:      nestados=buscaCiclo()
10:     Si SplithPathDoc(nestados)==falso
11:     Entonces
11:         Desplegar el estado falla
12:         Desplegar el conjunto falla
13:     Si no
14:         Desplegar el contraejemplo real
```

Figura 5.4: Algoritmo probar si el contraejemplo abstracto dado es espurio.

En el algoritmo mostrado en la figura 5.4 podemos observar que la comprobación del contraejemplo la hacemos en dos partes: primero comprobando que el prefijo abstracto pueda ser la imagen de un prefijo concreto y después comprobar la parte del periodo. El principio del método es el siguiente: tomando el estado inicial abstracto debemos encontrar los estados iniciales concretos que se proyectan en él. Si ningún estado inicial concreto se proyecta en él determinamos falla desde el estado inicial. Por el contrario, si existen estados iniciales que si se proyectan en él, el siguiente paso es determinar los sucesores de esos estados concretos, entonces aplicamos la proyección de variables a ese conjunto. El resultado lo intersecamos con el siguiente estado abstracto. Si la intersección es vacía, entonces determinamos falla; en caso contrario seguimos aplicando las mismas operaciones hasta terminar con el prefijo. Una vez terminado el prefijo buscamos el ciclo cumplidor y posteriormente analizamos la parte del periodo en la función *splithPathDoc*.

Al conjunto de estados concretos que se proyectan en un estado abstracto dado lo llamamos *preimagen* de ese estado abstracto. Dado un estado abstracto $\widehat{s}$, el conjunto de estados concretos s tales que $h(s) = \widehat{s}$ es denotado por $h^{-1}(\widehat{s})$. Se extiende a secuencias la definición de la siguiente manera: $h^{-1}(T)$ es el conjunto de trayectorias concretas dadas por la siguiente expresión

$$\{< s_1, ..., s_n > | \bigwedge_{i=1}^{n-1} h(s_i) = \widehat{s_i} \wedge I(s_1) \wedge \bigwedge_{i=1}^{n-1} R(s_i, s_{i+1})\}$$

La *imagen delantera* de un estado la definimos como el conjunto de sus estados sucesores.

El algoritmo para la prueba del prefijo se muestra en la figura 5.5. La entrada es el prefijo del contraejemplo abstracto y la salida es la determinación de que el prefijo abstracto es la imagen de un prefijo concreto o no. El conjunto cS guarda los estados que se encuentran en la intersección que se efectúa en cada iteración del algoritmo.

```
pruebaPrefijo
1:  Entrada: Contraejemplo
2:  Salida: prefijo_valido
3:  estadoActual=Contraejemplo
4:  estadosIniciales()
5:  cS=Interseccion()
6:  estadoActual=estadoActual->Siguiente
7:  j=1
8:  Mientras cS no sea vacío y j <prefijo
9:      j++
10:     ImagenDelantera()
11:     cS=Interseccion()
12:     Si cS no es vacío y j <prefijo
13:     Entonces
14:         estadoActual=estadoActual->Siguiente
15: Si cS no es vacío
16: Entonces
17:     Regresa prefijo_valido = verdadero
18: Si no
19:     Regresa prefijo_valido = falso
```

Figura 5.5: Algoritmo para determinar si el prefijo abstracto puede corresponder con un prefijo concreto.

La función *estadosIniciales*() nos devuelve los estados iniciales concretos del sistema. La función *Interseccion*() se encarga de intersecar el conjunto de estados concretos actuales con el estado abstracto actual del prefijo. La función *imagenDelantera*() nos regresa el conjunto de estados sucesores de los estados contenidos en el conjunto cS.

La principal diferencia entre nuestros algoritmos y los mostrados en [34] es que nosotros determinamos únicamente la imagen delantera de los estados concretos que se proyectan en los estados abstractos del contraejemplo y los intersecamos directamente con el siguiente estado abstracto del contraejemplo, sin tener que determinar la preimagen de los estados abstractos, que en la mayoría de los casos es muy caro, y posteriormente realizar la intersección.

Si se ha pasado la prueba del prefijo entonces aplicamos el algoritmo *buscaCiclo*(), mostrado en la figura 5.2, que nos regresa el contraejemplo cumplidor.

El contraejemplo espurio puede ser determinado a lo largo de la trayectoria del prefijo sin mucha dificultad, pero al llegar al periodo puede ocurrir que el contraejemplo espurio se produzca al momento del cerrar el ciclo. En la figura 5.6 se muestra un contraejemplo que aparentemente es un contraejemplo válido, pero si analizamos con detalle podemos ver que el contraejemplo es espurio y estaba oculto(enrollado) al cerrar el ciclo. Por tanto necesitamos desdoblar el ciclo tantas veces como sea necesario.

La figura 5.7 muestra la forma en que se desdobló el periodo y podemos observar con claridad en qué conjunto se produjo la falla.

El número de veces necesario que se debe desdoblar el periodo para asegurar la detección de un posible contraejemplo espurio es [34]

$$min = \min_{i \leq j \leq n} |h^{-1}(\widehat{s_j})|$$

El algoritmo *unwind* se encarga de desdoblar el periodo tantas veces como el mínimo número de estados en las preimágenes de los estados del periodo. Esto con el fin de asegurar que no se produzca una situación como la mostrada en la figura 5.6. Desdoblar el contraejemplo abstracto es simplemente copiar el periodo el número de veces establecido al final del contraejemplo abstracto obtenido de la función *buscaCiclo*.

Una vez obtenido el contraejemplo cumplidor podemos probar que dentro del

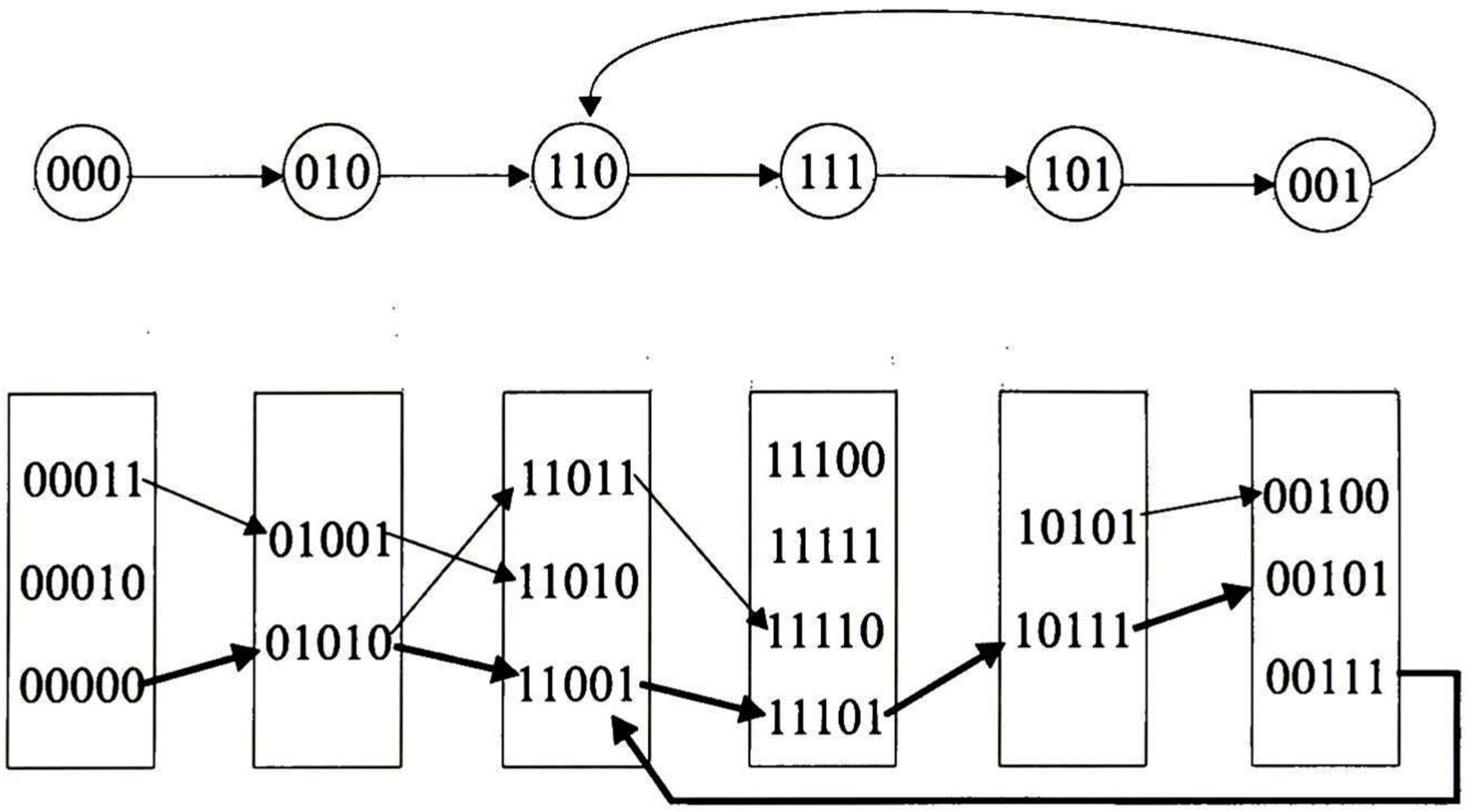


Figura 5.6: Contraejemplo abstracto que no es la imagen de un contraejemplo concreto.

periodo no se encuentra una falla antes de proceder a aplicar el algoritmo unwind. El algoritmo que se aplica para probar el periodo por primera vez es similar al aplicado para probar el prefijo abstracto. En esa misma prueba obtenemos el número del menor conjunto de las preimágenes de los estados del periodo. Ese número es el que nos sirve para determinar el número de veces que debemos desdoblar el periodo.

Si no se produjo ninguna falla, es decir el contraejemplo abstracto es la imagen de un contraejemplo concreto desde el prefijo hasta una vez el periodo, entonces procedemos a aplicar el algoritmo *unwind*. Al terminar el algoritmo *unwind* obtenemos una trayectoria que contiene tantos como *min+1* veces el periodo para descubrir un contraejemplo espurio. Sólo nos falta volver a aplicar la comprobación a la parte restante del periodo.

Cada vez que se hace una operación de intersección de conjuntos y el resultado no es vacío, se deben guardar los estados que están en la intersección. Al final obtenemos un conjunto de trayectorias concretas inducido por los estados en los conjuntos almacenados.

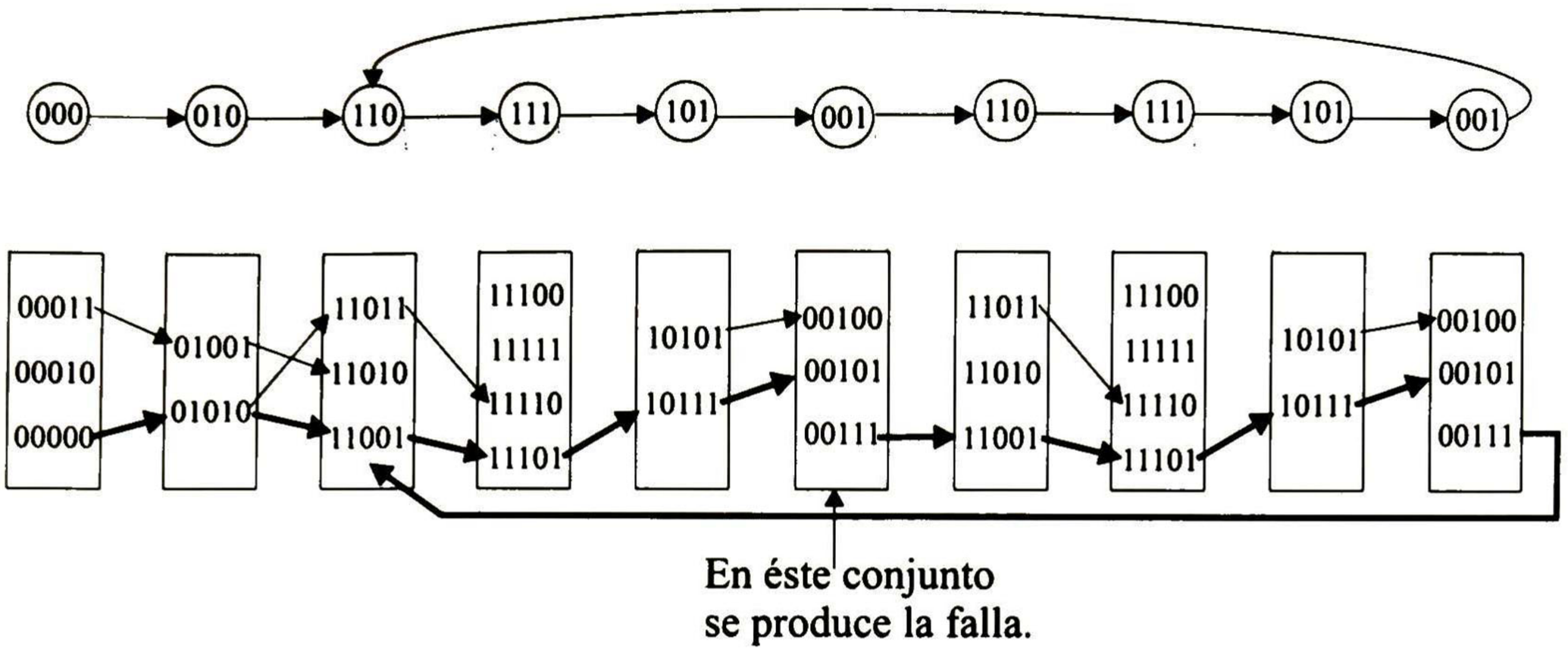


Figura 5.7: El contraejemplo abstracto ha sido desdoblado y ahora es evidente en donde se produce la falla.

5.4. Contraejemplo Válido en un ABGE

Si hemos pasado todas la pruebas de correspondencia de un contraejemplo abstracto a un contraejemplo concreto no podemos aún decir que dicho contraejemplo abstracto es válido, necesitamos ahora mostrar que el contraejemplo concreto obtenido cumple igualmente con las condiciones de marcado establecidas en el sistema. Como hemos mencionado, en realidad no obtenemos un contraejemplo como tal, si no que hemos obtenido una colección de conjuntos que inducen por lo menos un contraejemplo. El siguiente paso es encontrar un contraejemplo de los inducidos por los conjuntos de estados obtenidos en las intersecciones.

Lo que se hace ahora es tomar uno de los estados obtenidos en la última intersección y a partir de él caminar hacia atrás para formar la trayectoria, tomando en cada paso un estado de cada conjunto. La decisión sobre cual estado tomar queda determinada primeramente por el cumplimiento de las condiciones de marcado. Una vez satisfechas esas condiciones de marcado podemos tomar cualquier estado del conjunto actual.

El algoritmo que se muestra en la figura 5.8 nos permite elegir un contraejemplo concreto que cumple con las condiciones de marcado (condiciones de aceptación en

términos de ABGEs) determinadas en el sistema. Las entradas del algoritmo son los conjuntos de estados que obtenemos en cada intersección y que inducen por lo menos un contraejemplo concreto. La otra entrada es la colección de conjuntos de aceptación.

```
validarContraejemplo
1:  Entrada: ColeccionAceptacion PilaTr //conjuntos obtenidos en las intersecciones
2:  Salida: contraejemplo_valido
3:  estadoAnterior=PilaTr
4:  conjuntoActual=estadoAnterior->NumeroConjunto
5:  tope=conjuntoActual-Prefijo
6:  recorrido=0
7:  Mientras tope >0
8:      num=conjuntoActual
9:      Mientras anterior->Siguiente->NumeroConjunto=num
10:         anterior=anterior->Siguiente; //se adelanta el apuntador hasta el siguiente conjunto
11:     Si recorrido ≥ numestados
12:     Entonces
13:         addPreImagen(eligeEstado2(anterior,num))
14:     Si no
15:         eligeEstado2(anterior,num)
16:     recorrido++
17:     conjuntoActual=anterior->Siguiente->NumeroConjunto
18:     Si anterior
19:     Entonces
20:         anterior=anterior->Siguiente
21:     tope–
22: Si verificaColecciones()
23: Entonces
24:     Regresa contraejemplo_valido = verdadero
25: Si no
26:     Regresa contraejemplo_valido = falso
```

Figura 5.8: Algoritmo para saber si el contraejemplo concreto obtenido cumple con las condiciones de aceptación del sistema.

La función $eligeEstado2(estado, num)$ se encarga de elegir uno de los estados dentro del conjunto dado, siempre cuidando que se cumplan las condiciones de marcado. Cada colección que contiene alguno de los estados elegidos es marcada para no volver a elegir algún estado de ella. Al final la función $verificaColecciones()$ comprueba que todas las colecciones hayan sido tomadas en cuenta, en caso afirmativo el contraejemplo es válido, en caso contrario es espurio por no tener las condiciones de marcado

necesarias.

Una vez que se ha terminado la comprobación de las condiciones de aceptación y se han pasado satisfactoriamente, podemos asegurar que el contraejemplo concreto obtenido a partir de un contraejemplo abstracto es válido.

5.5. Refinamiento

El ciclo de verificación usando proyecciones se desarrolla de la manera siguiente:

1. Construimos $\mathcal{A} := proy(\mathcal{A}, Voc(\varphi))$ y determinamos si $\mathcal{A}_1 \models \varphi$.

2. Si la respuesta es afirmativa, entonces sabemos que $\mathcal{A}_1 \models \varphi$. De lo contrario, tomamos el contraejemplo abstracto que arroja la herramienta de verificación por comprobación de modelos y tratamos de ver si se trata o no de un contraejemplo espurio.

3. Si no tenemos un contraejemplo abstracto espurio, entonces sabemos que φ no se cumple en $\mathcal{A}$. De lo contrario, tomamos el subconjunto AP_2 de AP definido por $AP_2 := Voc(\varphi) \cup \{p_{i1}\}$, donde p_{i1} es una variable proposicional que no pertenece a $Voc(\varphi)$. Con este conjunto construimos la siguiente abstracción, $\mathcal{A}_2 := proy(\mathcal{A}, AP_2)$ y cerramos el ciclo volviendo al primer paso. A esta nueva abstracción le llamamos un *refinamiento de la abstracción* anterior.

Nuestra expectativa es encontrar finalmente un AP_m tal que $\mathcal{A}_m := proy(\mathcal{A}, AP_m)$ nos dé una respuesta concluyente (se cumpla φ en él, y por tanto también en $\mathcal{A}$, o que se encuentre un contraejemplo no espurio). Pero puede ocurrir que el procedimiento nos lleve finalmente de regreso a $\mathcal{A}$, y si este autómata es demasiado grande, tener que parar en alguna construcción intermedia debido a la explosión del espacio de estados.

En la figura 5.9 se muestra el ciclo de verificación utilizando abstracciones.

Con los resultados obtenidos en el capítulo 4 podemos ver que es posible hacer un refinamiento de una manera relativamente sencilla de un modelo abstracto. Únicamente debemos agregar variables de tal forma que tengamos un modelo abstracto mas

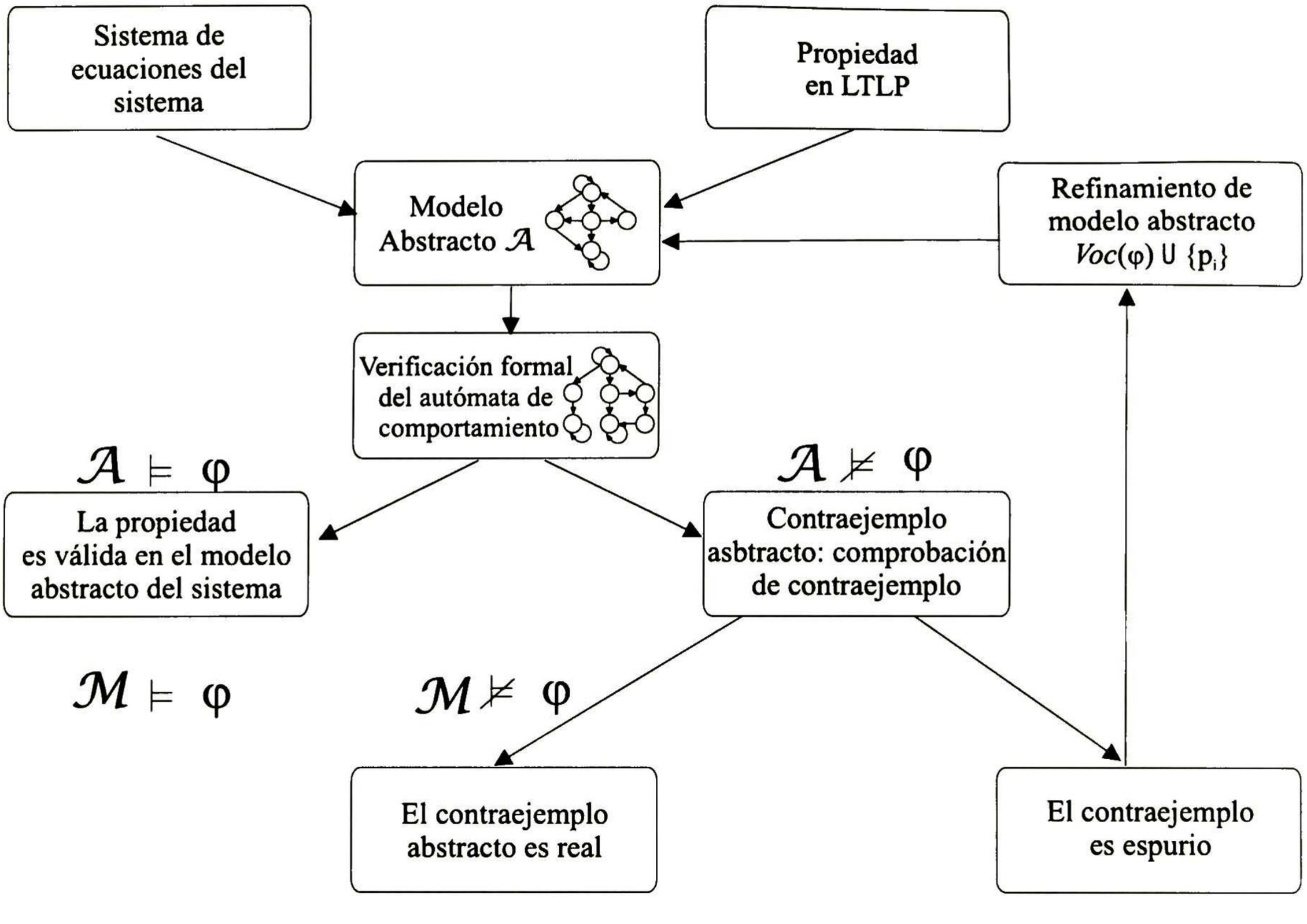


Figura 5.9: El proceso de verificación de un modelo abstracto.

grande que el anterior, pero que sea lo suficientemente pequeño para ser verificado, y que nos permita obtener un resultado concluyente acerca del contraejemplo obtenido.

La cuestión ahora es saber cuales variables debemos agregar para generar la siguiente abstracción. Una posible opción es utilizar el grafo de dependencia de variables.

Para integrar una nueva variable al sistema, en el archivo de la propiedad se debe especificar el conjunto de variables adicionales. Después de haber escrito la propiedad se escribirá el caracter @ (arroba) y después las variables deseadas separadas por comas y al final otro caracter @.

El algoritmo es básicamente el mismo, sólo que ahora deberá tomar en cuenta las variables adicionales. Estas variables adicionales estarán habilitadas para el cumplimiento de estados iniciales, marcados, prohibidos y precedencias. En la siguiente sección ejemplificaremos el uso de refinamientos para detectar contraejemplos reales.

5.6. Aplicación de la Detección de Contraejemplos Espurios y Refinamiento de Abstracciones

Utilizaremos en esta sección los resultados obtenidos en el capítulo 4 en la verificación de los modelos abstractos del sistema de las bombas alternantes.

Las propiedades que generaron contraejemplos se muestran en la tabla 5.1

#Prop.	Codificación en LTL	Res. Verif.	#Est.	#Trans.
3	$\Box((B1 \wedge \neg H \wedge \Diamond H) \rightarrow \bigcirc\bigcirc(B1 \wedge B2))$	No se cumple	8	42
4	$\Box((B2 \wedge \neg H \wedge \Diamond H) \rightarrow \bigcirc\bigcirc(B1 \wedge B2))$	No se cumple	8	42
5	$\Box((\neg B301 \wedge L) \rightarrow \bigcirc B1)$	No se cumple	8	54
6	$\Box((B301 \wedge L) \rightarrow \bigcirc B2)$	No se cumple	8	54
7	$\Box((L \vee \neg B1 \wedge \neg B2) \rightarrow \Diamond(B1 \vee B2))$	No se cumple	8	48
9	$\Box((B301 \wedge B302) \rightarrow \neg B302))$	No se cumple	4	10
10	$\Box((L \wedge \neg B1) \rightarrow \bigcirc B1)$	No se cumple	4	16
11	$\Box((L \wedge \neg B2) \rightarrow \bigcirc B2)$	No se cumple	4	16
12	$\Box((H \wedge B1 \wedge B2) \rightarrow \Diamond \neg H)$	No se cumple	8	42
13	$\Box((H \wedge B1 \wedge B2) \rightarrow \Diamond \neg LL)$	No se cumple	16	88
14	$\Box(OSR \rightarrow \bigcirc \neg OSR$	No se cumple	2	4

Tabla 5.1: Propiedades establecidas para el sistema de las bombas alternantes que no se cumplieron en los modelos abstractos.

La tabla 5.2 muestra los resultados de comprobar los contraejemplos abstractos obtenidos en el proceso de verificación. Podemos observar que un contraejemplo abstracto pudo corresponder directamente a un contraejemplo real, mientras que los demás necesitaron al menos un refinamiento.

El contraejemplo abstracto de la propiedad 14 pudo corresponder directamente a un contraejemplo real, el cual se muestra en la figura 5.10. La interpretación es: en la

#Prop.	Resultado Comprobación	#Refinamientos	#Est.	#Est. Anal	#Var
3	Propiedad Válida	2	15	23	5
4	Propiedad Válida	2	15	23	5
5	Propiedad Válida	2	16	18	5
6	Propiedad Válida	2	16	18	5
7	Propiedad Válida	1	12	13	4
9	Contraejemplo Válido	2	12	6	4
10	Contraejemplo Válido	1	6	19	3
11	Contraejemplo Válido	1	6	19	3
12	Propiedad Válida	2	17	50	6
13	Contraejemplo Válido	3	26	10	7
14	Contraejemplo Válido	0	2	4	2

Tabla 5.2: Resultado de la comprobación de los contraejemplos abstractos de las algunas propiedades.

primer línea se detalla el tipo de resultado obtenido, en este caso fue un contraejemplo válido. De la línea 2 a la 9 se muestra la secuencia de estados que pertenecen al contraejemplo, del lado izquierdo y derecho se encuentran los nombres de las variables y cada columna representa un estado del contraejemplo concreto. La parte del prefijo es la primer parte del contraejemplo concreto y la parte del periodo es el ciclo infinito de estados donde la propiedad no se cumple y se encierra entre | : : |.

Los estados analizados son 4, el periodo tiene un único estado que además es el último estado del prefijo.

5.6.1. Análisis de Contraejemplos Espurios

La propiedad 10 indica que siempre que se detecta el nivel bajo y la bomba 1 no está encendida, entonces en el siguiente instante se enciende la bomba 1. Sin problema podemos observar que la propiedad no debe cumplirse, ya que por la alternancia de la operación de las bombas puede ser que el turno de encendido sea de la bomba 2.

Analizaremos con más detalle el proceso de comprobación del contraejemplo de la propiedad 10, el cual se muestra a continuación:

$$00,\ 01,\ 01,\ (11,\ 00,\ 01,\ 10)$$

```
1:  Contraejemplo válido, estados analizados 4
2:  OSR    0  1  1  |: 1 :|  OSR
3:  B301   0  0  0  |: 0 :|  B301
4:  B302   0  0  0  |: 0 :|  B302
5:  B1     0  0  0  |: 0 :|  B1
6:  B2     0  0  0  |: 0 :|  B2
7:  L      0  0  0  |: 0 :|  L
8:  H      0  0  0  |: 0 :|  H
9:  LL     1  1  1  |: 1 :|  LL
```

Figura 5.10: Contraejemplo concreto obtenido de comprobar el contraejemplo abstracto de la propiedad 14.

El contraejemplo fue obtenido al verificar el modelo abstracto con la propiedad 10 en la herramienta VR. El primer paso en la comprobación del contraejemplo es tratar de hacer corresponder el prefijo abstracto en el modelo concreto.

El modelo abstracto del sistema con la propiedad 10 se muestra en la figura 5.11. Podemos seguir una secuencia en donde la propiedad no se cumple a partir del estado inicial, por ejemplo la secuencia del contraejemplo en donde se detecta el nivel bajo y en el siguiente estado se vuele a detectar el nivel bajo sin haberse encendido la bomba. Esa secuencia no necesariamente corresponde a una secuencia en el modelo concreto.

El resultado de la validación fue un contraejemplo espurio. En la figura 5.12 se muestra la forma en que la herramienta de construcción de modelos imprime el resultado.

La interpretación del resultado es la siguiente: en la primer línea se indica que el resultado es espurio, se da el número del estado abstracto en donde se produce la falla en la secuencia y la tupla de valores de las variables. En la línea 2 y 3 se indican los nombres de las variables y el valor que toman en el estado abstracto donde se produce la falla. En la línea 4 y 5 se muestra la secuencia de conjuntos de estados que sí tuvieron una correspondencia en el modelo concreto. El primer conjunto contiene los estados iniciales concretos que se proyectan en el estado inicial abstracto, en este caso hay dos estados iniciales concretos que se proyectan en el estado inicial abstracto. El segundo

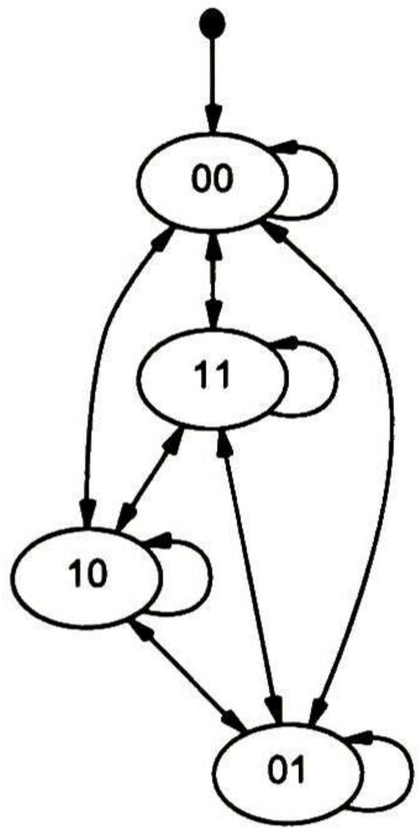


Figura 5.11: Modelo abstracto del sistema de las bombas y la propiedad 10. El orden de variables es (B1, L).

```
1:  El contraejemplo es espurio en el estado 3: 01
2:  B1  0
3:  L   1
-------------------------------------------------
4:  El computo concreto es:
5:  { 00000000, 00000001 } { 00000101 }
-------------------------------------------------
6:  El conjunto donde se encuentra el error es:
7:  OSR     0   0   0
8:  B301    0   0   0
9:  B302    0   0   0
10: B1      1   1   1
11: B2      0   0   0
12: L       1   1   0
13: H       1   0   0
14: LL      1   1   1
```

Figura 5.12: Resultado de la comprobación del contraejemplo de la propiedad 10 en el sistema de las bombas alternantes.

conjunto indica los estados concretos que se proyectan en el segundo estado abstracto, y así sucesivamente (si hubiera más conjuntos de estados). De la linea 6 a la línea 14 se muestra el conjunto de estados concretos donde se produce la falla. Ese conjunto de estados corresponde a los estados sucesores del conjunto de estados concretos que se proyectaron en el estado abstracto anterior al estado donde se produjo la falla, para este ejemplo el conjunto que se obtiene son los estados sucesores del segundo conjunto de la linea 5. Podemos observar que se necesitaba que el valor de la variable B1 fuera 0, pero todos los estados concretos del conjunto falla tienen en la variable B1 el valor 1. Allí es donde se produce la falla, ya que no hay estados concretos que puedan continuar con la correspondencia del contraejemplo abstracto en el concreto.

Lo que sigue es efectuar un refinamiento del modelo abstracto. Para realizar el refinamiento del modelo abstracto necesitamos agregar variables al modelo, en este caso las variables que están relacionadas directamente con B1 son: LL, B2 y B302. El refinamiento lo hicimos por partes, agregando una variable a la vez.

Al agregar la variable LL obtuvimos un nuevo modelo abstracto, la verificación de ese modelo abstracto nuevamente produjo un contraejemplo.

El modelo abstracto para el refinamiento se muestra en la figura 5.13. Podemos observar que el modelo no es mucho más grande que el anterior pero contiene información más detallada acerca del funcionamiento del sistema.

El contraejemplo producido se muestra a continuación:

000, 001, 011, 001, (000, 111, 101, 100, 011, 001)

De la misma forma que anteriormente podemos recorrer el contraejemplo: iniciamos en el estado en donde todas las variables son 0, en el siguiente estado se detecta el nivel bajo bajo, posteriormente en el estado siguiente se detecta el nivel bajo y según lo indicado por la propiedad 10, debería encender la bomba 1 en el siguiente estado, pero la bomba 1 aún no se ha encendido en ese estado. Allí es donde no se cumple la propiedad y los siguientes estados muestran el periodo donde ya no importa la secuencia por que la propiedad se no se ha cumplido desde el prefijo.

Este contraejemplo abstracto sí corresponde a un contraejemplo en el modelo concreto y se muestra en la figura 5.14.

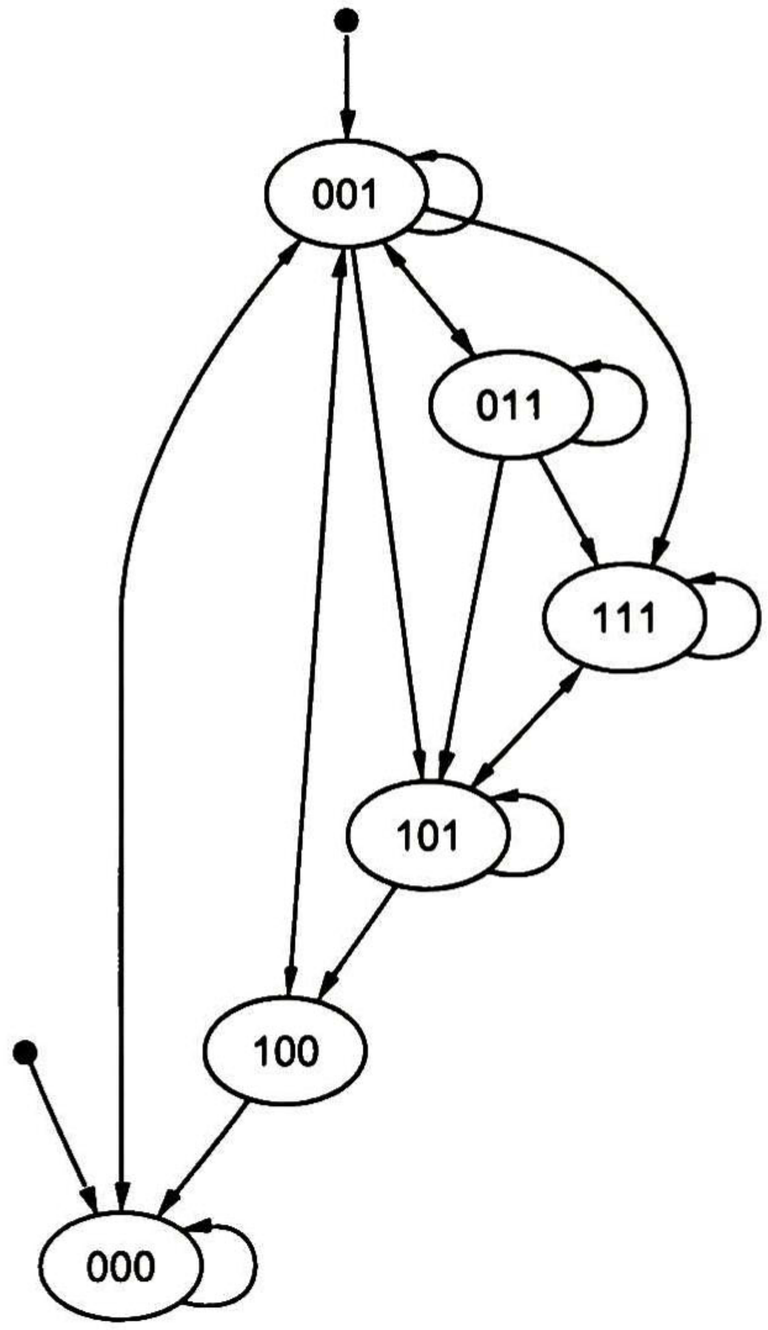


Figura 5.13: Refinamiento del modelo abstracto del sistema de las bombas y la propiedad 10. El orden de variables es (B1, L, LL,).

```
1:  Contraejemplo válido, estados analizados 5
2:  OSR    0  1  0  0  |: 0 :|  OSR
3:  B301   0  0  1  0  |: 0 :|  B301
4:  B302   0  0  1  1  |: 1 :|  B302
5:  B1     0  0  0  0  |: 0 :|  B1
6:  B2     0  0  0  1  |: 1 :|  B2
7:  L      0  0  1  0  |: 0 :|  L
8:  H      0  0  0  0  |: 0 :|  H
9:  LL     0  1  1  1  |: 1 :|  LL
```

Figura 5.14: Contraejemplo concreto obtenido de comprobar el contraejemplo abstracto de la propiedad 10.

Ahora ya sabemos que el contraejemplo abstracto efectivamente corresponde a un contraejemplo concreto. Con el uso del refinamiento podemos obtener un resultado concluyente acerca de contraejemplos abstractos obtenidos en el proceso de verificación de modelos abstractos.

5.7 Conclusiones

La detección de contraejemplos espurios nos es útil para saber si la propiedad verificada en un modelo abstracto es válida o no. Si tenemos un contraejemplo espurio podemos intentar hacer un refinamiento al modelo abstracto y correr nuevamente la verificación con el nuevo modelo abstracto. Otra solución podría ser intentar buscar otro ciclo de tal manera que podamos obtener nuevas trayectorias abstractas y con ellas realizar la comprobación del contraejemplo espurio. Esta opción puede ser útil, pero nos podría llevar mucho tiempo probar muchos ciclos y aún así no obtener un resultado satisfactorio. Por otra parte podríamos explorar un refinamiento más complejo tratando de refinar únicamente el estado abstracto donde se encontró la falla. La ventaja de aplicar el refinamiento en la forma en que lo hacemos es que tenemos la seguridad de que la nueva proyección cumple los teoremas establecidos en el capítulo 4.

Capítulo 6

Conclusiones

6.1. Contenido del Capítulo

En este capítulo se expresan las conclusiones a las que llegamos en la realización de este trabajo y se da una lista de algunos problemas que pueden conducir a líneas de trabajo e investigación.

6.2. Conclusiones

La verificación formal por comprobación de modelos es una técnica relativamente nueva que ha ganado aceptación en el ámbito industrial. Para poder manipular sistemas grandes es necesario contar con técnicas que nos permitan trabajar eficientemente la información que representa el modelo del sistema. La comprobación de modelos mediante el cálculo explícito tiene la ventaja de trabajar con los estados y sus transiciones directamente, pero tiene la desventaja de que muy pronto se produce el problema de explosión de estados. Una forma de tratar con este problema es a través de técnicas de reducción del espacio de estados. Para que funcionen de manera óptima éstas técnicas se debe trabajar con ellas desde la construcción del modelo en forma incremental, y no tratar de reducir el modelo una vez que este se ha construido por completo. Las operaciones necesarias en la construcción de un modelo reducido pueden ser muy costosas en cuanto a tiempo de cálculo se refiere. Esas operaciones

las hemos implementado con técnicas de cálculo simbólico.

En la construcción de un modelo del sistema podemos aprovechar el conocimiento que tengamos de este para tratar de que el modelo contenga únicamente los comportamientos posibles, ya sea a través de una correcta definición de los estados iniciales, precedencias de variables, estados prohibidos ó construyendo el modelo tomando en cuenta una propiedad a verificar. Así mismo, teniendo un amplio conocimiento del sistema podemos elaborar especificaciones más precisas, de tal manera que, cuando encontremos un contraejemplo en el proceso de verificación podamos interpretarlo de manera correcta y saber exactamente si el problema fue causado por el diseño o por una especificación incorrecta.

La combinación de técnicas de cálculo simbólico y explícito es una opción viable para construir modelos explícitos reducidos de un sistema de una manera incremental.

En el desarrollo de este trabajo se lograron todas las metas establecidas y podemos señalar los siguientes resultados:

- Desarrollo de un método de modelado explícito a partir de la traducción de la de las ecuaciones de estado siguiente que representan un controlador de un sistema dado en lenguaje de escalera.

- Diseño y desarrollo de los algoritmos de modelado en una herramienta computacional. Los algoritmos hacen una construcción incremental a partir de los estados iniciales, evitando el análisis de alcanzabilidad y recortes del modelo.

- Modelado del sistema como un ABGE para permitir la compatibilidad con las herramientas en [43] y con [11].

- Incorporación de técnicas que permiten aprovechar el conocimiento del sistema para la generación de un modelo del mismo, de tal manera que el modelo generado sea fiel a la realidad. De forma indirecta se producen modelos más pequeños que el original. Estas técnicas son: 1)definición de los estados iniciales del sistema, 2)definición de precedencias de variables y 3)definición de estados prohibidos del sistema.

- Incorporación de una técnica de reducción del espacio de estados a la herramienta de modelado.

- Formalización de la técnica de reducción del espacio de estados implementada, llamada **Simulación por Abstracción de Variables.**

- Incorporación de análisis de contraejemplos obtenidos en la verificación modelos abstractos del sistema en la herramienta de modelado. Incluye la detección de contraejemplos espurios y el mapeo de contraejemplos abstractos en contraejemplos reales.

- Incorporación de refinamientos sobre modelos abstractos para su posterior verificación.

- Exploración de la combinación de técnicas de cálculo simbólicas con cálculo explícito.

- Diseño y construcción de una Interfaz gráfica que facilite el proceso de verificación por comprobación de modelos.

- Incorporación en la interfaz gráfica las herramientas desarrolladas en [43]y [11].

- Incorporación en la interfaz gráfica de la herramienta de modelado desarrollada en esta tesis.

- Se escribió una ponencia para AMIDIQ'03

6.3. Trabajo futuro

A través del desarrollo de este trabajo pudimos observar algunos puntos en donde se puede seguir investigando para incrementar la eficiencia de la herramienta y el método de modelado propuesto. Algunas de las mejoras incluyen:

- Relajar las hipótesis establecidas en la generación del modelo.

- Incorporación de más técnicas de reducción del espacio de estados.

simulación por proyección de variables.

3. Corroborar que un contraejemplo obtenido en la verificación de un modelo abstracto corresponde a un contraejemplo en el modelo concreto del sistema.

4. Realizar un refinamiento sobre un modelo abstracto.

Para cada una de las funciones de la herramienta se necesita un archivo especial: el archivo para la construcción del modelo, el archivo para la fórmula que representa la propiedad mediante la cual se hace la simulación por proyección de variables y el archivo del contraejemplo. Para el refinamiento se usa el archivo de la propiedad junto con las variables adicionales agrupadas entre caracteres @. Debe tomarse en cuenta que la información contenida en los archivos de entrada es sensible a las mayúsculas y minúsculas.

Las palabras reservadas para los archivos y los símbolos utilizados en cada uno de los archivos se muestran en la tabla A.1.

Cualquier símbolo diferente se tomará como carácter desconocido.

A.2.1. Generación del Modelo del Sistema

Para realizar la primer función, la herramienta de modelado necesita la descripción del sistema (dada por las ecuaciones lógicas de estado siguiente que corresponden a los escalones del diagrama de escalera) y opcionalmente conocimiento adicional del funcionamiento del sistema (estados iniciales, marcados, prohibidos y relaciones de precedencia). Esta información es detallada en un archivo que, por convención, llevará la extensión *.ldd*, el archivo es conocido entonces como *archivo del sistema.*

Las secciones del archivo del sistema son:

- **Ecuaciones.**

  En esta sección se especifican las ecuaciones de estado siguiente que representan la lógica de escalera del controlador. Ésta sección es indispensable para generar el modelo, ya que define la relación de transición entre los estados. La sección comienza con la palabra reservada *ecuaciones* y entre llaves se escribe cada una

Palabras reservadas	Función
ecuaciones	Define la sección ecuaciones
iniciales	Define la sección estados iniciales
marcados	Define la sección estados marcados
prohibidos	Define la sección estados prohibidos
precedencia	Define la sección precedencias
ALLSTATES	Todos los estados son marcados
prec	Define una regla de precedencia débil
precfuerte	Define una regla de precedencia fuerte
ltlp	Determina que el archivo es de una propiedad
contraejemplo	Determina que el archivo es de un el contrajemplo
Delimitadores	Nombre del delimitador
{, }	Llave izquierda y derecha
, ;	Coma, punto y coma
=, :=	Asignación
(,)	Paréntesis izquierdo y derecho
1, 0	Constantes boolenas
$[0-1][0-1]*$	Estados del contraejemplo
$[a-zA-Z][a-zA-Z0-9]*$	Variables
$--$, //	Comentarios
@	Arroba

Tabla A.1: Palabras reservadas y símbolos de un archivo de entrada

de las ecuaciones de estado siguiente (cada una representa un escalón del diagrama de escalera). Las ecuaciones se escriben como se definieron en el capítulo 3.

- **Estados Iniciales**.
  En esta sección se define el ó los estados iniciales del sistema. Esta parte es opcional y representa las condiciones iniciales de las variables del sistema. La sección comienza con la palabra reservada *iniciales* y entre llaves se especifican las variables a las cuáles se les asigna un valor inicial.

- **Estados Marcados**.
  En esta sección se definen los estados marcados del sistema, los cuales corresponden a los estados de aceptación del ABGE generado. En términos físicos

ésta sección corresponde a aquéllos estados en donde se quiere resaltar el hecho de haber realizado una tarea o donde se ha alcanzado un punto clave del funcionamiento sistema. La sección comienza con la palabra reservada *marcados* y entre llaves se especifica la colección de conjuntos que representa los estados marcados. Cada conjunto es agrupado también por medio de llaves y cada conjunto se define por un grupo de variables con los valores que corresponden a los estados marcados para ese conjunto.

- **Estados Prohibidos**.

  En esta sección se definen los estados prohibidos del sistema, los cuales representan situaciones que el sistema nunca debe alcanzar ya sea por cuestiones de seguridad o por la misma naturaleza de la operación del sistema. La sección comienza con la palabra reservada *prohibidos* y entre llaves se especifica la colección de conjuntos que representan los estados prohibidos. Cada conjunto es agrupado a su vez por medio de llaves y cada conjunto se define por un grupo de variables con los valores que corresponden a los estados prohibidos para ese conjunto.

- **Precedencias**.

  En esta sección se define una parte importante del comportamiento del sistema, la cual es relativa a la forma en que los valores de las variables pueden evolucionar. En términos físicos significa que un evento debe o no suceder antes que otro. La sección comienza con la palabra reservada *precedencias* y entre llaves se especifican las tuplas de variables que representan las relaciones de precedencias.

Un ejemplo de un archivo que contiene todas las secciones se muestra en la figura A.1.

La forma en que opera la herramienta para generar el modelo del sistema es la siguiente:

1. Se realiza la lectura del archivo de entrada mediante un analizador léxico y semántico implementado en Flex y Bison [33].

ecuaciones
{
$B301 = LL \wedge OSR;$
$OSR = \neg LL;$
$B302 = (B302 \vee B301) \wedge (\neg B302 \vee \neg B301);$
$B1 = LL \wedge ((\neg B302 \wedge L) \vee B1 \vee (B2 \wedge (H \vee B1)));$
$B2 = LL \wedge ((B302 \wedge L) \vee B2 \vee (B1 \wedge (H \vee B2)));$
}

iniciales
{
$B301 = 0;$
$OSR = 0;$
$B302 = 0;$
$B1 = 0;$
$B2 = 0;$
$H = 0;$
$L = 0;$
}

marcados
{
{B301=1,B1=1, B2=0},
{B1=1,B2=1}
{

prohibidos
{
{B301=1, B302=1, B1=1, B2=1,},
{B1=0,B2=0,OSR=1}
{

precedencias
{
$LL = 1$ precfuerte $L = 1;$
$L = 1$ precfuerte $H = 1;$

$H = 0$ prec $L = 0;$
$L = 0$ prec $LL = 0;$
}

Figura A.1: Ejemplo de una archivo de sistema con las secciones que soporta el método de modelado.

2. Se crean las siguientes listas: variables del sistema, variables iniciales, estados marcados, estados prohibidos y precedencias.

3. Se escribe en el archivo de salida la lista de variables del sistema.

4. Se definen los estados iniciales del sistema y se escriben en el archivo de salida. Tomando en consideración que los estados no sean prohibidos.

5. Se calculan los estados sucesores de los estados iniciales y se guardan en una lista. Para calcular los sucesores sólo se toman en cuenta los estados que cumplen con las reglas de precedencia y de estados prohibidos. También se guardan en una lista los estados que cumplen con las condiciones de marcado.

6. Se calculan los estados sucesores de los estados que aparecen en la lista del paso anterior, siguiendo las mismas consideraciones de precedencias, prohibidos y marcados. Cada estado nuevo se guarda en la lista para posteriormente calcular sus sucesores. Se escribe el estado actual y sus sucesores en el archivo de salida.

7. Se escriben en el archivo de salida los estados que aparecieron como marcados.

Una vez que se han efectuado éstas operaciones obtenemos el ABGE que representa el modelo del sistema. El archivo de salida tiene la sintaxis definida en [43].

A.2.2. Generación del Modelo Abstracto

Para generar el modelo abstracto del sistema necesitamos dos archivos de entrada: el archivo del sistema y el archivo que contenga la propiedad con la cual generaremos el modelo abstracto.

Las propiedades son expresadas mediante la LTLP y cada propiedad es guardada en un archivo que por convención tiene la extensión .ltl. Las propiedades tienen la misma sintaxis que se presentó en la sección *Lógica temporal lineal proposicional* del capítulo 2. En la tabla A.2 se muestra la lista de símbolos permitidos en el archivo de propiedad.

Símbolos de la LTLP	Símbolos de un archivo de fórmula	
$\wedge$	/\	(Diagonal y diagonal invertida)
$\vee$	\/	(Diagonal invertida y diagonal)
$\neg$	~	(Tilde)
$\rightarrow$	->	(Guión y mayor que)
$\leftrightarrow$	<->	(menor que, guión y mayor que)
$\Box$	G	(Letra mayúscula "G")
$\Diamond$	F	(Letra mayúscula "F")
$\bigcirc$	X	(Letra mayúscula "X")
$\mathcal{U}$	U	(Letra mayúscula "U")
$\mathcal{V}$	V	(Letra mayúscula "V")
$\top$	true	
$\bot$	false	

Tabla A.2: Símbolos permitidos en un archivo de propiedad.

Los nombres de las variables proposicionales deben ser diferentes a los símbolos mostrados en la tabla A.2 ya que se tomarían como operadores y no como variables proposicionales en la etapa de verificación.

Una vez ingresados los archivos de entrada se efectúan las siguientes operaciones para generar el modelo abstracto del sistema:

1. Lectura del archivo de la propiedad.

2. Se crea la lista de variables contenidas en la propiedad.

3. Se realiza la lectura del archivo del sistema y se crean las listas mencionadas en la sección anterior.

4. Se comprueba que las variables de la propiedad sean un subconjunto de las variables del sistema.

5. Se escribe en el archivo de salida la lista de variables del sistema.

6. Se forma el sistema de ecuaciones lógicas para construir el BDD del sistema.

7. Se calculan los estados iniciales abstractos y se escriben en el archivo de salida. Tomando en consideración que los estados no sean prohibidos.

8. Se calculan los estados sucesores abstractos de los estados iniciales abstractos y se guardan en una lista aquellos que no han sido generados. Se escriben los estados generados en el archivo de salida. Solo se toman en cuenta los estados abstractos que cumplen con las reglas de precedencia y de estados prohibidos. También se guardan en una lista los estados abstractos que cumplen con las condiciones de marcado.

9. Se calculan los estados sucesores abstractos de los estados que aparecen en la lista del paso anterior, siguiendo las mismas consideraciones de marcados, prohibidos y precedencias. Se escriben en el archivo de salida los estados evaluados con sus respectivas transiciones.

10. Se escriben en el archivo de salida los estados abstractos que aparecieron como marcados.

Al terminar estas operaciones se obtiene un ABGE que representa el modelo abstracto del sistema tomando en cuenta una propiedad a verificar. Con la incorporación de los BDDs se logró disminuir considerablemente el tiempo de cálculo de los estados sucesores abstractos.

El archivo de salida tiene la sintaxis que se define en [43].

A.2.3. Comprobación del Contraejemplo

La herramienta VR descrita en [43] nos puede dar como resultado en el proceso de verificación un contraejemplo. Si el contraejemplo dado proviene de un modelo abstracto, debemos corroborar que el contraejemplo abstracto puede ser mapeado a un contraejemplo real.

Para realizar la comprobación del contraejemplo abstracto necesitamos tres archivos: el archivo del sistema, el archivo de la propiedad verificada y el archivo con el contraejemplo obtenido.

La forma en que funciona la herramienta para corroborar que el contraejemplo abstracto puede ser mapeado a un contraejemplo real es la siguiente:

1. Se realiza la lectura del archivo de la propiedad.

2. Se crea la lista de variables de la propiedad.

3. Se realiza la lectura del archivo del sistema.

4. Se crean las listas con la información del sistema.

5. Se comprueba que las variables de la propiedad son subconjunto de las variables del sistema.

6. Se realiza la lectura del archivo del contraejemplo. Se crea la lista con los estados del prefijo y del periodo.

7. Se comprueba que el número variables de la propiedad sean las mismas que las del contraejemplo.

8. Se crea un ciclo abstracto de la parte del periodo.

9. Se comprueba que el prefijo corresponda a un prefijo concreto. Y se genera el prefijo concreto.

10. Se comprueba que el periodo corresponda a un periodo concreto. Y se genera el periodo concreto.

11. Se corrobora que se cumplan las condiciones de marcado en el contraejemplo concreto que fue obtenido.

El resultado de las operaciones realizadas nos determina si el contraejemplo abstracto obtenido pudo ser mapeado a un contraejemplo real, en caso de que no haya sido así se muestra el estado donde ocurrió la falla.

A.2.4. Refinamiento

Para realizar el refinamiento necesitamos el archivo del sistema y el archivo de la propiedad. En el archivo de la propiedad vamos a escribir entre caracteres @ la lista de variables que van a formar parte del refinamiento del modelo abstracto. Cada variable va ir separada por comas. Cada variable que se adicione en el refinamiento es tomada en cuenta para las secciones de estados iniciales, estados marcados, estados

prohibidos y las reglas de precedencia. El algoritmo para construir el refinamiento es el mismo que para construir el modelo abstracto.

A.3. Estructuras de Datos

Se utilizaron 15 listas para contener la información referente a todas las funciones que realiza la herramienta.

Para las 2 primeras funciones es necesario guardar las listas de variables del sistema, estados iniciales, estados marcados, precedencias, estados prohibidos y la lista de variables de la propiedad. Para la tercer función es necesario calcular las imágenes delanteras de los estados concretos a ser mapeados y es necesario guardar el contraejemplo concreto que se va construyendo paso a paso.

Las estructuras las vamos a mostrar de acuerdo al orden en que aparecen en las funciones de la herramienta.

La primer estructura, mostrada en la figura A.2 define en donde se van a guardar las siguientes listas de variables: del sistema, de estados iniciales, de la propiedad a probar y de las variables que aparecen en el sistema pero no en la formula.

```
struct nodo
{
        char* Nombre
        int Valor
        int Tipo
        struct nodo* Siguiente
}*Lista, *VarEstado, *Iniciales, *VarFormula, *VarFormulaDif
```

Figura A.2: Estructura de datos *nodo* se encarga del almacenamiento de las variables que intervienen en la herramienta de modelado.

Los campos de la estructura guardan el nombre de la variable(Nombre), el valor de la variable(Valor) y el tipo de la variable(Tipo, por ejemplo: de estado, inicial, libre).

La siguiente estructura, mostrada en la figura A.3 define en donde se van a guardar los estados que ya se han generado y el ciclo que se forma en el contraejemplo.

```
struct estado
{
        char* Nombre
        int Numero
        int Existe
        int Inicial
        struct transiciones* transi
        struct estado* Siguiente
}*Buchi, *Ciclo
```

Figura A.3: La estructura de datos *estado* se encarga del almacenamiento de los estados que se han generado y del ciclo del contraejemplo.

Los campos de la estructura guardan el nombre del estado(Nombre), el número de estado(Número), si el estado ya ha sido tomado en cuenta al generar el ABGE (Existe), si pertenece a los iniciales ó si ya fue considerado en el ciclo(Inicial) y las transiciones a otros estados(Transi). La estructura *transiciones* únicamente contiene el nombre de los estados sucesores del estado actual.

La siguiente estructura, mostrada en la figura A.4 define en donde se guardan las variables que definen la colección de estados marcados concretos y abstractos y los estados del prefijo y el periodo.

```
struct conjunto
{
        char* Nombre
        int NumeroConjunto
        int Valor
        struct conjunto* Siguiente
}*Aceptación, *MarcadoAbs, *PilaTr
```

Figura A.4: Estructura de datos *conjunto* se encarga del almacenamiento de los estados marcados, los estados del prefijo y el periodo del contraejemplo.

Los campos de la estructura guardan el nombre de la variable que define al estado marcado y el nombre del estado del prefijo y periodo(Nombre), el numero de conjunto al que pertenece(NumeroConjunto) y el valor de la variable(Valor).

La siguiente estructura, mostrada en la figura A.5 define en donde se van a guardar las variables que definen a un estado prohibido.

```
struct pconjunto
{
        char* Nombre
        int NumeroConjunto
        int Valor
        struct pconjunto* Siguiente
}*Prohibidos
```

Figura A.5: Estructura de datos *pconjunto* se encarga del almacenamiento de las variables que definen un estado prohibido.

Los campos de la estructura guardan el nombre de la variable(Nombre), el número de conjunto al que pertenece la variable(NúmeroConjunto) y el valor de la variable(Valor).

La siguiente estructura, mostrada en la figura A.6 define en donde se van a guardar las variables que definen las reglas de precedencia.

```
struct tuplasprec
{
        char* Nombre1
        char* Nombre2
        int Valor1
        int Valor2
        int NumeroConjunto
        struct tuplasprec* Siguiente
}*ReglasPrecedencia
```

Figura A.6: Estructura de datos *tuplasprec* se encarga del almacenamiento de las variables que forman la tuplas de precedencias.

Los campos de la estructura guardan el nombre de la variable que precede en la relación(Nombre1), el nombre de la variable precedida en la relación(Nombre2), el valor de la variables (Valor1 y Valor2) y el número de precedencia al que pertenecen (NumeroConjunto).

La última estructura, mostrada en la figura A.7 define en donde se van a guardar los estados que se calculan en cada paso del algoritmo para detectar si un contra-ejemplo es espurio.

```
struct estadoc
{
        char* Nombre
        int Tipo
        struct estadoc Siguiente
}*cPreImagen, *cImagenDelantera, *cS
```

Figura A.7: La estructura de datos *estadoc* se encarga del almacenamiento de los estados de los conjuntos preimagen, imagendelantera y cS, utilizadas en la detección del contraejemplo.

Los campos de la estructura guardan el nombre del estado(Nombre) y el tipo del estado: en caso de que el estado que se evaluó pertenezca al prefijo o al periodo.

Esas fueron las estructuras de datos definidas y utilizadas en la herramienta desarrollada.

Para la construcción de modelo abstracto se hizo uso del paquete **Buddy** para BDDs, pero no se especificaron estructuras nuevas, sino que se utilizaron las estructuras definidas en Buddy. Las funciones que se utilizaron fueron la: existencial, allsat y apply para los diferentes conectivos lógicos.

A.4. Requerimientos, Compilación y Ejecución

La herramienta fue desarrollada en ANSI C en el sistema operativo Linux Mandrake 9. La herramienta ha sido probada en Linux: Mandrake 9, Red Hat 9 y Mandrake 9.2.

En la tabla A.3 se muestra la lista de archivos fuente de la herramienta.

ldd.c
estructuras.c
estructuras.h
abstraccion.c
contraejemplo.c
sistema.y
sistema.l
formulabdd.y
formulabdd.l
ltl.y
ltl.l
contraejemplo.y
contraejemplo.l
make

Tabla A.3: Archivos requeridos para compilar la herramienta

Para compilar la herramienta se requiere tener instalado el software que se presenta en la tabla A.4.

Linux
gcc 2.96-98
flex 2.5.4a-15
bison 1.9-19
buddy 2.2

Tabla A.4: Software requerido para la compilación

La compilación en Linux se realiza tecleando, en el mismo directorio donde se encuentran los archivos fuente, los siguientes comandos:

```
$ make linux

$ make all
```

El primer comando se encarga de la compilación de los archivos para las herramientas flex y bison, el segundo comando se encarga de compilar todos los archivos de código C.

Cuando el archivo binario se ha creado, éste pueden copiarse al directorio que se desee y ejecutarse como se indica adelante.

```
$ ./ldd nombre_archivosistema nombre_archivopropiedad
nombre_archivocontrajemplo
```

La salida después de ejecutar la herramienta será en diversos archivos, según haya sido la función que se haya invocado. Para la creación del modelos se crea un archivo con el mismo nombre del archivo del sistema pero con la extensión *fsm*. Para los modelos abstractos se crea un archivo con el nombre del archivo del sistema seguido de un guión bajo, el nombre del archivo de la propiedad y la extensión fsm. En el caso de los contraejemplos la salida es a pantalla y a un archivo llamado *resultado.res*.

En el apéndice 2 se muestra el diseño e implementación de una interfaz gráfica para un manejo más cómodo de la herramienta de modelado y de las herramientas de verificación.

Apéndice B

Manual de la Interfaz.

B.1. Contenido del Apéndice

En éste apéndice se detalla la estructura de la interfaz gráfica de verificación, la cual engloba las partes de modelado del sistema, verificación formal y generación de especificaciones. En la sección B.2, se hace una descripción de la herramienta, como está estructurada y la manera general en que trabaja. En la sección B.3 se explica la funcionalidad para la administración de los archivos de la herramienta. En la sección B.4 se especifican las funciones de la interfaz en el manejo de la edición de un archivo. En la sección B.5 se muestra la función para generar modelos a partir de la descripción de un sistema dado en lenguaje de escalera. En la sección B.6 se especifican las funciones para verificar un sistema descrito como un ABGE y para manipular las propiedades de dicho sistema descritas en LTLP. En la sección B.7 se detallan las funciones para generar un archivo en formato Postscript del modelo del sistema. En la sección B.8 se muestran las funciones para desplazarse entre las diversas ventanas de edición. Por último en la sección B.9 se presentan los elementos necesarios para realizar la instalación de la herramienta y la forma en que se instala.

B.2. Descripción General

Uno de los objetivos de ésta tesis fue incorporar el ciclo de verificación en una interfaz. Para lograr este objetivo se elaboró una interfaz gráfica desarrollada en Linux Mandrake 9 implementada en Kylix 3 de Borland. La interfaz contiene las funciones de la herramienta de Modelado descrita en éste trabajo, las funciones de la herramienta de verificación formal descrita en [43] y las funciones de la herramienta para convertir una fórmula de LTLP a un ABGE descrita en [11]. La Interfaz es también un editor de textos convencional el cual sirve para crear, visualizar y modificar los archivos generados para las diversas funciones de la interfaz.

La interfaz gráfica se muestra en la figura B.1 la cual está dividida en 9 partes:

1. Manejo de Archivos.
2. Edición de archivos.
3. Construcción de modelos a partir de la descripción del sistema por medio del lenguaje de escalera.
4. Generación del modelo de una propiedad.
5. Verificación formal de un modelo.
6. Generación de un archivo en formato postscript del modelo del sistema, la propiedad o del autómata de comportamiento.
7. Administración de ventanas de la interfaz.
8. Barra de estado.
9. Area de trabajo

En la parte de *barra de estado* se muestra el nombre del archivo del sistema cargado en memoria y el nombre de la propiedad cargada en memoria. En la parte referente al *área de trabajo* se agrupan las ventanas que contienen los archivos que actualmente se encuentran abiertos, estas ventanas se manejan como ventanas hijas. El número de ventanas que se pueden abrir está restringida por la cantidad de memoria de la

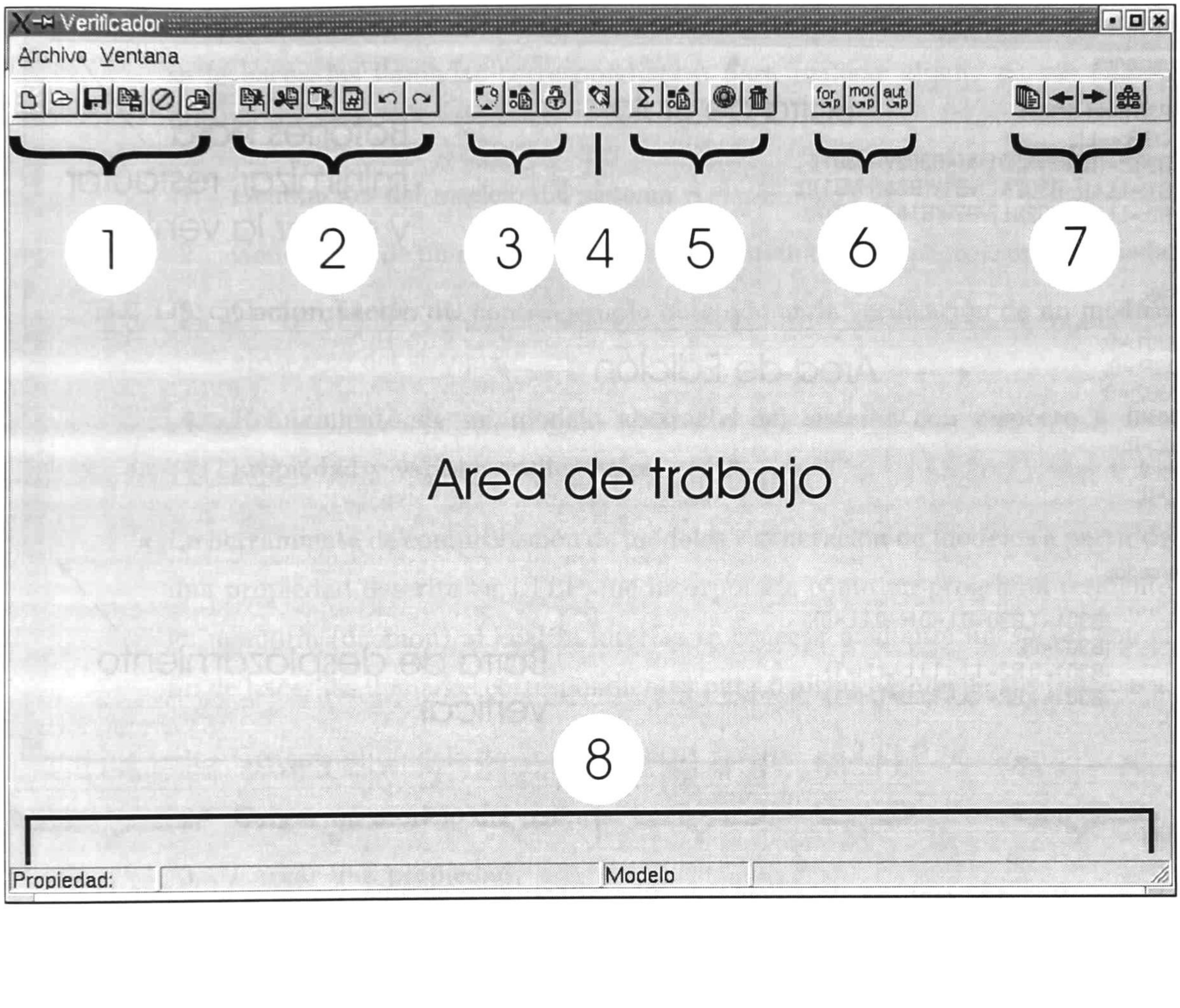


Figura B.1: Interfaz gráfica desarrollada para englobar las herramientas de verificación formal y modelado desarrolladas por el grupo de verificación formal del Cinvestav.

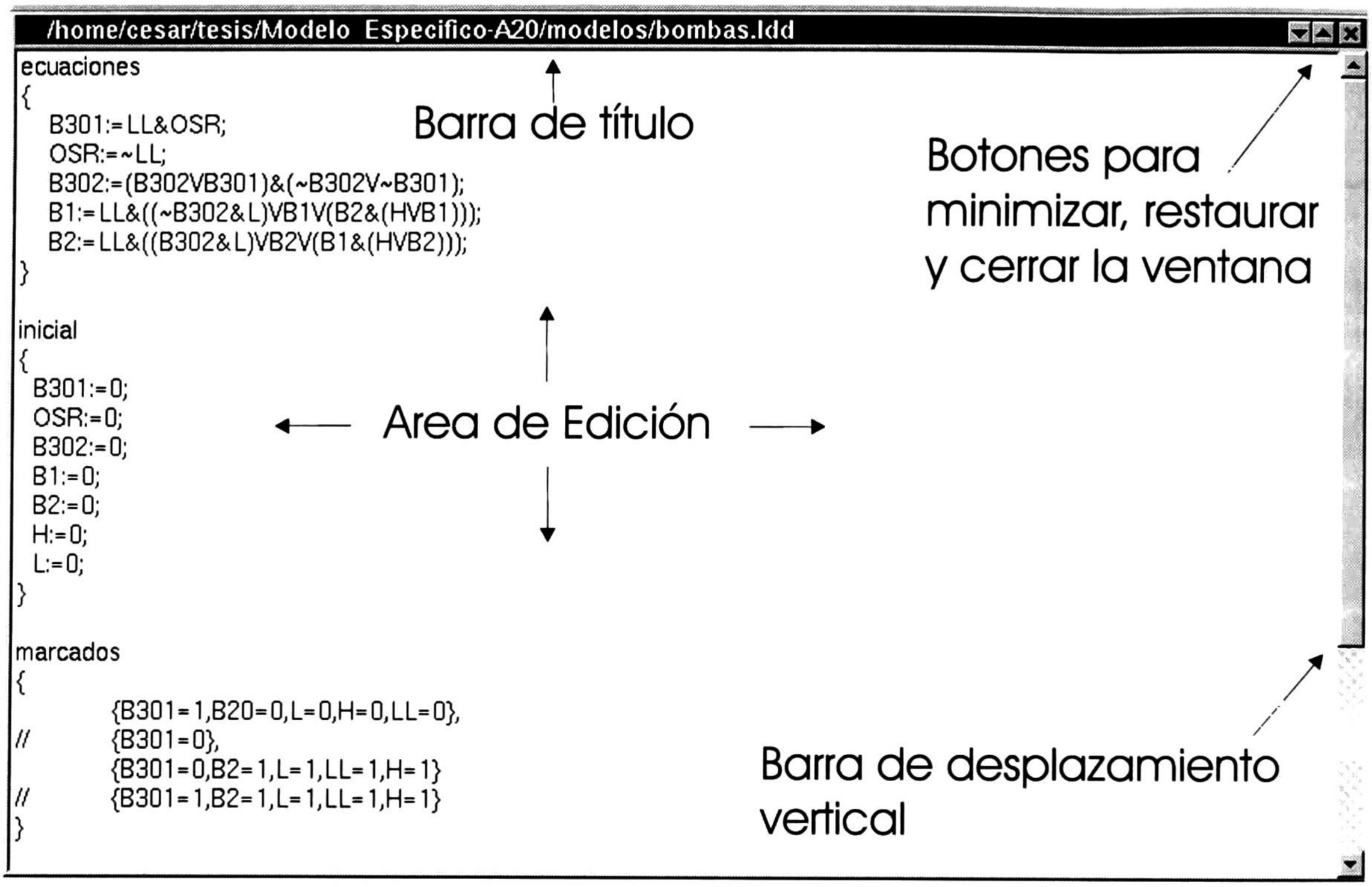


Figura B.2: La ventana de edición es administrada como una ventana hija por la interfaz. Se agrupa en el área de trabajo junto con las demás ventanas de edición.

computadora. Las partes que componen una ventana hija se muestran en la figura B.2.

Las herramientas antes mencionadas fueron incorporadas a la interfaz de la siguiente manera:

- La herramienta de generación de modelos a partir de la descripción del sistema en lenguaje de escalera, se incorporó como un módulo independiente al cual se le hacen las llamadas correspondientes para realizar alguna de sus cuatro funciones:

  1. Generación del modelo del sistema.
  2. Generación de un modelo abstracto del sistema con respecto a una propiedad.
  3. Comprobación del contraejemplo obtenido en la verificación de un modelo abstracto del sistema.
  4. Refinamiento de un modelo abstracto del sistema con respecto a una propiedad y variables adicionales.

- La herramienta de comprobación de modelos y generación de modelos a partir de una propiedad descrita en LTLP, fue incorporada como un programa residente en memoria (daemon) al cual la interfaz se conecta mediante un *socket* con el fin de hacer las llamadas correspondientes para realizar alguna de sus funciones:

  1. Generar el modelo de una propiedad descrita en LTLP.
  2. Cargar un modelo del sistema.
  3. Cargar una propiedad.
  4. Realizar la verificación formal de un modelo y una propiedad cargados en memoria.
  5. Generar el archivo en formato postscript del modelo del sistema cargado.
  6. Generar el archivo en formato postscript del modelo de la negación de la propiedad cargada.
  7. Generar el archivo en formato postscript del modelo del autómata de comportamiento.

La razón de dividir la interfaz en estas partes fue facilitar el proceso de verificación. En las siguientes secciones explicaremos brevemente cada una de las funciones que realiza la interfaz agrupadas en las partes mencionadas.

B.3. Archivo

La sección **archivo**, mostrada en la figura B.3 nos sirve para administrar los archivos de entrada y salida de las herramientas mencionadas.

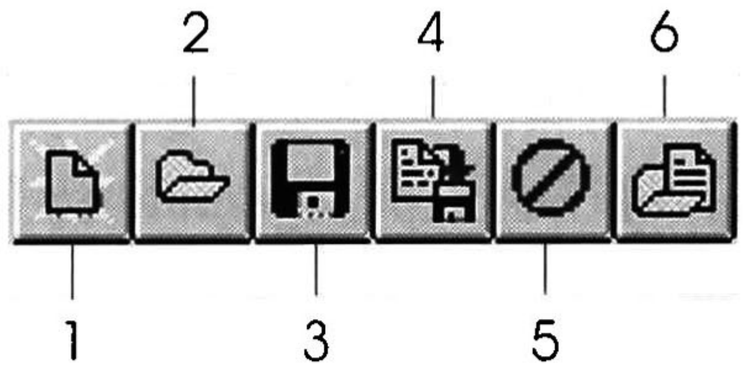


Figura B.3: Barra de herramientas para el manejo de archivos.

Las funciones que puede realizar son las siguientes:

1. **Generar un archivo nuevo**. La interfaz permite generar un archivo para su posterior edición. Al presionar el icono *nuevo* (icono 1) se crea un documento nuevo de texto y aparecen los menús y botones de las partes 2, 3, 4, 5 y 6 de la interfaz. El archivo creado toma por defecto el nombre de *nuevo*1 para el primer archivo creado y así sucesivamente. Se puede crear un número indeterminado de archivos. El límite es la memoria de la computadora.

2. **Abrir archivo**. El comando abrir archivo se habilita con el icono *abrir* (icono 2). Una vez presionado el botón se abre un cuadro de diálogo para buscar el archivo deseado. Por defecto la extensión del archivo para abrir es *.fsm* la cual corresponde a los modelos del sistema. Una vez seleccionado el archivo deseado se presiona el botón aceptar. El cuadro de diálogo se cierra y se abre el archivo deseado en una ventana nueva la cual se convierte en la ventana activa. Al abrir satisfactoriamente un archivo se habilitan los menús 2, 3, 4, 5 y 6 de la interfaz.

3. **Guardar archivo**. El comando guardar archivo se activa con el icono *guardar* (icono 3). Una presionado el botón se abre un cuadro de diálogo para nombrar el archivo actual si éste ha sido creado por el comando *nuevo*, y posteriormente guardarlo. Si el archivo fue abierto con el comando *abrir* simplemente el archivo se guardará sin hacer preguntas. El archivo a guardar es el que se encuentra en la ventana activa.

4. **Guardar archivo como**. El comando guardar archivo como se activa con el icono *guardar como* (icono 4). Una vez que se ha presionado el botón se abre un cuadro de diálogo para nombrar el archivo actual. El archivo que se va a guardar es el que se encuentra en la ventana activa.

5. **Cerrar archivo**. La función cerrar archivo se activa con el icono *cerrar* (icono 5). Una vez presionado el botón tratará de cerrar el archivo que se encuentra en la ventana actual. Si el archivo ha sido modificado y no se han guardado los cambios se abrirá un cuadro de diálogo preguntando si se quieren guardar los cambios, en caso afirmativo se guarda el archivo y después se cierra. En caso contrario simplemente se cierra el archivo actual.

6. **Actualizar archivo**. La función actualizar archivo se activa con el icono *actualizar*. Una vez presionado el botón, se verifica que el archivo en la ventana actual sea diferente al archivo en disco duro, si es diferente lo vuelve a abrir en la ventana activa, en caso contrario no hace nada.

Para comenzar a trabajar con la herramienta se necesita abrir un archivo o crear un archivo.

B.4. Edición

La sección **edición**, mostrada en la figura B.4 nos sirve para manipular los archivos de entrada y salida de las herramientas mencionadas.

Las funciones que puede realizar son las siguientes:

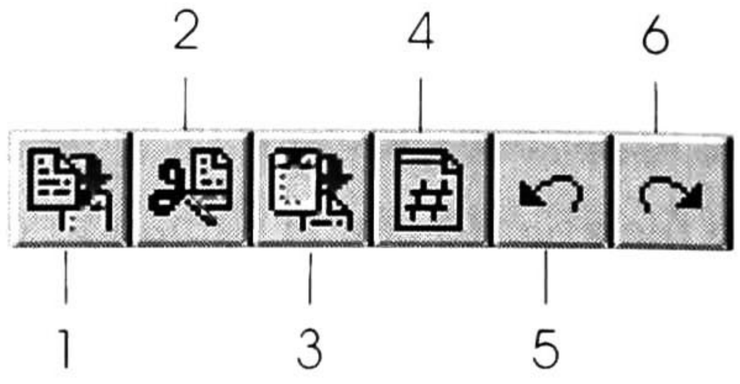


Figura B.4: Barra de herramientas para la edición de documentos.

1. **Copiar texto**. La función copiar texto se activa con el icono *copiar* (icono 1). Una vez presionado el botón se copia el texto seleccionado de la ventana actual al portapapeles del sistema. Si la selección es vacía no se realiza ninguna acción. Para seleccionar el texto se desplaza el cursor al inicio de la parte del texto de la ventana activa que se desea seleccionar, se presiona la tecla *shift* y se desplaza el cursor hasta la posición final del texto que se desea seleccionar. El texto seleccionado aparecerá resaltado.

2. **Cortar texto**. El comando cortar texto se activa con el icono *cortar* (icono 2). Una vez presionado el botón se corta el texto seleccionado de la ventana activa y se almacena en el portapapeles del sistema. Si la selección es vacía no se realiza ninguna acción.

3. **Pegar texto**. La función pegar texto se activa con el icono *pegar* (icono 3). Una vez presionado el botón se pega el contenido del portapapeles al comienzo del área donde se encuentra el cursor en la ventana activa. Si el portapapeles está vació no se realiza ninguna acción.

4. **Seleccionar todo el texto**. La función seleccionar todo el texto se activa con el icono *seleccionar* (icono 4). Una vez presionado el botón se selecciona todo el texto de la ventana activa.

5. **Deshacer acciones de edición**. Si se ha realizado alguna acción de edición (por ejemplo, se ha copiado texto) y se desea volver al estado anterior a esas acciones se debe activar el comando deshacer, lo cual se logra al presionar el

botón con el icono *deshacer* (icono 5). Cada acción de edición de texto que se efectúa se guarda en una lista de acciones, se pueden guardar hasta 256 acciones. Si el archivo actual ha sido guardado la lista de acciones se limpia y no se podrá deshacer ninguna acción realizada antes de haber guardado el archivo.

6. **Rehacer acciones de edición**. El comando rehacer realiza la función inversa al comando deshacer, y se activa al presionar el icono *rehacer* (icono 6). Se pueden rehacer un total de hasta 256 acciones deshechas. Si se ha guardado un archivo inmediatamente después del comando deshacer no se podrán rehacer las acciones anteriores.

B.5. Construcción de Modelos

La sección referente a la construcción de modelos se encarga de hacer las llamadas necesarias a la herramienta de modelado desarrollada en este trabajo.

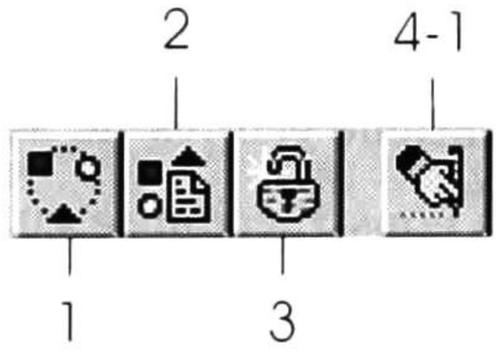


Figura B.5: Barra de herramientas para la construcción de modelos.

La barra de herramientas correspondiente se muestra en la figura B.5. Las funciones que realiza son:

1. **Construir Autómata**. La función construir automata se activa con el icono *construir* (icono 1). Una vez presionado el botón correspondiente se abre un cuadro de diálogo para abrir el archivo que contiene la información del sistema. Inmediatamente después se abre una cuadro de diálogo que pide el nombre del archivo en donde se guardarán los resultados de la construcción. Allí se indica el número de estados y transiciones y el tiempo de construcción. En caso de que

haya algún error en el archivo del sistema se indica en que parte se localiza. Si la función se realiza en forma correcta se genera un archivo que contiene el modelo del sistema y se guarda en el mismo directorio donde se encuentra el archivo de sistema, con el mismo nombre pero con la extensión *.fsm*. Si el archivo contiene errores no se genera el modelo del sistema.

2. **Construir autómata abstracto**. La función construir automata abstracto se activa con el icono *autómata abstracto* (icono 2). Una vez presionado el botón correspondiente se abre un cuadro de diálogo para abrir el archivo que contiene la información del sistema. Una vez cargado el archivo del sistema se abre otro cuadro de diálogo para cargar el archivo que contiene la propiedad mediante la cual se va a hacer la simulación por proyección de variables de la propiedad. La propiedad es dada en LTLP. Si la función se realiza en forma correcta se genera un archivo que contiene el modelo abstracto del sistema y se guarda en el mismo directorio que contiene el archivo del sistema, el nombre con el que se guarda se forma con el nombre del archivo del sistema seguido de un guión bajo y después el nombre del archivo de la propiedad, la extensión del archivo es .fsm. Si alguno de los archivos contiene errores no se genera el modelo abstracto del sistema. Los resultados de la construcción se guardan en un archivo especificado justo después de haber abierto el archivo de la propiedad (se abre un cuadro de diálogo pidiendo el nombre del archivo de resultados). Allí se indica nombre del archivo del sistema, el nombre del archivo de la propiedad, el numero de estados y transiciones y el tiempo de construcción. En caso de que haya algún error en alguno de los archivos se indica en que parte se localiza.

3. **Comprobar contraejemplo abstracto**. La función comprobar contraejemplo abstracto se activa con el icono *contraejemplo* (icono 3). Una vez presionado el botón correspondiente se abre un cuadro de diálogo para abrir el archivo que contiene la información del sistema. Una vez cargado el archivo del sistema se abre otro cuadro de diálogo para cargar el archivo de la propiedad, una vez cargado el archivo de la propiedad se abre otro cuadro de diálogo que pide el archivo que contiene el contraejemplo abstracto (el contraejemplo se obtiene en

la función de verificación). Inmediatamente después se pide al usuario el nombre de un archivo en donde se van a guardar los resultados de la comprobación del contraejemplo abstracto. La información que contiene el archivo de resultados es el nombre del archivo del sistema, el nombre del archivo de la propiedad, el nombre del archivo del contraejemplo. Si el contraejemplo es válido se genera el computo que lo produce, en caso contrario se indica el estado en el cual se generó la falla.

4. **Generar el automata de una propiedad**. La función generar el autómata de una propiedad se activa con el icono *propiedad* (icono 1 parte 4). Una vez presionado el botón correspondiente se abre un cuadro de diálogo para abrir el archivo que contiene la propiedad. Posteriormente se realiza una llamada a la herramienta construida en [11] y si la propiedad está correctamente escrita se genera un archivo con el modelo de la propiedad descrita. El archivo tiene el mismo nombre que el archivo de la propiedad pero con la extensión .fsm.

B.6. Verificación de Modelos

La parte referente a la verificación formal por comprobación de modelos se muestra en la figura B.6, la cual se encarga de realizar la verificación de un modelo del sistema junto con una propiedad especificada.

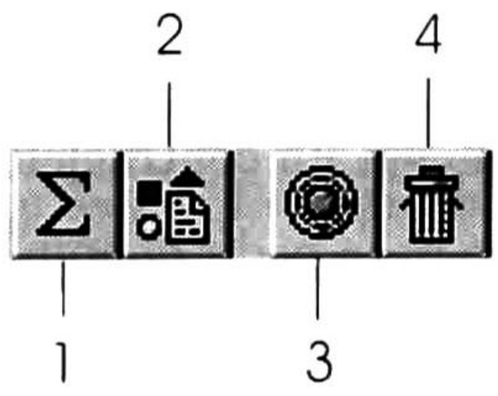


Figura B.6: Barra de herramientas para realizar la verificación formal.

Las funciones que realiza son:

1. **Cargar formula**. El comando cargar fórmula se activa al presionar el botón con el icono *cargar fórmula* (icono 1). Una vez presionado el botón se abre un cuadro de diálogo que pide el nombre del archivo que contiene la propiedad que se desea cargar. Si la propiedad ha sido cargada de manera correcta se muestra un aviso que así lo indica. En la barra de estado (parte inferior de la pantalla) aparece el nombre del archivo de la propiedad que está cargada actualmente.

2. **Cargar Automata**. El comando cargar autómata se activa al presionar el botón con el icono *cargar autómata* (icono 2). Una vez presionado el botón se abre un cuadro de diálogo que pide el nombre del archivo que contiene el autómata que representa el modelo del sistema. Si el autómata ha sido cargado de manera correcta se muestra un aviso que así lo indica. En la barra de estado (parte inferior de la pantalla) aparece el nombre del archivo del autómata que está cargado actualmente.

3. **Correr Verificación**. La función correr verificación se activa al presionar el botón con el icono *verificación* (icono 3). Una vez presionado el botón se abre un cuadro de diálogo con un nombre propuesto de archivo en el cual se van a guardar los resultados de la verificación. Para realizar el proceso de verificación se hace una llamada a la herramienta desarrollada en [43] para que realice la verificación del modelo del sistema cargado en memoria y de la propiedad cargada en memoria. En caso de que alguno de los archivos (modelo o propiedad) no esté cargado en memoria se muestra un aviso. Si las variables de la propiedad no son un subconjunto de las variables del sistema se muestra un aviso. Los resultados de la verificación se guardan en el archivo elegido. En caso de que la propiedad no es válida, se genera un contraejemplo que se guarda en un archivo, se abre un cuadro de dialogo para pedir el nombre y se pone el prefijo *contra.cte*. Si el contraejemplo generado fue obtenido al verificar un modelo abstracto, éste puede ser utilizado en la comprobación del contraejemplo abstracto.

4. **Borrar autómatas de la Memoria**. El comando borrar autómatas se activa al presionar el botón con el icono *borrar* (icono 4). Una vez presionado el botón se procede a eliminar los autómatas del modelo y de la propiedad que están

cargados actualmente en memoria. En caso de que no haya autómatas cargados en memoria no se hace nada.

B.7 Generación de Archivos con Formato Postscript

La barra de herramientas mostrada en la figura B.7 se encarga de generar archivos con formato postscript de los modelos generados. Las funciones que realiza son:

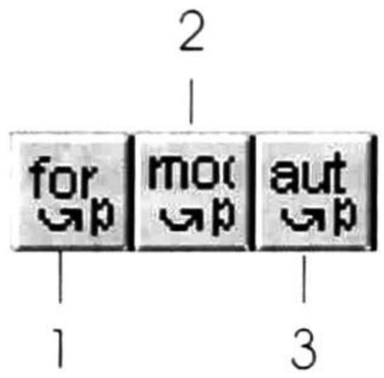


Figura B.7: Barra de herramientas para la generación de archivos en formato postscript.

1. **Generar el archivo con formato postscript del modelo de la negación de la propiedad**. El comando generar postscript del modelo de la negación de la propiedad se activa al presionar el botón con el icono *forps* (icono 1). Una vez presionado el botón se verifica que haya en memoria una propiedad cargada, en caso afirmativo se procede a generar el postscript. El archivo generado se guarda con el mismo nombre del archivo que contiene la propiedad cargada pero con la extensión .ps. En caso de que se haya generado el archivo correctamente aparece un aviso que así lo indica.

2. **Generar el archivo con formato postscript del modelo del sistema**. El comando generar postscript del modelo del sistema se activa al presionar el botón con el icono *modps* (icono 2). Una vez presionado el botón se verifica que haya en memoria un modelo cargado, en caso afirmativo se procede a generar el postscript. El archivo generado se guarda con el mismo nombre del archivo

que contiene el modelo del sistema pero con la extensión *.ps*. En caso de que se haya generado el archivo correctamente aparece un aviso que así lo indica.

3. **Generar el archivo con formato postscript del autómata de comportamiento**. El comando generar postscript del autómata de comportamiento se activa al presionar el botón con el icono *autps* (icono 3). Una vez presionado el botón se verifica que haya en memoria un automata de comportamiento cargado en memoria. El autómata de comportamiento se carga en memoria una vez que se ha realizado el proceso de verificación. En caso afirmativo se abre un cuadro de diálogo que pide el nombre del archivo que se generará. En caso de que se haya generado el archivo correctamente aparece un aviso que así lo indica.

B.8. Administración de Ventanas

La administración de ventanas nos permite tener un control de la visualización de los diversos archivos abiertos en la interfaz.

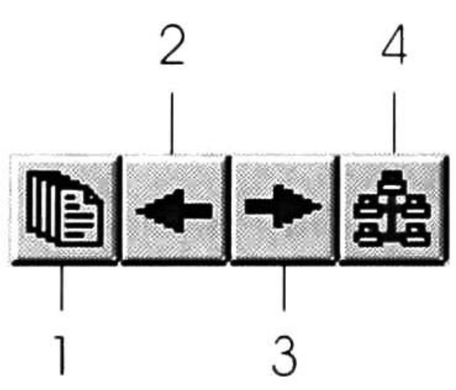


Figura B.8: Barra de herramientas para la administración de ventanas.

La figura B.8 nos muestra la barra de herramientas para administrar las ventanas. Las funciones que realiza son:

1. **Ventanas en Cascada**. La función ventanas en cascada se activa con el icono *cascada* (icono 1). Una vez presionado el botón correspondiente se ordenan las ventanas abiertas en la interfaz de manera que se muestra la barra de titulo de cada ventana ordenadas en forma vertical. La ventana activa es la que estaba activa justo antes de activar el comando.

2. **Ventana anterior**. La función ventana anterior se activa con el icono *anterior* (icono 2). Una vez presionado el botón correspondiente se activa la ventana anterior a la ventana actual. El orden de ventanas se determina por la forma en que fueron creadas.

3. **Ventana siguiente**. La función ventana siguiente se activa con el icono *siguiente* (icono 3). Una vez presionado el botón correspondiente se activa la ventana siguiente a la ventana actual. El orden de ventanas se determina por la forma en que fueron creadas.

4. **Ventanas en Mosaico**. La función mosaico se activa con el icono *mosaico* (icono 4). Una vez presionado el botón correspondiente todas las ventanas abiertas se distribuyen uniformemente en el área de trabajo de la interfaz. Mediante este comando podemos observar todos los archivos abiertos al mismo tiempo. La venta activa es la que estaba activa justo antes de activar el comando.

B.9. Instalación de la Interfaz

Para la instalación de la interfaz se deben tener los elementos necesarios para correr la herramienta desarrollada en [43], en ese trabajo se hace una descripción clara del software que se necesita así como de la manera en que se debe instalar.

La interfaz está desarrollada para Linux. Se ha probado la interfaz en los sistemas Linux Mandrake 9, Mandrake 9.2 y en Linux Red Hat 9.

Instrucciones de instalación:

- Se debe crear la carpeta de instalación llamada *InterfazVerificador* la cual debe contener los siguientes archivos:
  - PVerificador
  - ldd
  - vr
  - _mp

- front_mp
- libborqt.so.2.3.0[1]

- Se debe crear un archivo en esa misma carpeta que se llame PVerificador.bash, en ese archivo se deben editar las líneas mostradas en la figura B.9

```
cat PVerificador.bash
#!/bin/bash
export LD_LIBRARY_PATH=/home/usr/InterfazVerificador
/./home/usr/InterfazVerificador/PVerificador
```

Figura B.9: Instrucciones en el archivo PVerificador.bash

Donde */home/usr/* es la ruta donde se pondrá la carpeta InterfazVerificador.
Para ejecutar el programa se debe ejecutar el siguiente comando en una consola:

[/home/usr/InterfazVerificador/]$ sh PVerificador.bash &

Como mencionamos el archivo PVerificador.bash debe estar en la carpeta de instalación y esa ruta debe ser la de la terminal para poderlo ejecutar de manera correcta.

Una vez ejecutado el comando, la interfaz aparecerá y es ahora posible trabajar con las opciones descritas.

[1]esta librería es necesaria para que funcione la interfaz gráfica desarrollada en kylix

Bibliografía

[1] A. Aiken, M. Fähndrich, and Z. Su. Detecting races in relay ladder logic programs. *Int. J STTT(2000)*, 3:93–105, 2000.

[2] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Computer Aided Verification*, pages 521–525, 1998.

[3] H. R. Andersen. An introduction to binary decision diagrams. *Lecture notes for 49285 Advanced Algorithms E97, Department of Information Technology, Technical University of Denmark*, 1997.

[4] M. Bani and G. Frey. Formalization of existing plc programs: A survey. *CESA 2003*, pages Paper No. S2–R–00–0239, July 2003. Disponible en http://www.eit.uni-kl.de/frey/papers/Abstracts/V173.htm.

[5] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: deductive-algorithmic verification of reactive and real-time systems. In *8th International Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418, Saratoga, NY, July 1996. Springer-Verlag.

[6] R. K. Blackett. As good as gold. *IEEE Sprectrum*, pages 68–71, June 1996.

[7] R.S. Boyer and J.S. Moore. A computational logic handbook. Boston, 1998. Academic Press.

[8] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 428–432, New Brunswick, NJ, USA, July 1996. Springer Verlag.

[9] D. Bustan and O. Grumberg. Simulation based minimization. In *Proc. 17th Int'l Conference on Automated Deduction (CADE'00)*, volume 1831 of *Lecture Notes in Computer Science*, pages 255–270, Pittsburgh, PA, June 2000. Springer-Verlag.

[10] F. P. Carpio. Vhdl lenguaje para descripción y modelado de circuitos. In *VHDL Lenguaje para descripción y modelado de circuitos.* Universidad de Valencia, 1997.

[11] B. Casillas. Representación de una fórmula de lógica temporal lineal proposicional como un autómata de büchi generalizado etiquetado. Tesis de maestría, CINVESTAV del IPN, Unidad Guadalajara, 2004.

[12] V Chandra and R. Kumar. An industrial strength version of nqthm. *Anual conference on Computer Assurance*, June 1996.

[13] V. Chandra and R. Kumar. A new modeling formalism and automata model generator for a class of discrete event systems. *American Control Conference*, pages 4562–4567, June 2001.

[14] A. Cimatti, E. M. Clarke, F Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, April 2000.

[15] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *In Logic of Programs: Workshop, Yorktown Heights, NY*, 131, 1981.

[16] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.

[17] E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Sprectrum*, pages 61–67, June 1996.

[18] J. M. Cobleigh, L. A. Clarke, and L. J. Osterwil. The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification. *Department of Computer Science, University of Massachusetts Amherst*, 2000.

[19] D. L. Dill, A. J. Drexler, and A. J. Hu. Protocol verification as a hardware design aid. *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, October 1992.

[20] A. Dovier, C. Piazza, and A. Policriti. A fast bisimulation algorithm: test. Udine, 2000. Disponible en http://citeseer.ist.psu.edu/470332.html

[21] E. Allen Emerson and Richard J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Conference on Correct Hardware Design and Verification Methods*, pages 142–156, 1999.

[22] K. Fisler and M. Vardi. Bisimulation minimization in an automata-theoretic verification framework, October 1998. Disponible en: http://www.cs.rice.edu/{ kfisle, vardi}.

[23] M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, 1996.

[24] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Enviroment for Higher-Order Logic.* Cambridge Univ. Press, Cambrigde, UK, 1993.

[25] A. Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1992.

[26] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[27] Gerard Holzmann and Doron Peled. Partial order reduction of the state space. In *First SPIN Workshop*, Montrèal, Quebec, 1995.

[28] T. A. Hughes. *Programmable controllers.* Research Triangle Park, NC: Instrument Society of America, 3rd edition, October 2000. 334 pages.

[29] C. Ip and D. L. Dill. Better verification through symmetry. In *Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Canada, 1993.

[30] C. Kern and M. Greenstreet. Formal verification in hardware design: A survey. *ACM Transactions on Design Automation of Electronic Systems*, 4:123–193, June 1996.

[31] R.P. Kurshan. Formal verification in a commercial setting. *Design Automation Conference*, pages 258–262, June 1997.

[32] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[33] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc.* O'Reilly & Associates, 2nd/updated edition, October 1992. 366 pages.

[34] Y. Lu. *Automatic Abstraction in Model Checking.* PhD thesis, Carnegie Mellon University, 2000.

[35] Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems. In *Specification*, New York, 1991. Springer-Verlag.

[36] Z. Manna and A. Pnueli. Temporal verification of reactive systems: Safety. New York, 1995. Springer-Verlag.

[37] K. L. McMillan. *Symbolic Model Checking: An approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.

[38] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic, 1993.

[39] I. Moon. Modelling programmable logic controllers for logic verification. *IEEE Control Systems*, pages 53–59, 1994.

[40] I. Moon and G. Powers. Automatic verification of sequential control systems using temporal logic. *AIChE*, 38:67–75, 1992.

[41] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[42] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proc. Sixth Int'l Conf. Computer Aided Verification (CAV94)*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390, Stanford, Calif., 1994. Springer-Verlag.

[43] G. Perez. Comprobación de modelos explícita basada en autómatas de büchi y lógica temporal lineal. Tesis de maestría, CINVESTAV del IPN, Unidad Guadalajara, 2003.

[44] T. Probst, G. Powers, and I. Moon. Verification of a logically controled, solids transport system using symbolic model checking. *Computers chemical Engineers*, 21:417–429, 1997.

[45] A. Rajeev and D. Dill. A theory of timed automata. *Theoretical Computer Science*, pages 183–235.

[46] M. Raush and H. Hanish. Net condition/event systems with multiple condition outputs. *EFTA 95*, 1:592–600, October 1995.

[47] M. Raush and B. Krogh. Formal verification of plc programs. *American Control Conference*, pages 417–429, June 1998.

[48] O. Rossi and P. Schnoebelen. Formal modeling of timed function blocks for the automatic verification of ladder diagrams programs. 4^{th} *Int. Conf. Automation*

of Mixed Precesses: Hybrid Dynamic Systems (ADPM), Dortmund, Germany, September 2000.

[49] A. Sanchez, G. Rotstein, N. Alsop, and S. Macchietto. Synthesis and implementation of procedural controllers for even-driven operations. *AIChE Journal*, 45(8), 1999.

[50] O. Smet, S. Couffin, O. Rossi, G. Canet, J. Lesage, P. Schnoebelen, and H. Papini. Safe programming of plc using formal verification methods. 4^{th} *Int. PLCopen Conf. on Industrial Control Programming (ICP'2000)*, pages 73–78, October 2000.

[51] J. Wing. Hints to specifiers. In *Teaching and Learning Formal Methods, Academic Press*, Carnegie Mellon University, Pittsburgh, PA, 1996.

[52] S. H. Yang, O. Stursberg, P. W. H. Chung, and S. Kowalewski. Automatic safety analysis of computer-controlled plants. *Computers chemical Engineers*, 25:913–922, 2001.

[53] B. Zoubek. Automatic verification of programs for control systems. *MOVEP 02*, pages 435–440, May 2002.

[54] B. Zoubek, J.M. Roussel, and M. Kwiatkowska. Towards automatic verification of ladder logic programs. *CESA 2003*, pages Paper No. S2–R–00–0239, July 2003. Disponible en http://www.eit.uni-kl.de/frey/papers/Abstracts/V173.htm.

Centro de Investigación y de Estudios Avanzados del IPN
Unidad Guadalajara

El Jurado designado por la Unidad Guadalajara del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, aprobó la tesis:

Modelado y verificación por comprobación de modelos de una clase de sistemas de procesamiento industrial

del (la) C.

Cesar Alberto HERNANDEZ ARANA

el día 4 de Mayo de 2004.

Dr. Arturo del Sagrado Corazón
SÁNCHEZ CARMONA
Investigador Cinvestav 3B
CINVESTAV GDL
Jalisco

Dr. Deni Librado TORRES ROMÁN
Profesor Investigador 3A
CINVESTAV GDL
Jalisco

Dr. Raul Ernesto GONZÁLEZ TORRES
Investigador Cinvestav 2C
CINVESTAV GDL
Jalisco

Dr. Antonio RAMIREZ TREVIÑO
Investigador Cinvestav 2A
CINVESTAV GDL
Jalisco