Universidad de las Ciencias Informáticas

Facultad 9

Título: “Técnicas de Programación y Algoritmos para la Edición Gráfica de Programas en Lenguaje SFC”

Trabajo de Diploma para optar por el Título de
Ingeniero en Ciencias Informáticas

Autor: Carmen Luz Amador Bellón

Tutores: Lic. Rafael Arturo Trujillo Rasúa

Lic. Rolando Trujillo Rasúa

Co-tutor: Lic. Ramiro Feria Purón

Junio de 2007

DECLARACIÓN DE AUTORÍA

Declaramos ser autores de la presente tesis y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmo la presente a los ____ días del mes de _____ del año _____.

Carmen Luz Amador Bellón	**Lic. Rolando Trujillo Rasúa**
_____	_____
Firma del Autor	Firma del Tutor

DATOS DE CONTACTO

Lic. Rafael Arturo Trujillo Rasúa

Licenciado en Ciencias de la Computación, Universidad de Oriente, 2002. Profesor Instructor. Tres años de experiencia en el tema. Cinco años de graduado.

Lic. Rolando Trujillo Rasúa

Licenciado en Ciencias de la Computación, Universidad de la Habana, 2006. Profesor Adiestrado. Cuatro años de experiencia en el tema. Un año de graduado.

Dedicatoria

- A mis padres por su apoyo y comprensión.
- A mi familia.
- A mi pareja.

Agradecimientos

- A mis tutores por toda la ayuda.
- A todos los que me han ayudado a continuar…

Resumen

En este documento se presenta un análisis y se ofrece solución a los problemas fundamentales asociados al desarrollo de una *herramienta de edición gráfica de programas en lenguaje SFC* para la programación de *PLCs*. Primeramente, se incursiona sobre la definición sintáctica de SFC y sus elementos según lo establecido en la norma IEC 1131-3. Se discuten además los problemas esenciales referentes a la edición gráfica de programas SFC, a la vez que se da una panorámica acerca de las características y limitaciones de las herramientas de edición gráfica de SFC en los entornos comerciales existentes. Posteriormente son expuestos los resultados obtenidos: se proponen herramientas de desarrollo apropiadas para construir la herramienta deseada; se presenta y discute un conjunto de clases para representar los conceptos fundamentales en el dominio de la aplicación; se describen los algoritmos diseñados para la implementación de operaciones sobre un diagrama SFC; y por último se describen algoritmos para la representación geométrica de los elementos de SFC.

Palabras Claves

Algoritmos SFC
Controladores Lógicos Programables (PLCs)
Diagrama SFC
Edición gráfica de SFC
Lenguaje SFC
Programa SFC
Representación gráfica de SFC
SFC

Índice

Relación de figuras

Introducción

Los procesos industriales y de manufactura modernos son generalmente controlados por computadoras. Incluso muchos de los artefactos de nuestro uso cotidiano, tales como el teléfono o el televisor contienen microcontroladores que son, en esencia, computadoras en un solo chip. Estas computadoras no son, sin embargo, las computadoras de propósito general que resulta usual encontrar en las oficinas, sino computadoras industriales especialmente adaptadas a los propósitos de los sistemas de control. Ellas deben ser robustas, confiables y baratas. Robustas para poder trabajar bajo condiciones anormales que pueden incluir altas temperaturas, polvo excesivo, ruido electromagnético, etc. Confiables para trabajar sin interrupción durante 24 horas al día por 5 o 10 años. Baratas para poder penetrar en el mercado reduciendo los costos de los procesos productivos.

Una clase prominente de los controladores industriales son los llamados controladores lógicos programables (PLC), que comenzaron a desarrollarse en los inicios de la década del ‘70. En principio los PLC eran simples dispositivos que reemplazaban a los relés (relays) electromecánicos. A través del uso de circuitos integrados realizaban tareas simples de control secuencial aislados de otros controles y equipos de monitoreo. Estos dispositivos simples fueron creciendo hasta convertirse en sistemas complejos capaces de realizar cualquier aplicación de control incluyendo el control del movimiento, la manipulación de datos y otras funciones computacionales avanzadas.

El desarrollo del estándar IEC 1131-3 se inició para unificar los principales tipos de lenguajes utilizados en la práctica para la programación de PLCs en el mundo. Dentro de este estándar se establecen las normativas referentes a los lenguajes de programación: IL (Instruction List), ST (Structured Text), LD (Ladder Diagram) y FBD (Function Block Diagram). También son definidos elementos de SFC (Sequiential Function Chart) para estructurar la organización interna de los programas y subprogramas de los controladores programables.

Hasta la fecha no existen muchos compiladores para los lenguajes de la norma IEC 1131-3. Además la licencia que permite el uso y explotación de los mismos no es gratis en ninguno de los casos así como tampoco son gratis ninguno de los estándares definidos por la Comisión Electrotécnica Internacional (IEC).

El grupo “EROS” es un grupo de automatización industrial formado por especialistas de la empresa “SerCoNi” de Servicios de Comunicación y Computación del Níquel de Nicaro con el apoyo de profesores del Instituto Superior Minero Metalúrgico de Moa, Holguín. Este grupo, creado hace más de 15 años, ha trabajado en la línea de buscar soluciones cubanas a los problemas de automatización de la industria nacional. En función de este objetivo el grupo “EROS” ha desarrollado, desde su creación, diferentes dispositivos de hardware y un Sistema de Supervisión y Control, internacionalmente conocido como SCADA (Supervisory Control and Data Adquisition), que en la actualidad es el más difundido en el país. De igual modo participó en el proyecto del PLC cubano “Nova”. La necesidad de estandarizar alrededor de la norma IEC 1131-3 la programación de estos productos, conjuntamente con la imposibilidad de obtener copias de los compiladores de los lenguajes de la Norma, debido a las restricciones del bloqueo, motivaron el desarrollo de un proyecto para crear un entorno de desarrollo que soporte los 4 lenguajes de la norma IEC 1131-3 que permitiera la creación y compilación de los programas y eventualmente su puesta a punto. Como fase inicial del ambicioso proyecto, recientemente fue desarrollado un compilador para el lenguaje ST con elementos de SFC hacia la arquitectura Intel [Tru06].

Un entorno de desarrollo para la programación de PLCs se considera incompleto si no posee una herramienta para la edición gráfica de programas SFC. El compilador para el lenguaje ST con que se cuenta hasta el momento permite solamente desarrollar programas SFC de manera textual mediante la escritura de código fuente en el referido lenguaje. Sin embargo, los entornos de desarrollo modernos para la programación de PLCs están dotados de una herramienta que permite al programador la definición gráfica de los programas en lenguaje SFC. La ventaja fundamental consiste en que con la definición textual es más complejo diseñar, estructurar y poner a punto un programa SFC que con la definición gráfica. Además, la representación gráfica de un programa SFC posee un gran valor desde el punto de vista educativo y de la formación del programador de PLCs. Es en consecuencia de gran interés contar con una herramienta para la edición gráfica de programas SFC; una vez editado el programa, la misma herramienta pudiera generar el código ST correspondiente y éste a su vez ser procesado por el compilador con que ya se cuenta.

El desarrollo de una herramienta semejante, sin embargo, resulta una tarea relativamente compleja. Tal proceso no consta de un único problema a resolver, sino que lleva aparejado un conjunto de subproblemas de naturaleza disímil, los cuales es necesario enfrentar. A nuestro juicio, la definición de

una estrategia factible de representación gráfica para SFC y sus elementos, así como el soporte de un conjunto de operaciones para el desarrollo incremental de los programas, son los dos problemas fundamentales y ofrecen el mayor reto desde el punto de vista de la búsqueda de una solución apropiada. Estos y otros problemas son ampliamente tratados en este documento. Como es de esperar, los mismos son afrentados de manera particular por cada una de herramientas de edición comerciales existentes. Nuestra conjetura es que las soluciones tomadas para dar solución a estos problemas en tales herramientas se encuentran lejos de ser las más adecuadas, y poseen un grupo significativo de limitaciones. Así, la motivación fundamental está enfocada hacia la búsqueda de mejores soluciones, capaces de superar tales limitaciones.

El presente documento resume lo investigado acerca de la *edición gráfica de programas en lenguaje SFC*, y sobre todo, ofrece resultados teóricos y prácticos de gran utilidad para el desarrollo posterior de una herramienta de edición gráfica de programas SFC.

Como parte de las tareas acometidas para dar cumplimiento a los objetivos trazados, fue necesario estudiar la definición formal para SFC; además de ello, se investigó respecto a la representación usual – utilizada comúnmente en la literatura – para SFC y sus elementos, así como las herramientas de de edición gráfica de programas SFC en entornos comerciales existentes para la programación de PLCs.

La investigación hecha, sin embargo, no está conformada meramente por los necesarios estudios preliminares; los aportes presentados como parte de la solución constituyen de hecho la parte medular de la misma. Por consiguiente, durante la presentación de los estudios preliminares se omiten cuestiones en las cuales es posible ahondar mediante la revisión de la bibliografía descrita y de los sistemas mencionados; ello se hace a favor de brindar un mayor nivel de detalle a la hora de mostrar los resultados genuinos.

En lo que resta se asume que el lector está familiarizado con materias como el análisis y diseño de algoritmos, la notación “O-grande”, algunas estructuras de datos clásicas, el paradigma de la programación orientada a objetos, el lenguaje de programación C++, así como su librería estándar. Por otro lado, si bien se ofrecen fundamentos teóricos relacionados con SFC y la programación de PLCs en general, el individuo con experiencia en el tema – como es de esperar – tendrá un mejor entendimiento del contenido, encontrará un mayor placer en su lectura, y tendrá en mayor estima la utilidad del mismo.

A continuación aparecen por ese orden el problema a resolver, el objeto de estudio, el campo de acción, los objetivos generales y específicos, así como los métodos científicos utilizados en la investigación. Finalmente se explica de manera breve la estructura del documento.

Problema a resolver

Aportar los elementos necesarios para el desarrollo de una *herramienta de edición gráfica de programas en lenguaje SFC*, que supere a la vez las limitaciones de las herramientas de edición de SFC correspondientes a los entornos comerciales existentes para la programación de PLCs.

Objeto de estudio

Edición gráfica de programas en lenguaje SFC.

Campo de acción

Herramientas de desarrollo, técnicas de programación y algoritmos a utilizar para el desarrollo de una herramienta para la edición gráfica de programas en lenguaje SFC.

Objetivos Generales

El objetivo general de este trabajo consiste en proponer herramientas de desarrollo adecuadas, así como discutir técnicas de programación y algoritmos, para su utilización en el futuro desarrollo de una herramienta de edición gráfica de programas SFC.

Objetivos Específicos

Los objetivos específicos se enumeran a continuación:

1. Proponer herramientas de desarrollo apropiadas para la implementación de un editor gráfico de programas SFC.

2. Diseñar e implementar un conjunto de clases en Lenguaje C++ para la representación de los conceptos asociados a SFC y sus elementos, en correspondencia con lo establecido por la Norma IEC-1131. Discutir alternativas de diseño.

3. Establecer invariantes para cada una de las clases definidas. Explicar la semántica de las clases y sus respectivas interfaces, conveniencia y modo de uso.

4. Proponer algoritmos formales para la implementación de operaciones en las clases propuestas. Analizar aspectos referentes a la complejidad de los mismos, y proponer una representación conveniente para las clases, en favor de los requerimientos de eficiencia de dichos algoritmos.

5. Concebir una estrategia para la representación gráfica de programas SFC. Proponer algoritmos formales para la representación geométrica sobre el plano de programas SFC y sus elementos.

6. Iniciar el desarrollo de un prototipo de editor de programas SFC con funcionalidad básica sobre la plataforma Windows, haciendo uso de las herramientas de desarrollo, técnicas de programación y algoritmos propuestos.

Idea a defender

Las herramientas de desarrollo, técnicas de programación y algoritmos presentados en esta tesis son eficaces para el desarrollo de una herramienta de edición gráfica de programas en lenguaje SFC que supere las limitaciones y deficiencias de las herramientas de edición de SFC correspondientes a los entornos de desarrollo comerciales existentes para la programación de PLCs.

Métodos científicos

Durante el desarrollo de la investigación se hizo uso intensivo de varios métodos científicos, tanto teóricos como empíricos, los cuales se mencionan a continuación:

Métodos teóricos:

1. *Método hipotético deductivo*. Fue el principal método utilizado en la obtención de los algoritmos para la implementación de las operaciones sobre diagramas SFC (objetivo específico 4).

2. *Método de modelación*. En ocasiones fue necesario modelar algunos aspectos de la realidad mediante la introducción de nuevos conceptos, con el fin de facilitar la obtención de los resultados deseados. Por ejemplo, para los algoritmos geométricos diseñados (objetivo específico 5), fueron introducidos los conceptos de *representación geométrica de una transición*, y de *conjunto de segmentos asociados a un paso* para representación geométrica de transiciones disyuntivas divergentes, disyuntivas convergentes y simultáneas, respectivamente. Por otro lado, el diseño de un conjunto de clases para una aplicación (objetivos específicos 2 y 3) es un excelente ejercicio de modelación de la realidad.

Métodos empíricos:

1. *Método de la observación*. Fue este el método por excelencia utilizado en para el estudio de las herramientas de edición gráfica de programas en lenguaje SFC correspondientes a los entornos comerciales existentes para la programación de PLCs.

2. *Método de la medición*. Para la obtención de los algoritmos geométricos también se hizo uso de este método, en conjugación con el método experimental. También estuvo presente en cada uno de los análisis de complejidad algorítmica realizados a lo largo del trabajo.

3. *Método experimental*. Este fue el método fundamental utilizado durante la obtención de los algoritmos geométricos para la representación gráfica de los elementos de SFC (objetivo específico 5), aparejado al método de la medición y de la observación.

Estructura del documento

En el primer capítulo se hace inicialmente una breve introducción a los PLCs. Acto seguido, se incursiona sobre la definición sintáctica de SFC y sus elementos, según lo establecido en la norma IEC 1131-3. Además de ello, se discuten los problemas fundamentales asociados a la construcción de una herramienta para la edición gráfica de programas SFC; en particular se abordan cuestiones referentes a la representación gráfica de los elementos de SFC, y a las operaciones sobre un diagrama SFC. Al mismo tiempo, se ofrece una panorámica general acerca de las características de las herramientas comerciales existentes.

El segundo capítulo recoge los aportes fundamentales de esta tesis. Primeramente se proponen herramientas de desarrollo apropiadas para construir la herramienta deseada. Posteriormente es presentado y sometido a discusión un conjunto conveniente de clases para representar los conceptos fundamentales en el dominio de la aplicación. A continuación, aparecen algunos algoritmos diseñados para la implementación de operaciones sobre un diagrama SFC. Por último, se incluyen también algoritmos formales para la representación geométrica de los elementos de SFC.

El cierre del documento está constituido por las conclusiones y recomendaciones, seguidas por la relación bibliográfica.

Capítulo1. Estudios Preliminares

En este capítulo se presentan los fundamentos teóricos y la parte esencial de estudios preliminares realizados como parte de la investigación, con el propósito de facilitar la comprensión de las soluciones y los resultados expuestos en el siguiente capítulo.

Inicialmente una breve introducción a los PLCs. Acto seguido, se incursiona sobre la definición sintáctica de SFC y sus elementos, según lo establecido en la norma IEC 1131-3. Además de ello, se enuncian y discuten los problemas fundamentales asociados a la construcción de una herramienta para la edición gráfica de programas SFC; en particular se hace hincapié en aquellas cuestiones referentes a la representación gráfica de los elementos de SFC y a las operaciones sobre un diagrama SFC. Al mismo tiempo, se ofrece una panorámica acerca de las características de las herramientas de edición gráfica de programas SFC correspondientes a los entornos de desarrollo comerciales existentes, enfatizando en las limitaciones e inconvenientes fundamentales de dichas herramientas.

1.1. Introducción a los PLCs

Los PLCs [Jac04] son controladores lógicos programables encargados de controlar procesos industriales. Estos dispositivos se han convertido en sistemas complejos capaces de realizar aplicaciones de control, manipulación de datos y otras funciones computacionales avanzadas.

El *hardware* de un PLC consta de una unidad de procesamiento (*CPU*) o microcontrolador, memoria y puertos de E/S desde donde las señales pueden ser recibidas y enviadas. Generalmente la recepción de señales se asocia a sensores que indican el estado actual de una etapa del proceso de producción en tanto la emisión de señales se asocia a actores capaces de controlar la siguiente etapa o recalibrar alguna etapa previa.

El *software* de un PLC se divide generalmente en:

1. *Firmware*

Conjunto de instrucciones que el fabricante incluye en la *ROM* y son ejecutadas cada vez que el PLC es iniciado. Análogo al *BIOS* de una computadora personal (*PC*), es ejecutado en el instante del iniciado del sistema y se encarga de comprobar el estado de los diferentes dispositivos antes de pasar el control al sistema operativo o *kernel*.

2. *Kernel*

   Núcleo o parte esencial de un sistema operativo. Provee los servicios básicos del resto del sistema. Es igualmente proveído por el fabricante del PLC, pero a diferencia del *firmware* es posible reemplazarlo por versiones más actualizadas (igualmente distribuidas por el fabricante del PLC).

3. *Programa a ejecutar*

   Código programado por el cliente final (*end user*) del PLC acorde con los requerimientos del proceso industrial en cuestión.

En virtud de que la programación de los PLCs era realizada generalmente por ingenieros, los lenguajes de programación de PLCs se desarrollaron un poco al margen de las tendencias modernas de programación. Ellos han sido diseñados para facilitar las tareas de control de procesos industriales. Como en el *hardware* de los PLCs no existe el nivel de estandarización que existe en las PCs, cada fabricante proporciona entornos de programación diferentes para sus productos, los cuales en ocasiones trabajan con diferentes lenguajes de programación. Esto provoca que los programas desarrollados para familias de PLCs no puedan transportarse a PLCs de otros fabricantes y en ocasiones ni siquiera a otras familias de PLCs de un mismo fabricante.

Para resolver esta polémica se inició el desarrollo del estándar IEC 1131-3 [IEC93] que pretende unificar, al menos a un nivel sintáctico, los principales tipos de lenguajes utilizados en la práctica para la programación de PLCs en el mundo. Para cumplimentar este propósito establece las normativas referentes a los lenguajes de programación para controladores programables. Los mismos se dividen en

dos lenguajes textuales: IL (*Instruction List*) y ST (*Structured Text*), y dos lenguajes gráficos: LD (*Ladder Diagram*) y FBD (*Function Block Diagram*).

También son definidos elementos de SFC (*Sequiential Function Chart*) para estructurar la organización interna de los programas y subprogramas de los controladores programables. Dichos elementos de SFC son concebidos con el fin de ser definidos tanto textual como gráficamente. La definición textual de estos elementos de SFC está orientada exclusivamente a ser insertada en el lenguaje ST como una extensión del mismo, mientras que las definiciones gráficas son apéndices aplicables a cualquiera de los cuatro lenguajes definidos en la Norma. La definición sintáctica de SFC es un potente y expresivo formalismo matemático utilizado en la programación de PLCs. El lenguaje SFC, conjuntamente con los restantes cuatro lenguajes definidos por la Norma conforman la plataforma para el desarrollo de programas para PLCs.

Como es de suponer, de manera similar a que es posible construir entornos o herramientas de desarrollo para la programación de las PCs, así mismo es posible concebir – y de hecho existen – herramientas de desarrollo para la programación de PLCs. Sin embargo, hasta el momento se conocen pocos entornos de de desarrollo de uso comercial, y ninguno de ellos es gratuito. Los entornos comerciales más conocidos son herramientas muy versátiles y útiles para la programación de PLCs, si bien poseen sus propias limitaciones e inconveniencias desde diferentes puntos de vista. En particular, los mismos soportan cada uno la programación sobre los cuatro lenguajes definidos por la Norma, además de contar con su propia herramienta de edición gráfica de programas en SFC.

En correspondencia con los objetivos del presente trabajo, fueron estudiadas con profundidad las herramientas de edición de programas SFC correspondientes a dos de los más modernos y avanzados entornos de desarrollo comerciales para la programación de PLCs: OATs 2.0 [Haa04] y TwinCAT 2.9.0 [Bec04]. En este capítulo no se pretende ofrecer un estudio minucioso y detallado acerca de todas las características de interés de estas herramientas; sin embargo, el estudio de las mismas tuvo gran relevancia en el análisis que se hace más adelante acerca de la representación gráfica de los elementos de SFC, de las operaciones sobre diagramas, así como de las herramientas de edición gráfica de programas SFC. Sí se hace énfasis de forma general sobre las limitaciones más significativas que estas herramientas presentan.

1.2. Especificaciones del Lenguaje SFC

En este epígrafe se brinda una definición formal de SFC y sus elementos, necesarios para la comprensión del resto del documento. Las definiciones que aquí se ofrecen pretenden solamente servir de base para la comprensión de la sintaxis de la definición de programas SFC. En particular, son obviados los aspectos referentes a la semántica de ejecución de los mismos. La mayoría de ellos carecen de interés para el trabajo que se presenta, y sólo son mencionados en caso de ser preciso. Basta mencionar por el momento que un SFC puede ser ejecutado, y que posee una semántica de ejecución en dependencia de su definición sintáctica.

Básicamente, un SFC (o *programa SFC*, como también se denomina indistintamente en lo adelante) está compuesto por tres conjuntos de elementos fundamentales: *Pasos*, *Acciones* y *Transiciones*. La noción de *pasos* de un SFC asemeja a la de estados en un autómata. Un SFC tiene además un único paso inicial. Por su lado las transiciones establecen conexiones entre conjuntos de pasos, e igualmente tienen su contraparte en las transiciones de un autómata. Las transiciones de un SFC se clasifican según su semántica en *transiciones simples* y *transiciones simultáneas*. Una acción es a su vez o bien un programa en uno de los lenguajes de la Norma, o bien un programa SFC mismo. Existen además once *modificadores de acción* diferentes. El par formado por una acción y un modificador que la condiciona conforma una *instancia de acción*. Cada paso de un SFC tiene asociado además un *bloque de acciones*, que no es más que un conjunto de instancias de acciones. La asociación de un modificador de acción a una acción en una instancia condiciona la semántica de ejecución de la acción, del paso a cuyo bloque pertenece la instancia, y del programa en cuestión.

1.2.1. Acciones

Definición 1.2.1.1 (Acción)

Una *acción* está conformada por uno (y sólo uno) de los elementos siguientes:

- S un fragmento de programa en uno de los lenguajes de la norma IEC 1131-3. Una acción de este tipo se denomina *acción ST* (*step transformation*).
- un SFC en cuestión (definido más adelante). Una acción de este tipo se denomina *acción SFC*.

El conjunto de todas las acciones de un SFC será denotado por *A*. Las acciones en un SFC aparecen etiquetadas con un *nombre de acción*.

Definición 1.2.1.2 (Modificador de Acción)
Un *modificador de acción* se define como uno de los elementos del conjunto:
{N, P, P0, P1, S, R, L, D, SD, DS, SL}.

Los modificadores de acción en un SFC son asociados a acciones dentro del mismo, afectando la semántica de ejecución de la acción en cuestión y por consiguiente la semántica de ejecución del SFC mismo.

Definición 1.2.1.3 (Instancia de Acción)
Al par *(m, a)* conformado por un modificador de acción *m* y una acción *a* de un SFC se le denomina *instancia de acción*.

Se denotará por I al conjunto de todas las instancias de acciones que se pueden formar a partir de los elementos del conjunto de acciones *A* de un SFC y los once modificadores de acción definidos anteriormente.

Definición 1.2.1.4 (Bloque de acciones)
Un *bloque de acciones* en un SFC se define como un conjunto (posiblemente vacío) de instancias de acciones.

En otras palabras, un bloque de acciones en un SFC no es más que un subconjunto de I, o si se prefiere, un elemento del conjunto potencia de I.

1.2.2. Pasos

Definición 1.2.2.1 (Paso)
Un *paso* de un SFC es una entidad, a la cual está asociado un bloque de acciones.

Como se mencionó con anterioridad, la noción de *paso* en un programa SFC asemeja a la de estado en un autómata. Los pasos en un SFC aparecen etiquetados de forma única.

1.2.3. Transiciones

Definición 1.2.3.1 (Condición)
Una *condición* es una expresión booleana en uno de los cuatro lenguajes definidos por la Norma.

Definición 1.2.3.2 (Transición Simple)
Una *transición simple* es un terceto (p_0, c, p_f), donde

- p_0, y p_f son pasos, denominados *paso anterior* y *paso posterior* respectivamente en *la transición simple*, y
- c es una condición.

Definición 1.2.3.3 (Transición Simultánea)
Una *transición simultánea* es un terceto (P_0, c, P_f), donde

- P_0, y P_f son conjuntos pasos, denominados *conjunto de pasos anteriores* y *conjunto de pasos posteriores* respectivamente en la *transición simultánea*, y c es una condición.

Los conjuntos P_0, y P_f de una transición simultánea deben contener ambos al menos un elemento, y el cardinal de al menos uno de ellos debe ser mayor que uno.

Se dice que un paso está *involucrado* en (o *forma parte* de) una transición si el paso es un *paso anterior* o *paso posterior* en la misma.

1.2.4. SFCs

A continuación se brinda la definición formal de programa SFC:

Definición 1.2.7 (SFC)

Un *SFC* es un sexteto $S = (P, p_0, A, T_{smp}, T_{sml}, f)$, donde

- P es un conjunto finito no vacío de pasos;
- p_0 perteneciente a P es el *paso inicial*;
- A es un conjunto finito de acciones,
- T_{smp} es un conjunto finito de transiciones simples;
- T_{sml} es un conjunto finito de transiciones simultáneas; y
- $f: P \rightarrow 2^I$ es una función que asigna a cada paso un conjunto de instancias de acciones que conforman el *bloque de acciones* de dicho paso, siendo I el conjunto de todas las instancias de acciones posibles que se pueden conformar a partir de los elementos de A.

Se requiere que el conjunto P de pasos en un SFC contenga al menos un elemento.

1.3. Representación gráfica de los elementos de SFC

En muchas de las ramas del conocimiento es conveniente contar con una representación gráfica para formalismos definidos de manera abstracta. En áreas como la teoría de grafos o la teoría de autómatas, por ejemplo, el esbozo de grafos y autómatas posee un extraordinario valor educativo, apreciativo y demostrativo.

El desarrollo de programas en lenguaje SFC no es la excepción. Contar con una representación gráfica para un SFC y sus elementos permite, en primer lugar, tener una visión general instantánea del programa SFC, de sus elementos, las partes o fragmentos que lo conforman, así como de su semántica. De esta forma, se hace más fácil el diseño, la estructuración y la puesta a punto de un programa SFC. Por otra parte, desde el punto de vista de los entornos de programación, aunque es posible definir un SFC y sus elementos de manera textual mediante la escritura de código fuente en lenguaje ST, la definición gráfica permite utilizar SFC para conformar partes de programas en cualquiera de los restantes lenguajes de la Norma.

El estándar IEC 1131-3 no define con exactitud cómo han de ser representados gráficamente los elementos de un programa SFC. Sin embargo existe, como en otras áreas, una representación gráfica en el plano más o menos convencional por separado – muy útil y simple – para cada uno los elementos que conforman un SFC (pasos, transiciones, condiciones, acciones, instancias de acciones, bloques de acciones, etc). La misma es presentada en el epígrafe 1.3.1. El panorama realmente se complica a la hora de obtener una representación para el SFC como un todo, siguiendo los preceptos utilizados en la representación convencional para elementos aislados.

Todas las consideraciones que aparecen en el resto del documento son relativas a representaciones gráficas en el plano para SFC y sus elementos. De aquí en lo adelante se denominará a la representación gráfica de un SFC como *diagrama SFC*.

1.3.1. Representación gráfica usual

El esbozo comúnmente utilizado para representar los elementos de SFC por separado es sencillo, aún en el caso de las transiciones.

Las acciones ST en un diagrama ni siquiera necesitan representación, y las acciones SFC son representadas precisamente como SFC mismos. Un paso es representado por un rectángulo, en ocasiones etiquetado. Si el bloque de acciones de un paso *p* es no vacío, a la derecha del paso aparece una tabla de dos columnas y tantas filas como instancias de acciones contiene el bloque de acciones asociado a *p*. Cada fila en la tabla corresponde a una instancia; la primera celda corresponde al modificador de acción, y la segunda a la etiqueta de la acción correspondiente. El paso y su respectiva tabla son unidos por un segmento horizontal. La figura 1 muestra la representación de un paso cuya etiqueta es *"p"* y cuyo bloque de acciones asociado es *{(R, acción_B), (P, acción_A), (N, acción_C)}*.

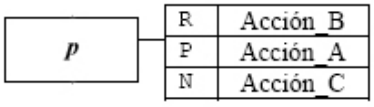

Figura 1. Representación usual para un paso y su correspondiente bloque de acciones.

La representación gráfica de las transiciones de un SFC, sin embargo, ofrece cierta dificultad. Representar transiciones por separado es igual de sencillo como para el resto de los elementos. A las transiciones simples corresponde un segmento vertical que une al rectángulo que representa al paso anterior (colocado siempre en la parte superior) con el rectángulo que representa al paso posterior (dibujado en la parte inferior). La condición en la transición aparece representada como un pequeño segmento horizontal en el punto medio del segmento que une ambos pasos (figura 2).

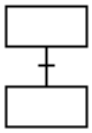

Figura 2. Representación usual para transiciones simples.

Para las transiciones simultáneas, por su lado, existen tres posibilidades diferentes:

1. Que la transición contenga un único paso anterior (figura 3).
2. Que la transición contenga un único paso posterior (figura 4).
3. Que la transición contenga más de un paso anterior y más de un paso posterior (figura 5).

Esta representación usual es obvia y no necesita explicación detallada; es fácil figurar la definición de la transición a partir de su representación gráfica. En todos los casos los pasos anteriores aparecen alineados horizontalmente en la parte superior, así como los pasos posteriores en la parte inferior.

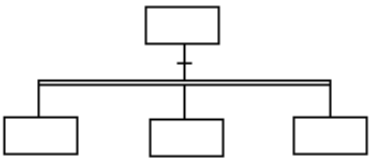

Figura 3. Representación usual para transiciones simultáneas con un único paso anterior.

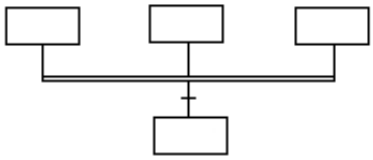

Figura 4. Representación usual para transiciones simultáneas con un único paso posterior.

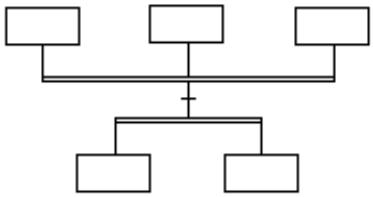

Figura 5. Representación usual para transiciones simultáneas con más de un paso posterior y más de un paso posterior.

La complejidad realmente radica a la hora de representar el SFC como un todo. ¿Es siempre posible esbozar a la vez todos los elementos – en particular las transiciones – de un programa SFC de la manera descrita anteriormente? La respuesta es negativa aún en casos muy sencillos, asumiendo que cada paso es representado en el diagrama sólo una vez – es decir, mediante un único rectángulo. Considérese tan solo el caso de un SFC con pasos *p1, p2* y *p3*, y con las transiciones simples *t1 = (p1, c1, p2), t2 = (p2, c2, p3)* y *t3 = (p3, c3, p1)*, lo cual es perfectamente posible (figura 6). Es posible sin duda imaginar situaciones mucho más complejas aún. En pocas palabras, la representación aislada usual para las transiciones de un SFC no resulta suficiente para la representación de un programa SFC arbitrario. Esto constituye un problema y conduce a la búsqueda de soluciones.

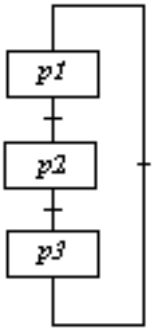

Figura 6. Transiciones con imposibilidad de representación usual en un mismo diagrama SFC.

En lo que sigue se dan algunos pasos hacia la búsqueda de una solución complaciente. Primeramente, nótese que todas las veces son utilizados solamente segmentos horizontales y verticales para la representación de transiciones. Las soluciones para dar solución al problema anterior están comúnmente orientadas a:

- ofrecer una representación clara y legible del programa;
- evitar el uso de segmentos diagonales para representar transiciones o partes de ellas; y
- reducir tanto como sea posible en número la longitud de los segmentos utilizados.

En un primer intento de acercamiento a obtener una representación las características anteriores, se introduce el concepto de *transición disyuntiva divergente y transición disyuntiva convergente* en un diagrama SFC:

Definición 1.3.1.1 (Rama)

Una *rama* es un par *(c, p)*, conformado por una condición c y un paso *p* de un SFC.

Definición 1.3.1.2 (Transición Disyuntiva Divergente)

Una *transición disyuntiva divergente* es un par *(p, R)*, conformado por un paso *p* (denominado *paso anterior* de la transición) y un conjunto de ramas *R*. Al conjunto de pasos $P = \{ x: \exists \underline{c} \ tal\ que\ (c, x) \in R\}$ se le denomina *conjunto de pasos posteriores* de la transición.

Sea $S = (P, p_0, A, T_{smp}, T_{sml}, f)$, un SFC, *p* un paso arbitrario en *P* para el cual el conjunto $T = \{t_1 = (p, c_1, p_1), t_2 = (p, c_2, p_2), \dots , t_n = (p, c_n, p_n)\}$, de todas las transiciones simples en T_{smp} para las cuales *p* es el paso anterior, tiene al menos dos elementos. El conjunto de transiciones *T* es representado en el diagrama de *S* como una única *transición disyuntiva divergente* $t = (p, \{(c_1, p_1), (c_2, p_2), \dots , (c_n, p_n)\})$, conformada por el paso anterior *p* y un conjunto de ramas $\{(c_1, p_1), (c_2, p_2), \dots , (c_n, p_n)\}$.

La figura 7 muestra la representación usual, para dos transiciones disyuntivas divergentes distintas, la cual asemeja mucho a la utilizada para transiciones simultáneas con un único paso anterior.

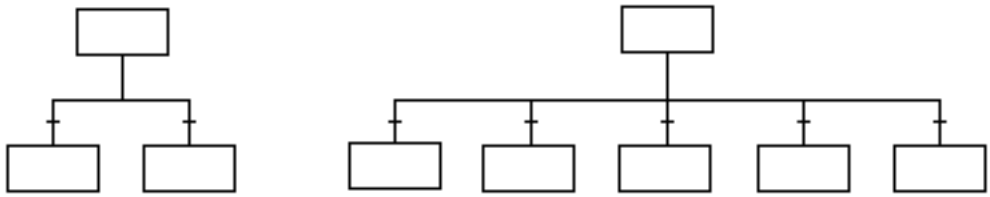

Figura 7. Representación usual para transiciones disyuntivas divergentes.

La semántica de la transición disyuntiva divergente *t* en el diagrama de *S* corresponde a aquella del conjunto de transiciones *T* en *S*. De manera análoga, se introduce el concepto de transición disyuntiva convergente:

Definición 1.3.1.3 (Transición Disyuntiva Convergente)

Una *transición disyuntiva convergente* es un par *(R, p)*, conformado por un conjunto de ramas *R* y un paso *p* (denominado *paso posterior* de la transición). Al conjunto de pasos $P = \{ x: \exists \underline{c} \ tal\ que\ (c, x) \in R\}$ se le denomina *conjunto de pasos anteriores* de la transición.

Sea $S = (P, p_0, A, T_{smp}, T_{sml}, f)$, un SFC, *p* un paso arbitrario en *P* para el cual el conjunto $T = \{t_1 = (p_1, c_1, p), t_2 = (p_2, c_2, p), \ldots, t_n = (p_n, c_n, p)\}$, de todas las transiciones simples en T_{smp} con las que es imposible conformar transiciones disyuntivas divergentes y en las cuales *p* es el paso posterior, tiene al menos dos elementos. El conjunto de transiciones *T* es representado en el diagrama de *S* como una única *transición disyuntiva convergente* $t = (\{(c_1, p_1), (c_2, p_2), \ldots, (c_n, p_n)\}, p)$, conformada por el conjunto de ramas $\{(c_1, p_1), (c_2, p_2), \ldots, (c_n, p_n)\}$ y el paso posterior *p*.

La figura 8 muestra la representación usual, para dos transiciones disyuntivas convergentes distintas, la cual asemeja mucho a la utilizada para transiciones simultáneas con un único paso posterior.

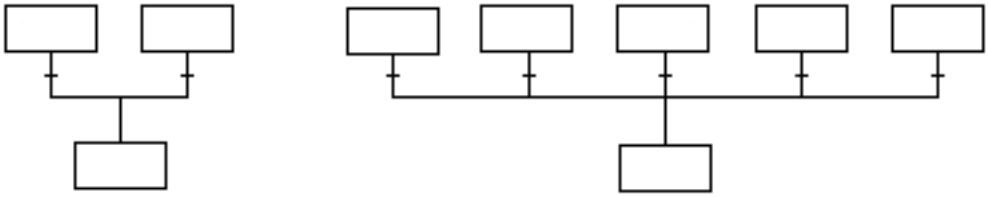

Figura 8. Representación usual para transiciones disyuntivas convergentes.

La semántica de la transición disyuntiva convergente *t* en el diagrama de *S* corresponde a aquella del conjunto de transiciones *T* en *S*.

La introducción de los conceptos de transición disyuntiva divergente y transición disyuntiva convergente en el diagrama en sustitución de conjuntos de transiciones simples está todavía lejos de ser una solución completa y definitiva. Comúnmente y en ausencia de herramientas de edición el problema se aborda tratando de obtener una representación del SFC guiada, además de los anteriores, por los siguientes principios:

- representar los pasos, acciones y bloques de acciones en la forma descrita previamente;
- organizar lo pasos del SFC en el diagrama de manera que las transiciones simultáneas sean representadas en la medida de lo posible de forma usual;
- representar el resto de las transiciones simultáneas de una manera plausible;
- organizar los pasos del SFC en el diagrama de manera que el mayor número posible de transiciones disyuntivas convergentes y divergentes sean representadas en la forma descrita;
- representar el resto de las transiciones disyuntivas convergentes y divergentes utilizando convenientemente segmentos horizontales y verticales para “conectar” los pasos de la transición, de manera que la sintaxis de la misma sea reflejada con claridad y sin ambigüedad;
- organizar los pasos del SFC en el diagrama de manera que el mayor número de transiciones simples que no conforman transiciones disyuntivas divergentes o convergentes sean representadas en la forma usual;

- representar el resto de las transiciones simples convenientemente, utilizando segmentos horizontales y verticales para “conectar” los pasos de la transición, de manera que la sintaxis de la misma sea reflejada con claridad y sin ambigüedad; y
- al seguir cualquiera de los principios anteriores, evitar cualquier superposición o solapamiento de los elementos del diagrama que afecte la claridad, legibilidad y comprensión del mismo.

Nótese que se trata sólo de un conjunto de principios a seguir por su orden y no un algoritmo ni un formalismo para el esbozo de diagramas SFC. En ocasiones puede no resultar una tarea sencilla. Aún más, es imposible predecir la estructura de todo programa arbitrario para PLCs que el programador pueda desarrollar o concebir en el futuro. A continuación se muestra un diagrama SFC sencillo concebidos a la luz de dichos preceptos (figura 9). El mismo corresponde a la representación gráfica del SFC

$S = (P, p_0, A, T_{smp}, T_{sml}, f)$, donde

$P = \{Paso\_1, Paso\_2, Paso\_3, Paso\_4, Paso\_5, Paso\_6, Paso\_7, Paso\_8, Paso\_9, Paso\_10, Paso\_11, Paso\_12, Paso\_13, Paso\_14, Paso\_15, Paso\_16\}$;

$p_0 = Paso\_1$;

$A = \{Acción\_A, Acción\_B, Acción\_C\}$;

$T_{smp} = \{(Paso\_1, c1, Paso\_2), (Paso\_2, c2, Paso\_3), (Paso\_2, c3, Paso\_4), (Paso\_3, c4, Paso\_5), (Paso\_5, c5, Paso3), (Paso\_10, c6, Paso\_11), (Paso\_10, c7, Paso\_16)\}$;

$T_{sml} = \{(\{ Paso\_4\}, c8, \{ Paso\_6, Paso\_7, Paso\_8\}), (\{Paso\_7, Paso\_8\}, c9, \{ Paso\_9\}),$

$(\{Paso\_6, Paso\_9\}, c10, \{ Paso\_10\}), (\{Paso\_11\}, c11, \{ Paso\_12, Paso\_13\}),$

$(\{Paso\_12, Paso\_13\}, c12, \{ Paso\_14, Paso\_15\})\}$; y finalmente

$f = \{(Paso\_1, \Phi), (Paso\_2, \{(N, Acción\_A), (S, Acción\_B)\}), (Paso\_3, \{(R, Acción\_B), (P, Acción\_A), (N, Acción\_C)\}), (Paso\_4, \Phi), (Paso\_5, \Phi), (Paso\_6, \Phi), (Paso\_7, \Phi), (Paso\_8, \Phi), (Paso\_9, \Phi), (Paso\_10, \Phi), (Paso\_11, \Phi), (Paso\_12, \Phi), (Paso\_13, \Phi), (Paso\_14, \Phi), (Paso\_15, \Phi), (Paso\_16, \Phi)\}$.

El paso inicial *Paso_1* es representado a través de un par de rectángulos concéntricos.

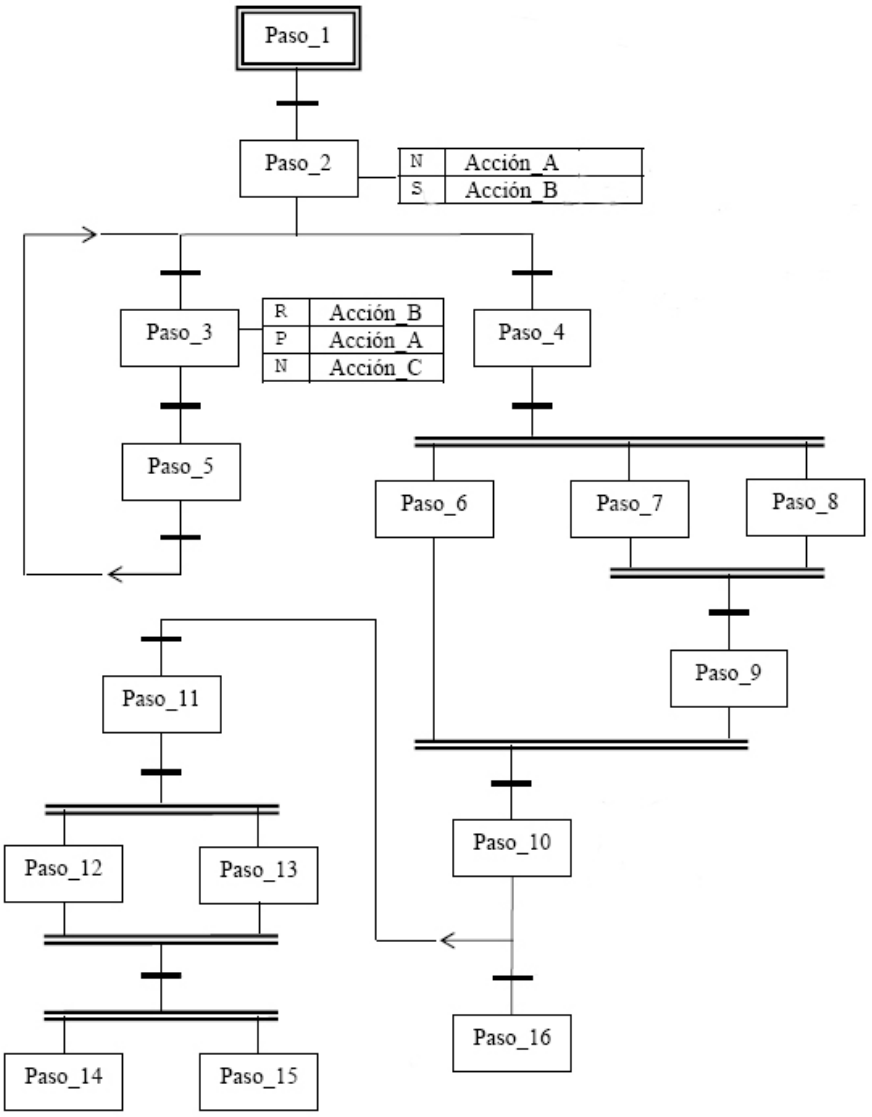

Figura 9. Representación plausible para un programa SFC.

Por último, apuntar que en ocasiones se hace distinción entre las representaciones de las transiciones simultáneas con un único paso anterior, las que poseen un único paso posterior, y aquellas con más de un

paso anterior y más de un paso posterior; definiéndolas como *transiciones simultáneas divergentes*, *transiciones simultáneas convergentes* y *transiciones Rendezvous*, respectivamente. Desde el punto de vista de la semántica de las transiciones, tal distinción resulta irrelevante.

1.3.2. Representación gráfica y herramientas de edición

Existe una diferencia considerable entre encontrar o proponer una representación adecuada sobre una hoja de papel para un SFC previamente concebido, y el editar o desarrollar de manera incremental un diagrama SFC, a través de una herramienta de edición con tal fin.

Hay dos aspectos fundamentales a considerar a la hora de enfrentarse al desarrollo de una herramienta de edición de diagramas SFC. El primero de ellos tiene que ver con la estrategia utilizada por el editor para representar los elementos de SFC; en particular la disposición y representación de pasos y transiciones en el diagrama. Un segundo aspecto no menos importante consiste en establecer y dar soporte a un conjunto de operaciones que permita al usuario final de la herramienta – el programador de PLCs – la edición o desarrollo incremental del diagrama y sus partes, así como su modificación. Entre ambos existe una estrecha relación y dependencia mutua; ninguno de ellos puede ser considerado al margen del otro. Las operaciones definidas y soportadas dependen en gran medida de la estrategia de representación, así como la estrategia de representación debe tener en cuenta las operaciones sobre el diagrama. Este epígrafe pretende abordar algunas cuestiones referentes al primero de ellos.

¿Cómo establecer una estrategia conveniente posible para la representación de un diagrama SFC en una herramienta de edición? Quizás el lector ya se haya percatado de que, sin duda alguna, la cuestión principal tiene que ver con la *disposición de los pasos* en el plano. De ello depende el crucial problema de la representación gráfica de las transiciones del diagrama, y un problema menor asociado al solapamiento de los elementos representados. El primero de ellos reviste el mayor interés.

La cuestión clave es la siguiente: *¿debería la herramienta conceder al usuario o programador de PLCs control total sobre la disposición de los pasos en el diagrama, o intervenir de algún modo e imponer ciertas condiciones o restricciones acerca de la locación de los mismos?* Considérense ambas alternativas:

A. *La herramienta de edición impone restricciones o condiciones acerca de la disposición de los pasos en el diagrama*. El editor toma partida en el control de la ubicación de los pasos y las transiciones, en relación con las operaciones definidas y soportadas, y en correspondencia con las operaciones solicitadas por el usuario durante el proceso de edición. El usuario de la herramienta participa pero no tiene control absoluto sobre la ubicación de los pasos. Esta variante requiere la definición de dos estrategias; una primera para la disposición de los pasos, y una segunda para la disposición de las transiciones. La responsabilidad de la locación de ambos, pasos y transiciones, recae sobre la herramienta, en combinación por supuesto con las operaciones realizadas por el usuario sobre el diagrama en edición y el estado del mismo.

Las principales ventajas y desventajas de esta alternativa pueden ser enumeradas como sigue:

+ libera al programador parcialmente y en apariencia del problema de disponer adecuadamente los pasos y las transiciones en el diagrama;

+ libera al programador del problema menor acerca de la superposición de los elementos del diagrama;

- priva al programador de PLCs de la posibilidad de tener control total sobre la colocación de los pasos en el diagrama, de modo que queda limitada la flexibilidad en cuanto a la organización y estructuración del programa; y por ende

- conduce a editores de SFC rígidos;

- exige del programador de PLCs una gran previsión acerca del aspecto final del programa para lograr una disposición de estados y transiciones cercana a la deseada;

- en ocasiones fuerza a los editores a ofrecer operaciones obscuras sobre el diagrama y que dejan al mismo en un estado indefinido o no conforme al estándar; y

- conduce a representaciones para los diagramas que muchas veces están lejos de asemejar la representación usual.

Las ventajas aparecen precedidas por un “+” y las desventajas por un “-”. Hasta qué punto es conveniente dejar ofrecer al programador control total sobre la disposición de los pasos es un tema

a discutir. Luego de un análisis profundo, se llegó a la conclusión de que estas ventajas clasifican como ventajas menores, y que las desventajas señaladas constituyen grandes desventajas.

A pesar de los inconvenientes señalados, ésta resultó ser la variante utilizada por las herramientas de edición de diagramas SFC en los entornos comerciales para la programación de PLCs estudiados (OATs y TwinCAT), con sus respectivos inconvenientes. Es muy posible que tal decisión haya sido influenciada de alguna forma por la selección previa de las herramientas usadas para el desarrollo de las mismas. Para una mejor comprensión de las ideas expuestas, se recomienda consultar las herramientas mencionadas y sus correspondientes documentaciones ([Haa04] y [Bec04]).

B. *La herramienta concede al programador el control total sobre la disposición de los pasos en el diagrama*. Los pasos del SFC pueden ser colocados y movidos libremente a través del diagrama. En tal caso, sólo se necesita de una buena estrategia para la representación de las transiciones; el programador se encuentra a cargo del resto. Tal estrategia debe tomar en cuenta que, si el programador escoge una disposición para los pasos involucrados en una transición *t* "próxima" a la disposición esperada para la representación usual de *t*, entonces la representación ofrecida para *t* debe ser igualmente "próxima" a la representación usual; de lo contrario, una alternativa apropiada debe ser brindada.

   Las principales ventajas y desventajas de esta alternativa pueden ser enumeradas como sigue:

   + ofrece al programador de PLCs libertad absoluta sobre la disposición de los pasos en el diagrama;

   + da la posibilidad de construir editores de SFC más flexibles en cuanto a la organización y estructuración del programa;

   + abre la posibilidad de obtener representaciones para los diagramas que estén "próximas" a la representación usual.

   - recae sobre el programador de PLCs la responsabilidad de buscar una locación conveniente para los pasos del diagrama;

- corresponde programador de PLCs lidiar con el problema menor acerca de la superposición de los elementos del diagrama.

Como anteriormente, las ventajas aparecen precedidas por un “+” y las desventajas por un “-”. Luego de ser sometidas a consideración, se decidió que las ventajas señaladas constituyen grandes ventajas, y que es posible lidiar con las desventajas sin mayor dificultad. En consecuencia, es ésta la variante que se propone para el desarrollo posterior de una herramienta de edición de diagramas SFC, y fue la utilizada para la concepción del prototipo iniciado.

1.3.2.1. Representación gráfica en OATs

La herramienta de edición gráfica de programas SFC en el entorno de desarrollo OATs, como se mencionó con anterioridad, impone restricciones acerca de la disposición de los pasos y además de las transiciones en el diagrama. En el epígrafe 1.4.1 se describen brevemente la forma en que son realizadas las operaciones de inserción de tales elementos en el diagrama. La imposición de tales restricciones acarrea consigo las limitaciones mencionadas en el epígrafe 1.3.2. Al respecto, lo más significativo tiene que ver con la rigidez en cuanto a la organización del programa y la legibilidad del mismo.

La figura 10 muestra un diagrama SFC muy sencillo editado en OATs. Para aseverar lo dicho hasta el momento, tan solo trate el lector sin conocimiento de la herramienta de figurar la definición sintáctica del programa SFC asociado.

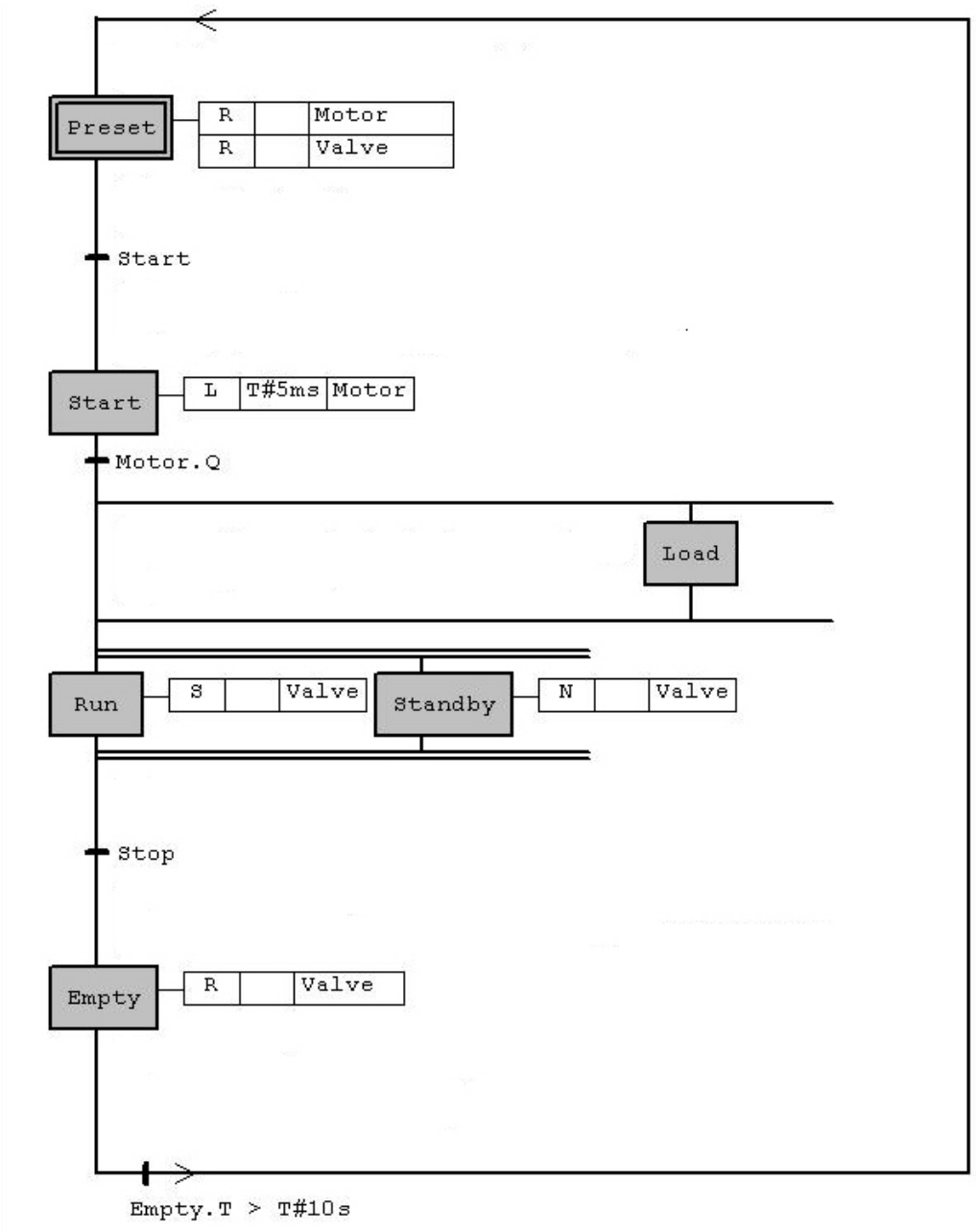

Figura 10. Diagrama SFC en el editor gráfico de OATs.

1.3.2.2. Representación gráfica en TwinCAT

Al igual que OATs, la herramienta de edición gráfica de programas SFC en el entorno de desarrollo TwinCAT impone restricciones acerca de la disposición de los pasos y además de las transiciones en el diagrama. En el epígrafe 1.4.1 se describen brevemente la forma en que son realizadas las operaciones de inserción de tales elementos en el diagrama. Una vez más, la imposición de tales restricciones acarrea consigo las limitaciones mencionadas en el epígrafe 1.3.2. Al respecto, lo más significativo tiene que ver con la rigidez en cuanto a la organización del programa y la legibilidad del mismo.

La figura 11 muestra un diagrama SFC muy simple editado en TwinCAT, el cual con optimismo no está aún completamente desarrollado. Se incita al lector a intentar figurar la definición sintáctica conforme al estándar del programa SFC asociado.

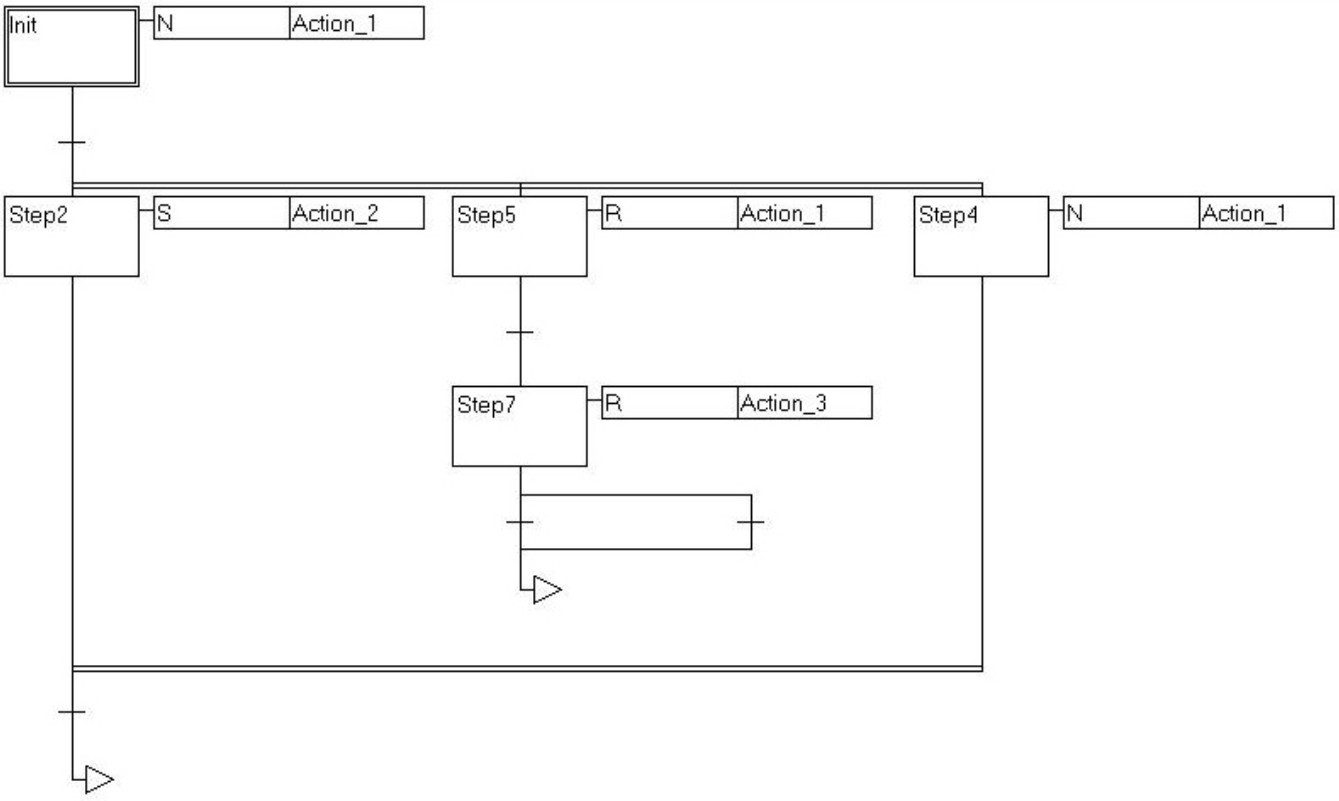

Figura 11. Diagrama SFC en el editor gráfico de TwinCAT.

1.4. Operaciones sobre un diagrama SFC

Como se señaló con anterioridad, el propósito de una herramienta de edición de diagramas SFC no consiste en tomar una definición matemática $S = (P, p_0, A, T_{smp}, T_{sml}, f)$ de un SFC y encontrar u ofrecer una representación gráfica del mismo. Se trata, por el contrario, de construir y desarrollar incrementalmente el diagrama correspondiente a un SFC deseado. Para ello, la herramienta debe definir y dar soporte a un *conjunto de operaciones sobre un diagrama SFC*. El estándar IEC 1131-3 no establece nada concerniente a la construcción incremental u operaciones sobre un diagrama SFC. Cada herramienta construida debe brindar sus propias alternativas.

Básicamente, la construcción incremental de un diagrama se logra a través operaciones de inserción y eliminación de los elementos que lo conforman (pasos, acciones y transiciones), operaciones sobre bloques de acción (operaciones sobre conjuntos), y de un conjunto de operaciones complementarias menos interesantes sobre el diagrama mismo y sus elementos. Definir y dar soporte a tal conjunto de operaciones a la hora de desarrollar editor de SFC no constituye una tarea trivial.

En las herramientas comerciales para la edición de diagramas SFC estudiadas, ocurre que las operaciones provistas por la herramienta y su semántica sobre el diagrama están fuertemente influenciadas por la estrategia de representación gráfica en una forma desafortunada. Lo mismo sucede en sentido contrario. Por otro lado, se definen operaciones que en ocasiones dejan al diagrama en un estado indefinido o no válido; además de ello, se introducen conceptos no universales específicos de la aplicación para definir operaciones sobre el diagrama.

En una herramienta ideal para el desarrollo de diagramas SFC:

- se establece un conjunto de operaciones con semántica bien definida sobre el diagrama, fáciles de realizar;
- no se introducen conceptos no universales para definir operaciones sobre el diagrama;
- no se definen operaciones sin sentido o que puedan dejar al diagrama en un estado indefinido o no válido (esto es, las operaciones sobre el diagrama son cerradas);

- la definición de las operaciones sobre el diagrama y su semántica no son influenciadas por la estrategia de representación gráfica utilizada por la herramienta;

Tómese por analogía las operaciones de inserción y eliminación de aristas y vértices sobre un grafo. Para insertar una nueva arista en el grafo, ambos vértices unidos por la misma han de estar presentes en el grafo. De lo contrario, sería necesario definir la semántica de inserción de aristas con vértices inexistentes en el grafo – lo cual sería indeseado. Luego de tal operación, el grafo quedaría en un estado no válido o no definido. Algo similar ocurre con las operaciones de inserción de transiciones en un diagrama SFC. Como otra analogía, considérese la eliminación de un vértice en un grafo. Esta es una operación interesante, pues implica la eliminación también de todas las aristas asociadas al vértice, lo cual garantiza la obtención de un grafo válido después de la operación. Análogamente, la semántica de eliminación de un paso en un diagrama SFC debe incluir operaciones sobre las transiciones en las cuales el paso está involucrado para garantizar la integridad del diagrama. Esto no significa, sin embargo, que la eliminación de todas las transiciones en las que el paso está involucrado es necesariamente la semántica más apropiada. Resumido en pocas palabras, las operaciones sobre un diagrama SFC deben de ser cerradas.

Las herramientas de edición de diagramas SFC correspondientes a los sistemas comerciales conocidos fallan en definir un conjunto de operaciones sobre el diagrama que se ajuste a las características mencionadas anteriormente. Tal es el caso de los sistemas OATs y TwinCAT. Ello constituye otra de las limitaciones o inconvenientes fundamentales de dichas herramientas.

1.4.1. Operaciones sobre el diagrama en OATs

Para ilustrar el estilo de las operaciones sobre el diagrama en OATs, se describe brevemente a continuación la manera en que las operaciones de inserción de pasos y de inserción de transiciones simples son concebidas.

A la inserción de pasos en el diagrama se da soporte de la siguiente manera: dado un estado determinado del diagrama, y tras la solicitud del usuario de la herramienta de insertar un nuevo paso en el diagrama, la herramienta ofrece un conjunto de posiciones dentro del diagrama en las cuales podría colocarse el paso.

Una vez que el usuario selecciona la posición deseada, la herramienta realiza la expansión correspondiente al diagrama. La figura 12 muestra, dado el estado del diagrama en la figura 10, el conjunto de alternativas para la posición de un nuevo paso a insertar luego de la solicitud hecha por el usuario. Las diferentes posiciones son señaladas mediante pequeños cuadrados.

Nótese sobre todas las cosas que este procedimiento en nada asemeja al utilizado históricamente para esbozar un diagrama SFC. Lo más natural y parecido a lo usual sería permitir al usuario ubicar los pasos a su antojo en cualquier posición del diagrama, para luego definir o modificar el conjunto de transiciones existentes.

La inserción de transiciones simples es manejada de manera similar. La figura 13 muestra, dado el estado del diagrama en la figura 10, el conjunto de alternativas para la posición de una nueva transición simple a insertar luego de la solicitud hecha por el usuario. Las diferentes posiciones son señaladas también mediante pequeños cuadrados.

Baste mencionar, por ejemplo, que en las posiciones en las que aparecen cuadrados que no están encima de un segmento correspondiente a una transición, es posible insertar transiciones simples para las cuales el paso anterior y el paso posterior no existen en el diagrama. Hay situaciones más oscuras, por ejemplo, en las que se permite la inserción “consecutiva” de varios segmentos correspondientes a transiciones simples.

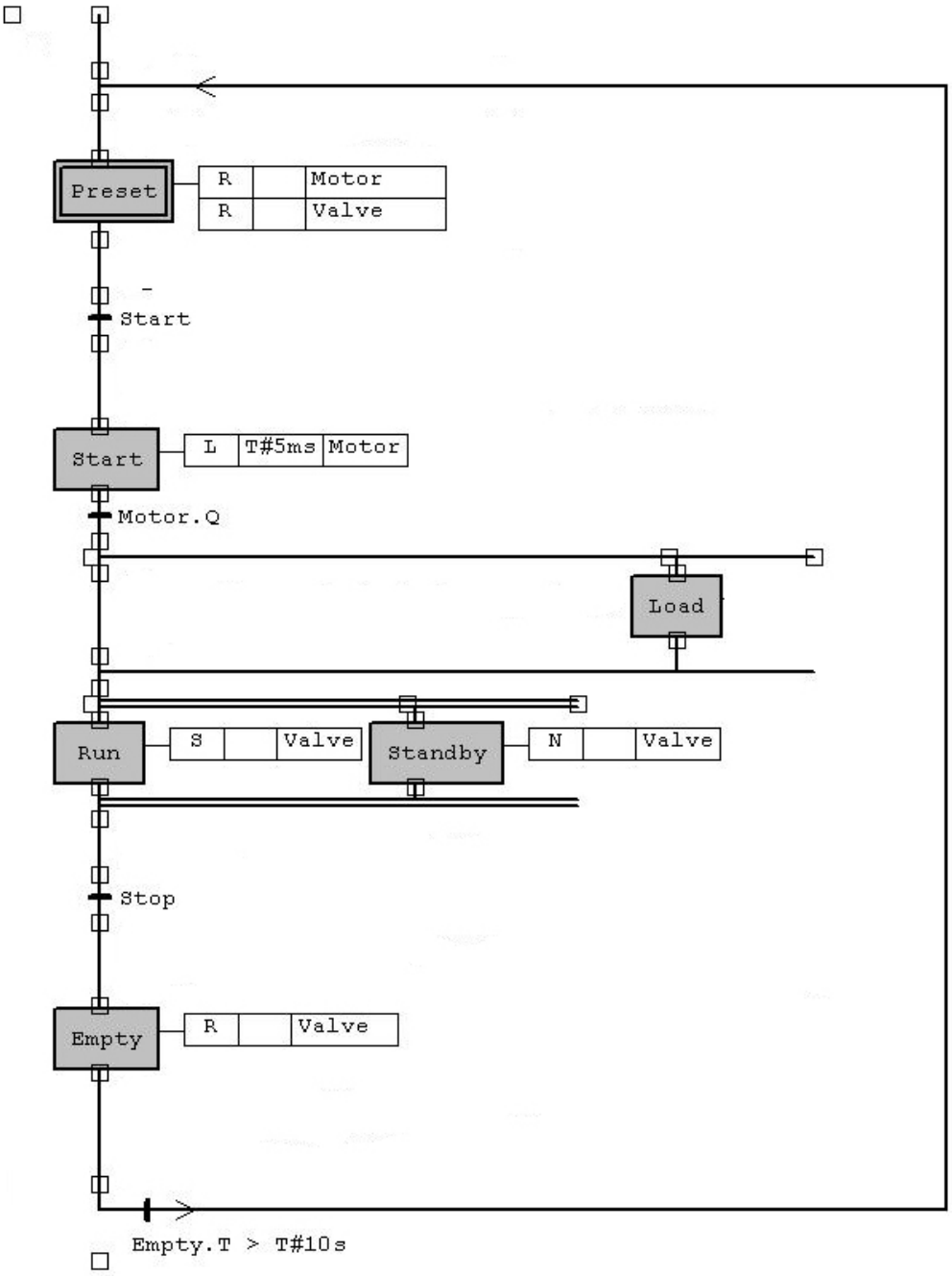

Figura 12. Alternativas tras la solicitud de inserción de un nuevo paso en OATs.

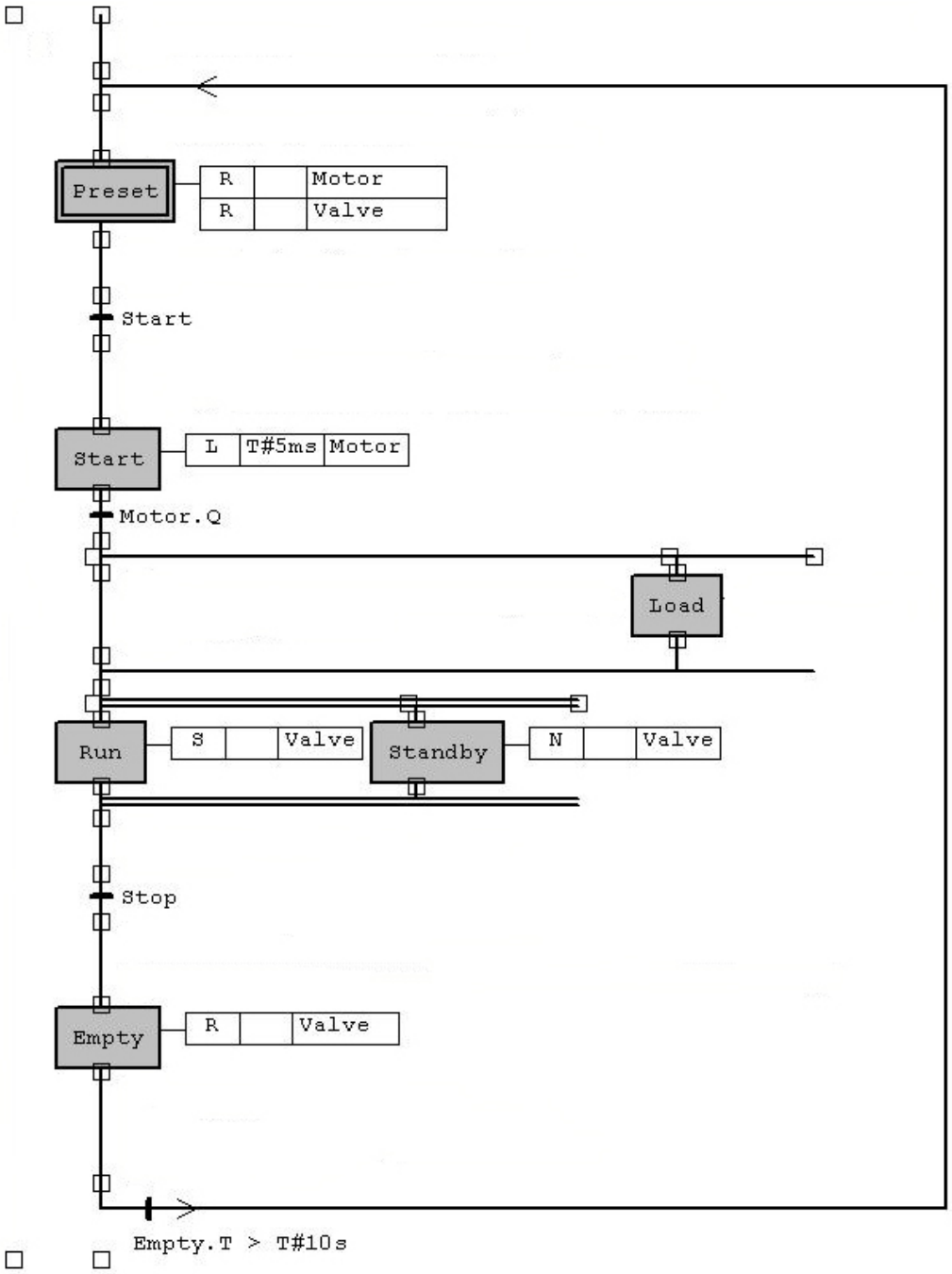

Figura 13. Alternativas tras la solicitud de inserción de una nueva transición simple en OATs.

1.4.2. Operaciones sobre el diagrama en TwinCAT

Las operaciones sobre el diagrama en TwinCAT son en esencia muy similares a las operaciones en OATs. De hecho, ambos sistemas guardan mucho en común tanto en cuanto a las operaciones sobre el diagrama como a la representación gráfica de SFC y sus elementos.

En el caso de TwinCAT, el usuario debe seleccionar previamente el paso, transición o fragmento de esta sobre la cual desea operar, y luego solicitar la operación deseada. El conjunto de operaciones disponibles tras una selección dependen del tipo de objeto seleccionado y del estado actual del diagrama.

1.5. Conclusiones

En este capítulo se incursionó sobre la definición sintáctica de SFC y sus elementos, según lo establecido en la norma IEC 1131-3. Asimismo, se discutieron los problemas fundamentales asociados a la construcción de una herramienta para la edición gráfica de programas SFC; en particular se abordaron cuestiones referentes a la representación gráfica de los elementos de SFC, y a las operaciones sobre un diagrama SFC. Además de ello, se ofreció una panorámica acerca de las características de las herramientas de edición gráfica de programas SFC correspondientes a los entornos de desarrollo comerciales existentes, con énfasis en las limitaciones que las mismas poseen.

Los elementos expuestos pretenden facilitar la comprensión del capítulo siguiente, donde se presentan y someten a discusión herramientas de desarrollo, técnicas de programación y algoritmos varios para ser utilizados en el desarrollo de una herramienta de edición de diagramas SFC que supere las limitaciones mencionadas.

Capítulo2. Descripción de la solución propuesta

Introducción

El presente capítulo recoge los aportes fundamentales de esta tesis. Los mismos establecen las bases para el desarrollo posterior de una herramienta para la edición gráfica de programas en lenguaje SFC fuera de los inconvenientes que poseen las herramientas comerciales conocidas. Varios de los epígrafes son dedicados a detallar la solución a un problema específico relacionado con la edición gráfica de programas en lenguaje SFC.

Inicialmente se hace un compendio de las características y funcionalidades ideales para una herramienta de edición gráfica de programas SFC. Luego, se proponen herramientas de desarrollo apropiadas para construir la herramienta deseada. Posteriormente es presentado y sometido a discusión un conjunto de clases para representar los conceptos fundamentales en el dominio de la aplicación. A continuación, aparecen algunos algoritmos diseñados para la implementación de operaciones sobre un diagrama SFC, así como un conjunto de algoritmos formales para la representación geométrica de los elementos de SFC. Por último se evalúan los avances realizados en el desarrollo de un prototipo funcional.

2.1. Características y funcionalidades propuestas para una herramienta de edición gráfica de programas SFC

Una vez estudiados y analizados los elementos fundamentales y las especificaciones de *Sequential Function Chart*, los aspectos relacionados a la representación gráfica de diagramas SFC, las operaciones sobre los mismos y las herramientas de edición de diagramas correspondientes a los entornos de programación de PLC existentes, se procedió en primera instancia a proponer un conjunto importante de características y funcionalidades que debe tener un buen editor de diagramas SFC.

Dichas características corresponden a cuestiones como las operaciones definidas sobre el diagrama, la representación gráfica de los elementos de SFC, la navegación sobre el diagrama, entre otras. Las mismas son claves en el diseño general de una herramienta de edición de diagramas SFC y del prototipo

funcional, y tuvieron que ver de manera significativa en la selección de las herramientas de desarrollo propuestas y utilizadas en la construcción del prototipo, así como en el diseño de clases.

Una buena herramienta de edición de programas SFC y el prototipo funcional debe poseer las siguientes características:

1. Establecer un conjunto completo de operaciones con semántica bien definida sobre el diagrama.
2. No introducir conceptos no universales para la definición de operaciones sobre el diagrama.
3. No definir operaciones sin sentido o que puedan dejar al diagrama en un estado indefinido o no válido (las operaciones sobre el diagrama deben ser cerradas);
4. La definición de las operaciones sobre el diagrama y su semántica no debe influenciar (ni ser influenciada por) la estrategia de representación gráfica de los elementos de SFC utilizada por la herramienta.
5. Conceder al programador de PLCs (usuario final de la herramienta) control total sobre la disposición de los pasos en el diagrama.
6. Brindar una representación gráfica coherente y comprensible para los elementos en el diagrama, en la medida de lo posible próxima a la representación usual. La sintaxis del programa en general y de cada una de sus partes y elementos, debe ser claramente distinguible sin ambigüedad.
7. Ofrecer la posibilidad de ocultar total o parcialmente diferentes elementos en el diagrama (por ejemplo, omitir la representación de los bloques de acción).
8. Permitir una fácil navegación sobre el diagrama, incluyendo operaciones de acercamiento/alejamiento (zoom in/out), de manera que cualquiera de las partes del diagrama – en particular el diagrama como un todo – pueda ser completamente visualizada en pantalla.
9. Permitir la edición simultánea de varios diagramas.

Las cuatro primeras características tienen gran importancia desde el punto de vista conceptual. Una quinta ofrece un alto grado de libertad en cuanto a la disposición de los elementos en el diagrama y la organización y estructuración del programa. Las características seis y siete tienen gran importancia respecto a la comprensión y legibilidad del diagrama. Las dos últimas poseen una gran relevancia desde el punto de vista educativo, y del análisis y comparación de los programas.

2.2. Selección del lenguaje de programación y las herramientas de desarrollo

La selección del lenguaje de programación y las herramientas de desarrollo utilizadas para la construcción del prototipo funcional – y que a su vez se proponen para ser usadas en el desarrollo posterior de un editor de diagramas SFC – fue influenciada por:

- la plataforma sobre la cual se decidió desarrollar el prototipo (Windows);
- el tipo de aplicación a desarrollar; y
- las características propuestas para la herramienta, enunciadas en el epígrafe anterior (ver epígrafe 2.1).

En realidad no se hizo un estudio completo de todas las herramientas de desarrollo posibles a utilizar sobre la plataforma Windows, sino más bien se trató solamente de seleccionar y proponer un conjunto de ellas adecuado, que resultaran suficientes y convenientes para el desarrollo de una aplicación con las características buscadas.

2.2.1. Lenguaje de programación

El tipo de aplicación a desarrollar hace atractivas las características y facilidades brindadas por un lenguaje de programación de alto nivel con soporte para la programación orientada a objetos.

Para el desarrollo del prototipo se escogió el Lenguaje de Programación C++. A favor de la elección de C++ podría abogarse lo siguiente:

- Es un Lenguaje de Programación de propósito general exitoso, de uso ampliamente extendido.
- Posee una Librería Estándar muy bien diseñada y concebida, con facilidades muy convenientes para utilizar en el desarrollo de la aplicación.
- Tanto el Lenguaje como su Librería Estándar han sido excelentemente documentadas por el autor, facilitando la comprensión y el uso apropiado de los mismos.
- La Librería MFC (Microsoft Foundation Classes Library [MSD05]), propuesta y utilizada como herramienta de desarrollo, fue concebida en C++.

2.2.2. Herramientas de desarrollo

Librería MFC de Microsoft Visual Studio.Net 2005

La MFC es una poderosa herramienta para el desarrollo de aplicaciones sobre Windows, en particular para el desarrollo de aplicaciones de escritorio. La razón clave por la cual MFC fue escogida es que la misma ofrece “plantillas” o “modelos” de aplicación (*application frameworks*, [Str97]) muy seductoras, como la plantilla de aplicación MDI (*Multiple Document Interface*, [MSD05]). Las plantillas son aplicaciones que brindan una funcionalidad y estructura básica a la cual es posible agregar código específico. En particular la MDI brinda soporte de manera muy conveniente para el trabajo con múltiples ventanas en la aplicación. Sin embargo, fue necesario lidiar con los siguientes inconvenientes:

- el diseño de la MFC posee cierta complejidad;
- envuelve un voluminoso número de conceptos propios que resultan novedosos al programador sin experiencia en el trabajo con la MFC; por consiguiente
- toma tiempo al programador novicio la familiarización con la Librería.

Interfaz para gráficos OpenGL

Las herramientas de edición de diagramas SFC comerciales que fueron estudiadas, utilizan controles y primitivas de Windows para lograr representar gráficamente el diagrama SFC y sus elementos en pantalla.

Esta opción, sin embargo, posee sus inconvenientes y no resulta muy atractiva. La mayor parte de las limitaciones señaladas en el Capítulo 1 para tales herramientas son consecuencia precisamente del hecho de evadir o ignorar la utilización de una interfaz para gráficos adecuada.

Se decidió que lo más conveniente sería representar gráficamente el diagrama y sus elementos utilizando la interfaz para gráficos OpenGL ([DavNei], [San96]). Las principales ventajas radican en que OpenGL:

- es una herramienta ampliamente difundida y utilizada para el desarrollo de aplicaciones para el diseño asistido por computadoras, o aplicaciones CAD *(Computer Aided Design applications*);
- ofrece flexibilidad y versatilidad para la representación gráfica de los objetos;
- brinda la posibilidad de ocultar parcial o totalmente elementos en el esbozo del diagrama;
- ofrece la posibilidad de programar una navegación fácil y conveniente sobre el diagrama; y
- ofrece la oportunidad de escribir código portable a otras plataformas.

No obstante, durante el desarrollo del prototipo fue necesario lidiar con los siguientes inconvenientes:

- OpenGL brinda operaciones a un nivel relativamente bajo;
- no brinda soporte directo para algunas de las características y funcionalidades esperadas de la herramienta; por consiguiente
- recae sobre el programador (desarrollador de la herramienta) la dificultad de implementar operaciones como la selección de los objetos representados, el acercamiento y alejamiento, y la navegación a través del diagrama, a partir de operaciones primitivas de OpenGL.

Existe una herramienta con facilidades a un nivel más alto construida sobre OpenGL, llamada “Open Inventor” [Mer07], que soporta un diseño orientado a objetos para la modelación de los principales conceptos relativos a la modelación de gráficos en tercera dimensión. Sin embargo, tal herramienta no se encuentra disponible para su uso libre.

2.2.3. Uso del lenguaje y las herramientas de desarrollo

La intención de uso para cada una de las herramientas seleccionadas conjuntamente con el Lenguaje de Programación es la siguiente:

- Modelar los conceptos asociados a SFC, sus elementos y las operaciones sobre los mismos a través de un diseño de clases en C++, en conformidad con lo establecido por el estándar IEC-1131.
- Extender la funcionalidad ofrecida por la *plantilla de aplicación MDI* de la MFC, de manera que los *documentos (documents)* no sean más que los diagramas modelados a través del diseño de clases, y las *vistas (views)* muestren la representación gráfica del estado actual de los *documentos* o diagramas, utilizando las primitivas de OpenGL. Extender además la interfaz gráfica de usuario de manera que permita realizar operaciones de edición del diagrama sobre la *vista*, con sus correspondientes efectos sobre el estado del *documento*. Para obtener una panorámica acerca de la MFC, las aplicaciones MDI (*Multiple Document Interface application framework*) y la arquitectura Documento/Vista (*Document/View architecture*), consultar [MSD05].
- Implementar las operaciones de acercamiento/alejamiento, y en general la navegación sobre el diagrama a partir de las operaciones primitivas de OpenGL.

2.3. Diseño de clases

Una de las contribuciones de peso de esta tesis radica en el diseño e implementación de un conjunto apropiado de clases para la aplicación (*application classes*, [Str97]). "*La noción fundamental del diseño y la programación orientada a objetos es que el programa es un modelo de ciertos aspectos de la realidad. Las clases en el programa representan los conceptos fundamentales en la aplicación, y en particular, los conceptos fundamentales en la 'realidad' que se modela. Los objetos y artefactos de implementación son representados a través de objetos de tales clases.*" [1][Str97] .

[1] Stroustrup, Bjarne: "The C++ Programming Languaje", Third Edition, 1997.

Diseñar – y en adición implementar – un conjunto de clases adecuado no siempre es una tarea trivial, un diseño de clases puede verse dificultado, entre otras razones, por las siguientes:

- En la práctica existen conceptos de las más disímiles naturalezas.
- En ocasiones se necesita además la habilidad de conceptualizar convenientemente la realidad.
- En ocasiones es importante también representar la relación entre los conceptos. En un lenguaje de programación de propósito general, es imposible dar soporte directo para cualquier relación arbitraria entre clases.
- Usualmente existen varias alternativas de diseño, con sus respectivas ventajas y desventajas.
- Existen clases para las cuales no resulta trivial la definición/implementación de una interfaz conveniente.
- Existen clases para las cuales no resulta trivial escoger una representación conveniente.
- Existen clases para las cuales no resulta trivial establecer y mantener una invariante conveniente.
- En ocasiones el diseño de clases se ve influenciado por detalles de implementación.
- Existe un número apreciable de técnicas avanzadas de diseño; en muchas ocasiones algunas resultan de gran utilidad y su uso resulta determinante.
- La habilidad de diseñar clases, al igual que otras actividades, requiere de tiempo, práctica y experiencia para ser desarrollada.

Por consiguiente, tal actividad no ha de ser considerada como una actividad mecánica, ni ha de ser subestimada. El diseño de un conjunto de clases para el desarrollo de una herramienta de edición de diagramas SFC no es de los más complejos que se puedan concebir en tal universo, pero tampoco resulta ser de aquellos más triviales.

El propósito de este epígrafe no es sólo presentar un diseño de clases, sino también

- analizar otras alternativas de diseño;
- definir invariantes apropiadas para cada una de las clases; y
- explicar el uso correcto de las clases en casos necesarios.

En ocasiones se tiende a considerar completamente documentado un diseño de clases, mediante la presentación de un mero listado de los miembros de las clases, divididos en categorías, y la clasificación de las relaciones existentes entre las mismas. A menudo, la definición de invariantes adecuadas para algunas de las clases es un aspecto clave del diseño. Existen clases además para las cuales es imprescindible explicar detalladamente el uso correcto y semántica de su interfaz. Por otro lado, el análisis de otras alternativas de diseño – de ser aplicable – es un excelente ejercicio para comprender el por qué de la elección de la variante seleccionada; y algunas veces deja abierta la posibilidad de mejorías en el diseño.

Los conceptos de SFC y sus principales elementos – pasos, acciones y transiciones – constituyen conceptos claves en el dominio de la aplicación. Los mismos guardan una estrecha relación entre sí, de modo que las clases usadas para representar los mismos no pueden ser diseñadas o concebidas por separado, sino como un todo.

La interfaz de las principales clases que diseñadas son presentadas a continuación por partes; por ese orden se describen: tipos miembros de la clase, iteradores, constructores/destructor, y operaciones. La metodología utilizada para la presentación y discusión de las clases propuestas es muy similar a la utilizada para la documentación de la Librería Estándar de C++ (ver [Str97]).

2.3.1. Acciones

Fueron definidas las clases *STAction* y *SFCAction* para representar la noción de acción ST y acción SFC respectivamente. Se definió además una clase *Action* para representar el concepto común que comparten ambas clases. La clase *Action* sirve como base para las clases *STAction* y *SFCAction*, y ofrece servicios comunes a ambas clases.

2.3.1.1 Clase *Action*

La interfaz de la clase Action es bastante pequeña; la misma es presentada a continuación por partes.

Iteradores

La semántica de la eliminación de una acción ***a*** de un SFC incluye la actualización de todos los bloques de acción en los que ***a*** aparece instanciada. Recorrer exhaustivamente todos los pasos del SFC tras las instancias de ***a*** en todos los bloques de acciones sería en algunos casos muy ineficiente, en particular cuando la acción aparece instanciada en un número relativamente pequeño de bloques de acción. Una mejor alternativa – utilizada en el diseño propuesto – sería mantener en el diagrama SFC, para cada acción, el conjunto de pasos en los que la acción aparece instanciada en el bloque de acciones. Como las clases *SFC* (presentada más adelante) y *Action* son concebidas a la par, tal conjunto de pasos puede ser mantenido en – aunque no por – la acción misma. Ello trae la necesidad de acceder o *iterar* en determinadas ocasiones sobre el conjunto de pasos en los que la acción es instanciada:

```
class Action{
// ...
public:
        // tipos miembros
        typedef tipo_dependiente_de_la_implementación  step_iterator;

        // iteradores
        step_iterator steps_begin() const;
        step_iterator steps_end() const;
// ...
};
```

La iteración o el acceso a los pasos en cuyos bloques una acción aparece instanciada se hace a través de *iteradores* [Str97] . La función *steps_begin()* retorna un *iterador* al primer paso de la secuencia, mientras que *steps_end()* retorna un *iterador* a la posición siguiente al último elemento de la secuencia de pasos. Con ambos iteradores es suficiente para recorrer la secuencia completa. El tipo exacto de los iteradores retornados por estas funciones depende de la representación y la implementación de la clase *Action*. Definir un nombre de tipo estándar para el tipo de iteradores utilizados permite modificar la representación e implementación de la clase *Action* – en particular el tipo de iterador – sin necesidad de cambios en la interfaz de la clase, ni en el código escrito por el usuario de la misma (si bien el mismo necesitaría ser recompilado). Esta útil técnica de diseño es ampliamente usada en el diseño de la Librería Estándar de C++, no sólo para el caso de los iteradores sino en el diseño general de la misma.

Constructores

La clase posee un único constructor con un único parámetro correspondiente a la etiqueta o nombre de acción:

```
class Action{
// ...
public:
        // constructor(es)
        Action(const std::string& l);
// ...
};
```

La semántica del destructor por defecto de la clase es apropiada.

Operaciones

El conjunto de pasos en los que la acción aparece instanciada en el bloque de acciones es mantenido a través de dos pares de funciones miembro:

```
class Action{
// ...
public:
        // operaciones
        void bind(const Step*);
        void bind(step_iterator);

        void unbind(const Step*);
        void unbind(step_iterator);
// ...
};
```

La clase *Step*, usada para representar el concepto de paso, se describe más adelante. El parámetro actual para las funciones que reciben un *step_iterator* debe ser un iterador válido dentro de la secuencia definida por *steps_begin()* y *step_end()*; de lo contrario el comportamiento de las mismas es indefinido. Para implementar las operaciones anteriores de la manera más eficiente posible (*O(log n)*), los pasos en el conjunto deben mantenerse ordenados. La representación exacta usada en la implementación del prototipo fue mediante un objeto de la clase *std::set<const Step*>* de la librería estándar de C++, y el tipo exacto para *step_iterator*, *std::set<const Step*>::const_iterator*. Como se verá más adelante, los objetos

contenidos por una instancia de la clase SFC – en particular los pasos – son identificados de manera inequívoca a través de su dirección física en memoria. Por tanto, la comparación por defecto utilizada por *std::set<const Step*>* es suficiente.

Finalmente, se incluyó en la interfaz un par de funciones de acceso triviales:

```
class Action{
// ...
public:
        // funciones de acceso
        std::string label() const;
        void set_label(const std::string&);
// ...
};
```

2.3.1.2. Clase *STAction*

La clase *Action* sirve como base para la definición de la clase *STAction*, la cual representa el concepto de *acción ST*. No es propósito del editor realizar chequeo sobre el código de una *acción ST*; simplemente es visto como una cadena de caracteres. Excepto por la derivación de la clase *Action*, no hay nada de interesante en la definición de la clase *STAction*:

```
class STAction:  public Action{
// ...
public:
        // constructor
        STAction(const std::string& label, const std::string& code );

        // funciones de acceso necesarias (triviales)...
};
```

2.3.1.3. Clase *SFCAction*

La clase *Action* sirve como base para la definición de la clase *SFCAction*, la cual representa el concepto de *acción SFC*. En ocasiones es conveniente pensar en una *acción SFC* como un SFC mismo; por consiguiente, la interfaz de la clase *SFCAction* incluye un operador de conversión, de manera que una *acción SFC* puede ser utilizada indistintamente como un *SFC*:

```
class SFCAction:  public Action{
// ...
public:
        // constructor
        STAction(const std::string& label, SFC* sfc );

        // operador de conversión
        operator SFC*();

        // funciones de acceso necesarias(triviales)...
};
```

La clase *SFC*, definida para representar el concepto de diagrama SFC, se describe más adelante.

2.3.1.4. Consideraciones

La derivación de las clases *STAction* y *SFCAction* de la clase *Action* permitió a las dos primeras obtener ventaja del servicio común brindado por *Action* referente al mantenimiento del conjunto de estados en los que la acción aparece instanciada en el bloque de acciones. De no haber existido una clase común, hubiera sido necesario replicar tal servicio en ambas clases. Para clases así pequeñas, tal cuestión no es muy relevante; sin embargo, para clases más complejas, la explotación o uso de servicios comunes es crucial.

Por otro lado, la única ventaja adicional de tal estructura jerárquica proporciona conveniencia desde el punto de vista de la notación en la definición simple de un tipo *ActionInstance* para las instancias de acciones:

```
// Modificadores de acción
enum Modifier{N, P, P0, P1, S, R, L, D, SD, DS, SL};

struct ActionInstance{
        Modifier modifier;
        Action* action;

        ActionInstance(Modifier, Action*);
};
```

No existen operaciones en la clase *Action* que potencialmente necesiten ser redefinidas por clases derivadas de ella; más aún, no hay posibilidad alguna de que surja una nueva extensión al concepto de acción – esto es, un nuevo tipo de acción que necesite ser derivado de la clase *Action* – al menos en el dominio de la teoría de SFC. En tal sentido, los tipos STAction y SFCAction son *tipos concretos*, y la derivación de la clase Action es usada solamente para obtener las facilidades anteriormente mencionadas. La invariante de estas clases no involucra apsectos de gran interés.

2.3.2. Transiciones

Desde el punto de vista de la semántica de un SFC, existen solamente dos tipos de transiciones de interés: *transiciones simples* y *transiciones simultáneas*. Sin embargo, como fue descrito en el Capítulo I, en la representación gráfica de un SFC también se manejan los conceptos de *transición disyuntiva divergente*, *transición disyuntiva convergente*, *transición simultánea divergente*, *transición simultánea convergente* y *transición Rendezvous*. El problema fundamental consiste en decidir si es conveniente o no reflejar directamente tal distinción en el diseño, mediante la definición de clases por separado.

El diseño general del prototipo iniciado hace que la operación de representación del diagrama SFC en pantalla (*rendering*; [DavNei], [San96]) sea crítica en cuanto a tiempo de ejecución. Es muy ineficiente esperar hasta el momento de la representación para conformar aquellos conjuntos de transiciones simples que corresponden a transiciones disyuntivas divergentes y convergentes, respectivamente. Por tal motivo se decidió que las transiciones disyuntivas, tanto convergentes como divergentes, necesitan ser representadas explícitamente mediante clases en el diseño.

De forma contraria, aunque se requiere de representación diferente para *transiciones simultáneas convergentes*, *simultáneas divergentes* y *Rendezvous*, es fácil y eficiente establecer tal distinción en una transición simultánea mediante la simple inspección del cardinal del conjunto de pasos anteriores y el conjunto de pasos posteriores de una *transición simultánea*. Por consiguiente, se consideró innecesaria la representación por separado de de tres tipos de *transiciones simultáneas*.

2.3.2.1. Clase *SingleTransition*

Constructor

La clase *SingleTransition* emula el concepto de transición simple; su interfaz es bastante pequeña y posee un solo constructor cuya semántica es obvia:

```
class SingleTransition{
//...
public:
        // constructor
        SingleTransition(Step*, const Condition&, Step*);
// ...
};
```

Una condición es representada simplemente como una cadena de caracteres:

```
typedef std::string Condition;
```

Funciones de acceso

En un diagrama, el paso anterior de una transición simple puede verse como el paso desde el cual la transición “sale” (*outgoing step*), mientras el paso posterior como el paso de “entrada” en la transición (incoming). Un par de funciones brindan acceso a los pasos de la transición, y una tercera a la condición:

```
class SingleTransition{
//...
public:
```

```
        // funciones de acceso
        Step* outgoing();
        Step* incoming();

        Condition& condition();
//...
};
```

Nótese que es imposible modificar cuáles son los pasos que conforman la transición. Los pasos en un diagrama son identificados de manera inequívoca por su dirección física en memoria.

Otras operaciones

Con anterioridad se hizo mención al hecho de que, como parte de la semántica de la eliminación de un paso en el diagrama, es necesario realizar ciertas operaciones sobre las transiciones en las que el paso está involucrado, con el fin de garantizar la validez del diagrama resultante. Recorrer exhaustivamente con tal fin todas las transiciones del diagrama es muy ineficiente. Una mejor opción es mantener, para cada paso, un conjunto de transiciones en las que el paso está involucrado. De manera semejante a como se diseñó para las acciones, tal conjunto de transiciones es mantenida en la clase *Step* (presentada más adelante) a través de un conjunto de operaciones en su interfaz. La cuestión es que, luego de insertar cualquier transición en el diagrama SFC, es necesario, para cada paso *p* de la transición, añadir la misma al conjunto de transiciones en las que *p* está involucrado. La clase *SFC* (responsable de tal actualización) pudiera acceder a los pasos de la transición con tal fin a través de las funciones de acceso anteriores, o bien la clase SingleTransition pudiera ofrecer tal actualización como servicio. Tal libertad en el diseño es posible debido a que las clases son desarrolladas paralelamente:

```
class SingleTransition{
//...
public:
        // operaciones
        void bind_steps();
};
```

La invariante de esta clase es igualmente simple y no ofrece gran interés.

2.3.2.2. Clase *DivTransition*

La clase *DivTransition* representa la noción de *transición disyuntiva divergente*. En una *transición disyuntiva divergente*, cada paso posterior está asociado a una condición. Se definió como *rama* al par *(c, p)* conformado por un paso posterior de una transición disyuntiva divergente, y su condición asociada. Para hacer explícita esta asociación, se define el tipo *Branch*:

```
struct Branch{
        Step*const  step;
        Condition cond;

        Branch(Step*, const Condition&);
};
```

Iteradores

La clase *DivTransition* provee tipos y funciones que permiten iterar conveniente y eficientemente sobre el conjunto de ramas de la transición:

```
class DivTransition{
//...
public:
        // tipos miembros
        typedef tipo_dependiente_de la implementación incoming_iterator;

        // iteradores
        incoming_iterator incoming_begin() const;
        incoming_iterator incoming_end() const;
//...
};
```

Como se señaló en el caso de las acciones, esta útil técnica de diseño, ampliamente utilizada en la librería estándar de C++, permite modificar la implementación de la clase – particularmente los tipos involucrados – sin necesidad de cambios en la interfaz, y por consiguiente, en el código del usuario de la clase. Aún más, permite iterar sobre una secuencia de elementos sin que el usuario de la clase tenga que conocer el tipo exacto de los iteradores utilizados [Str97] . Los iteradores retornados por las funciones *incoming_begin()* y *incoming_end()* delimitan la secuencia de ramas de la transición.

Constructores

Se definió para la clase un constructor único, genérico:

```
class DivTransition{
//...
public:
        // constructor
        template <class In>
        DivTransition(Step*, In first, In last);
//...
};
```

Esto permite construir una *transición disyuntiva divergente* en la que sus ramas sean especificados a través de la secuencia *[first, last[*. El primer parámetro corresponde al paso anterior de la transición. El tipo *In* debe cumplir los requerimientos de un *"input iterator"*, como se especifica en la documentación de la librería estándar de C++ [Str97] . Esta técnica avanzada de diseño también es muy útil, y es muy utilizada en el diseño de los contenedores estándar (*standard containers*) de la Librería Estándar de C++.

Operaciones de inserción y eliminación

También fueron incluídas operaciones de inserción y eliminación de *ramas* en la transición:

```
class DivTransition{
//...
public:
        // operaciones de inserción de ramas
        void insert_incoming(Step*);
        void insert_incoming(const Branch&);

        template <class In>
        void insert_incoming(In first, In last);

        // operaciones de eliminación de ramas
        void erase_incoming(Step*);
        void erase_incoming(const Branch&);
        void erase_incoming(incoming_iterator);

        template <class In>
```

```
        void erase_incoming(In first, In last);
//...
};
```

Si al intentar insertar una *rama* se especifica solamente el paso, se toma la condición por defecto. Es posible eliminar una *rama* en la transición especificando el paso posterior correspondiente, la rama, o un iterador válido dentro de la secuencia de ramas definida por *[incoming_begin(), incoming_end()[*. También se pueden insertar o eliminar secuencias enteras delineadas por pares de iteradores. Al igual que en el caso del constructor, en ambos casos el tipo *In* debe cumplir los requerimentos de un *"input iterator"*.

Existen tres razones de peso para mantener ordenadas las ramas dentro de una *transición disyuntiva divergente*. La primera tiene que ver con la eficiencia de las operaciones anteriores. Manteniendo las ramas ordenadas es posible la inserción y eliminación de una rama en *O(log n)*, versus *O(n)* sin ordenar (esto respecto al número *n* de ramas en la transición). La segunda razón tiene que ver con la eficiencia de la comparación entre dos transiciones disyuntivas divergentes, cuya importancia se verá más adelante. Si las ramas están ordenadas, el caso peor se resuelve en *O(n)*; de otra forma la complejidad en el caso peor es de $O(n^2)$. El tipo exacto para representar el conjunto de ramas en una *transición disyuntiva divergente* en la implementación del prototipo fue *std::set<Branch>*, de la Librería Estándar de C++ [Str97] . Para ello fue necesario definir el *operador "<"* para el tipo Branch con una semántica conveniente:

```
bool operator <(const Branch&, const Branch&);
```

El mismo fue implementado mediante la comparación de las direcciones físicas de los pasos correspondientes a ambas ramas.

Una tercera y última razón tiene que ver con un tema no antes mencionado, referente al determinismo de la ejecución de un programa SFC. En síntesis, para garantizar el determinismo en la ejecución de un SFC, las transiciones simples con un mismo paso anterior deben ser ordenadas respecto a algún criterio bien definido. En una versión más desarrollada del prototipo, debe darse posibilidad al usuario de la clase de suplir como parámetro el orden para las ramas en una *transición disyuntiva convergente*. Por ejemplo:

```
template <class Cmp>
class DivTransition{
//...
```

```
private:
// ramas
std::set<Branch, Cmp> branches;
public:
        // constructor
        template <class In>
        DivTransition(Step*, In first, In last, Cmp);
};
```

Esta elegante técnica también es utilizada en el diseño de los contenedores de la Librería Estándar de C++ (s*tandard containers*).

Otras operaciones

En ocasiones se necesita conocer la cantidad de pasos posteriores de la transición. Desde el punto de vista del usuario de la clase, es muy ineficiente recorrer la secuencia de ramas de entrada con tal propósito. Un tipo y una función miembro brindan acceso al número de pasos posteriores o ramas en la transición:

```
class DivTransition{
//...
public:
        // tipos miembros
        //...
        typedef tipo_dependiente_de la implementación count_type;

        // operaciones
        count_type  incoming_count() const;
//...
};
```

La definición de un nombre de tipo en la interfaz de la clase para la cantidad de pasos posteriores es necesaria. El tipo exacto para la cantidad de ramas utilizado en la implementación del prototipo fue lógicamente *std::set<Branch>::size_type*, en correspondencia con la representación escogida para la clase.

Es posible también obtener un iterador a la rama correspondiente a un paso determinado. Esto permite, por ejemplo, modificar la condición asociada al paso:

```
class DivTransition{
//...
public:
        // operaciones
        incoming_iterator  find(Step*) const;
//...
};
```

Finalmente, se incluyó una función de acceso al paso anterior en la transición. La función *bind_steps()* sirve en la clase *DivTransition* las mismas necesidades discutidas en la clase *SingleTransition*:

```
class DivTransition{
//...
public:
        // operaciones
        Step*   outgoing() const;

        void bind_steps();
};
```

La invariante de la clase *DivTransition* es sencilla y fácil de mantener. Lo más significativo es que el conjunto de pasos posteriores debe contener al menos dos elementos.

2.3.2.3. Clase *ConvTransition*

La clase *ConvTransition* dentro del diseño representa la noción de *transición disyuntiva convergente*. El diseño, la interfaz y la semántica de esta clase es muy semejante a la de la clase *DivTransition*. Luego de la discusión hecha para la clase *DivTransition*, la clase *ConvTransition* es presentada solamente a través de su interfaz:

```
class ConvTransition{
//...
public:
        // tipos miembros
        typedef tipo_dependiente_de la implementación outgoing_iterator;
        typedef tipo_dependiente_de la implementación count_type;
```

```
        // iteradores
        outgoing_iterator outgoing_begin() const;
        outgoing_iterator outgoing _end() const;

        // constructor
        template <class In>
        ConvTransition(In first, In last, Step*);

        // operaciones de inserción de ramas
        void insert_ outgoing (Step*);
        void insert_ outgoing (const Branch&);

        template <class In>
        void insert_ outgoing (In first, In last);

        // operaciones de eliminación de ramas
        void erase_ outgoing (Step*);
        void erase_ outgoing (const Branch&);
        void erase_ outgoing (outgoing _iterator);

        template <class In>
        void erase_ outgoing (In first, In last);

        // otras operaciones
        Step*   incoming() const;
        count_type  outgoing _count() const;
        outgoing_iterator  find(Step*) const;

        void bind_steps();
};
```

Cada consideración hecha para la clase *DivTransition* tienen su contraparte válida también para la clase *ConvTransition*.

2.3.2.4. Clase *SimultTransition*

El diseño e implementación de la clase *SimultTransition*, definida para la representación del concepto de *transición simultánea*, sigue el estilo utilizado en el diseño de *DivTransition* y *ConvTransition*.

Iteradores

Una *transición simultánea* requiere la iteración sobre el conjunto de pasos anteriores y sobre el conjunto de pasos posteriores. Como es usual, se ofrecen tipos y funciones para la iteración:

```
class SimultTransition{
//...
public:
        // tipos miembros
        typedef tipo_dependiente_de la implementación outgoing_iterator;
        typedef tipo_dependiente_de la implementación incoming_iterator;

        // iteradores
        outgoing_iterator outgoing_begin() const;
        outgoing_iterator outgoing_end() const;

        incoming_iterator incoming_begin() const;
        incoming_iterator incoming_end() const;
//...
};
```

Constructores

Para construir una transición simultánea en necesario especificar el conjunto de pasos anteriores, el conjunto de pasos posteriores y la condición. Ambos conjuntos son especificados en el constructor de la clase mediante pares de iteradotes que delimitan secuencias de pasos:

```
class SimultTransition{
//...
public:
        // constructor
        template <class In, class In2>
        SimultTransition(In first, In last, const Condition&, In2 first2, In2 last2);
//...
};
```

Nótese que los tipos de iteradores utilizados para especificar ambas secuencia de pasos no tienen por qué ser necesariamente iguales. Esto hace al constructor de la clase aún más flexible y versátil. El

conjunto de pasos anteriores es especificado mediante la secuencia *[first, last[*, mientras que el conjunto de pasos posteriores mediante la secuencia *[first2, last2[*.

Operaciones de inserción y eliminación

Las siguientes operaciones permiten la inserción y eliminación de pasos anteriores y pasos posteriores en una transición simultánea:

```
class SimultTransition{
//...
public:
        // operaciones de inserción pasos
        void insert_outgoing(Step*);
        void insert_incoming(Step*);

        template <class In>
        void insert_outgoing(In first, In last);

        template <class In>
        void insert_incoming(In first, In last);

        // operaciones de eliminación de pasos
        void erase_outgoing(Step*);
        void erase_outgoing(outgoing_iterator);

        void erase_incoming(Step*);
        void erase_incoming(incoming_iterator);

        template <class In>
        void erase_outgoing(In first, In last);

        template <class In>
        void erase_incoming(In first, In last);

        void erase (Step*);
        void erase (incoming_iterator);
//...
};
```

Las funciones *erase()* eliminan un paso en la transición, ya sea que éste pertenezca al conjunto de pasos anteriores o al conjunto de pasos posteriores.

Los pasos anteriores, así como los posteriores, deben ser mantenidos en orden por la implementación; de esta forma las operaciones de eliminación e inserción simple de pasos en la transición tiene complejidad *O(log n)* en el caso peor. El tipo exacto utilizado en la implementación del prototipo para representar ambos conjuntos de pasos en la clase fue *std::set<Step*>*, de la Librería Estándar de C++.

Otras operaciones

En la interfaz se incluyen tipos y funciones miembros para acceder a la cantidad de pasos anteriores y pasos posteriores, respectivamente:

```
class SimultTransition{
//...
public:
        // tipos miembros
        //...
        typedef tipo_dependiente_de la implementación count_type;

        // operaciones
        count_type  outgoing_count() const;
        count_type  incoming_count() const;
//...
};
```

Por último, se incluyó la acostumbrada operación *bind_steps()* para transiciones, cuya utilidad se expuso en la presentación de la clase *SingleTransition*:

```
class SimultTransition{
//...
public:
        // operaciones
        void bind_steps();
};
```

La invariante de esta clase es relativamente simple, y fácil de mantener. Los conjuntos de pasos anteriores y posteriores deben contener ambos al menos un elemento, y el cardinal de al menos uno de ellos debe ser mayor que uno.

2.3.2.5. Alternativas de diseño

Pudiera resultar tentadora la derivación de las clases *SingleTransition*, *DivTransition*, *ConvTransition* y *SimultTransition* de una clase común *Transition*. En definitiva, todas representan *transiciones* en un diagrama SFC. Sin embargo, exceptuando el caso de la operación *bind_steps()*, sorprendentemente no existen operaciones comunes a las cuatro clases. La operación *bind_steps()* es un artefacto de implementación en el diseño general del conjunto de clases; aun más, ésta y otras operaciones deberían ser removidas de la interfaz (si bien no de las clases), como se explicará más adelante.

Por otro lado, el uso de una jerarquía tampoco sería útil, digamos, para mantener el conjunto de transiciones del diagrama en una misma estructura de datos. Los diferentes tipos de transiciones en un diagrama han de mantenerse separadas para garantizar la eficiencia de algunas operaciones sobre el mismo. Si en el futuro se decide reemplazar una clase por un nuevo tipo de transición en el diseño, o modificar el conjunto de tipos de transiciones consideradas, ello implicaría inevitablemente cambios en el diseño general y en la implementación del conjunto de clases. En tal sentido, los cuatro tipos definidos para representar las transiciones de un diagrama son *tipos concretos*.

Por otro lado, si el usuario del conjunto de clases tuviera la necesidad de insertar las clases para representar transiciones en una misma jerarquía, ello podría lograrse mediante el uso de la herencia y clases de envoltura (*wrapper classes* [Bec04]).

En conclusión, no hay ventaja alguna que se pueda obtener de tal derivación. El concepto de *transición* es una abstracción en el dominio de SFC, pero *no* constituye una abstracción en el dominio de la aplicación.

2.3.3. Pasos

Dentro del conjunto de clases, la clase *Step* fue concebida para representar el concepto de *paso* en un diagrama SFC. La interfaz de la misma es presentada a continuación por partes.

Iteradores

Como se comentó con anterioridad, la semántica de la operación de eliminación de un *paso* en un diagrama SFC incluye la realización de algunas operaciones con las transiciones en las que el paso está involucrado. El tipo de operación a realizar con la transición depende del tipo de transición (simple, disyuntiva divergente, disyuntiva convergente o simultánea). Así, pues, para la implementación de la operación de eliminación de un paso en el diagrama de la manera más eficiente posible, es necesario mantener, para cada paso, el conjunto de transiciones en las que el paso toma parte. Los siguientes tipos e iteradores definidos en la interfaz permiten iterar sobre tales transiciones:

```
class Step{
//...
public:
        // tipos miembros
        typedef tipo_dependiente_de la implementación single_iterator;
        typedef tipo_dependiente_de la implementación div_iterator;
        typedef tipo_dependiente_de la implementación conv_iterator;
        typedef tipo_dependiente_de la implementación simult_iterator;

        // iteradores
        single_iterator single_begin() const;
        single_iterator single_end() const;

        div_iterator div_begin() const;
        div_iterator div_end() const;

        conv_iterator conv_begin() const;
        conv_iterator conv_end() const;

        simult_iterator simult _begin() const;
        simult _iterator simult _end() const;
//...
};
```

Las secuencias *[single_begin(), single_end()[*, *[div_begin(), div_end()[*, *[conv_begin(), conv_end()[*, y *[simult_begin(), simult_end()[*, brindan acceso al conjunto de transiciones simples, transiciones disyuntivas divergentes, transiciones disyuntivas convergentes y transiciones simultáneas, respectivamente, en las que el paso está involucrado. Esta técnica de diseño ya fue discutida durante la presentación de las clases anteriores.

También fueron incluidos tipos y funciones que permiten iterar sobre las instancias de acciones del bloque de acción asociado al paso:

```
class Step{
//...
public:
        // tipos miembros
        //...
        typedef tipo_dependiente_de la implementación block_iterator;

        // iteradores
        //...
        block _iterator block_begin() const;
        block_iterator block_end() const;
//...
};
```

Constructores

Para la clase *Step* se definió un único constructor:

```
class Step{
//...
public:
        // constructor
        explicit Step(const std::string&);
//...
};
```

Solamente es necesario especificar la etiqueta correspondiente al *paso*.

Operaciones

Un conjunto de funciones permite agregar o eliminar elementos dentro del conjunto de transiciones en las que el paso está involucrado. Tales operaciones son necesarias porque tal conjunto de transiciones es mantenido *en* pero no *por* la clase *Step*:

```
class Step{
//...
public:
        // operaciones
        void bind_to(SingleTransition*);
        void bind_to(DivTransition*);
        void bind_to(ConvTransition*);
        void bind_to(SimultTransition*);

        void unbind(SingleTransition*);
        void unbind(DivTransition*);
        void unbind(ConvTransition*);
        void unbind(SimultTransition*);
//...
};
```

Para garantizar la implementación de estas funciones de la manera más eficiente posible, las transiciones deben mantenerse ordenadas dentro de cada uno de los cuatro conjuntos, de manera que la complejidad en el caso peor tanto en la inserción como en la eliminación sea *O(log n)*. Los tipos exactos utilizados en la implementación del prototipo para representar el conjunto de transiciones simples, transiciones disyuntivas divergentes, transiciones disyuntivas convergentes y transiciones simultáneas asociadas a un paso fueron *std::set<SingleTransition*>*, *std::set<DivTransition*>*, *std::set<ConvTransition*>*, *std::set<SingleTransition*>*, respectivamente. La utilización de punteros a transiciones existentes en el diagrama permite que la comparación entre transiciones – mediante la comparación de direcciones físicas en memoria – tenga lugar en tiempo constante (*O(1)*).

En cierto punto de la implementación de las clases también es preciso conocer la cantidad exacta de elementos en cada uno de estos conjuntos de transiciones. Un tipo miembro y un conjunto de funciones sirven con tal propósito:

```
class Step{
//...
```

```
public:
        // tipos miembros
        //...
        typedef tipo_dependiente_de la implementación count_type;

        // iteradores
        //...
        count_type single_count() const;
        count_type div_count() const;
        count_type conv_count() const;
        count_type simult_count() const;
//...
};
```

Se incluyeron además funciones para modificar el *bloque de acciones* asociado al *paso*:

```
class Step{
//...
public:
        // operaciones sobre el bloque de acciones
        void insert_instance(const ActionInstance&);
        void insert_instance(block_iterator, const ActionInstance&);

        void erase_instance(block_iterator);
        void erase_instance(Action*);

        void clear_block();
//...
};
```

Las instancias de acción son insertadas por defecto al final del bloque de acciones. Es posible insertar una nueva instancia de acción una posición del bloque de acciones especificada a través de un iterador. El iterador especificado debe ser un iterador válido dentro de la secuencia *[block_begin(), block_end()[*.

Por otra parte, es posible eliminar una instancia en el bloque especificando su posición a través de un iterador. Al igual que en la inserción, el iterador especificado debe ser un iterador válido dentro de la secuencia *[block_begin(), block_end()[*. La función *erase_instance(Action*)* permite eliminar todas las instancias de una acción en el bloque; por su parte, *clear _block()* permite eliminar todas las instancias del bloque.

Finalmente, fueron incluídas un par de funciones para acceder a la etiqueta del paso:

```
class Step{
//...
public:
        // operaciones de acceso
        const std::string label() const;
        void set_label(const std::string&);
};
```

2.3.4. SFCs

Dentro del diseño de clases elaborado, la clase *SFC*, para la representación del concepto de diagrama SFC, tiene un papel protagónico y atrae el mayor interés. El resto de las clases en el diseño fueron concebidas simplemente con el propósito de implementar y ofrecer una interfaz conveniente para los usuarios de la clase *SFC*. No es de asombro que la misma ofrezca los principales retos en cuanto al diseño y el modo de uso, la implementación, así como el establecer y mantener una invariante apropiada para la clase. Estos y otros aspectos son discutidos en lo adelante.

2.3.4.1. Interfaz

En cierta forma, la clase *SFC* asemeja mucho a un “contenedor de objetos” (*container* [Str97]). No exactamente un contenedor homogéneo, cuyos objetos son todos de un mismo tipo; más bien una especie peculiar de contenedor que almacena tres tipos de elementos fundamentales – pasos, acciones y transiciones – y que en adición brinda soporte para la interrelación que se establece entre ellos. Esta noción constituye, sin lugar a dudas, el punto clave en el diseño y la concepción de la clase *SFC*.

En consecuencia de lo anterior, muchas de las operaciones sobre un SFC asemejan a aquellas brindadas por las “clases contenedoras” (standard containers) de la Librería Estándar de C++; tal semejanza es reflejada en el diseño de parte de la interfaz de la clase *SFC*.

Iteradores

Un *diagrama SFC* está compuesto por tres elementos fundamentales: pasos, acciones y transiciones. Un conjunto apropiado de tipos y funciones en la interfaz de la clase permiten iterar sobre los elementos fundamentales de un SFC:

```
class SFC{
//...
public:
        // tipos miembros
        typedef tipo_dependiente_de la implementación step_iterator;

        typedef tipo_dependiente_de la implementación st_action_iterator;
        typedef tipo_dependiente_de la implementación sfc_action_iterator;

        typedef tipo_dependiente_de la implementación single_iterator;
        typedef tipo_dependiente_de la implementación div_iterator;
        typedef tipo_dependiente_de la implementación conv_iterator;
        typedef tipo_dependiente_de la implementación simult_iterator;

        // iteradores
        step_iterator step_begin() const;
        step_iterator step_end() const;

        st_action _iterator st_action_begin() const;
        st_action_iterator st_action_end() const;

        sfc_action_iterator sfc_action_begin() const;
        sfc_action_iterator sfc_action_end() const;

        single_iterator single_begin() const;
        single_iterator single_end() const;

        div_iterator div_begin() const;
        div_iterator div_end() const;

        conv_iterator conv_begin() const;
        conv_iterator conv_end() const;

        simult_iterator simult_begin() const;
        simult_iterator simult_end() const;
};
```

Es posible iterar sobre el conjunto de *pasos* del diagrama a través de la secuencia definida por los iteradores *step_begin()* y *step_end()*. La iteración sobre el conjunto de *acciones* de un *diagrama SFC* se realiza por partes; la secuencia *[st_action_begin(), st_action_end()[* y *[sfc_action_begin(), sfc_action_end()[* permiten la navegación a través del conjunto de *acciones ST* y *acciones SFC*, respectivamente. De forma similar, la iteración a través del conjunto de *transiciones* se logra a través de la iteración por separado sobre las secuencias *[single_begin(), single_end()[*, *[div_begin(), div_end()[*, *[conv_begin(), conv_end()[* y *[simult_begin(), simult_end()[*. No se consideró que hubiese ventaja alguna en ofrecer al usuario de la clase un iterador "homogéneo" sobre el conjunto de *acciones*. Lo mismo ocurrió para el caso de las *transiciones*; una razón más por la cual que fue desechado el uso de una jerarquía de clases para las mismas.

La iteración sobre los elementos de un diagrama SFC es potencialmente la operación más crítica en cuanto al tiempo de ejecución, desde el punto de vista del usuario de la clase. En una aplicación donde el diagrama SFC y sus elementos necesiten ser representados en la interfaz gráfica de usuario repetitivamente en fracciones de segundo (por ejemplo, cada vez que el cursor del mouse cambia de posición), es crucial que la iteración sobre los elementos del diagrama sea implementada lo más eficientemente posible. La plataforma conformada por lo contenedores (*containers*) e iteradores (*iterators*) de la Librería Estándar de C++ (conocida como *STL framework* [Str97]) ofrece excelente soporte para este fin.

No es de sorprender entonces que los contenedores estándar de C++ hayan sido utilizados internamente en la clase SFC para representar los respectivos conjuntos de pasos, acciones y transiciones en el diagrama; que los nombres de tipo definidos para los iteradores en la interfaz de la clase correspondan a los nombres de tipos definidos en la interfaz de los contenedores utilizados; y que los iteradores retornados por las catorce funciones sean aquellos que provee la Librería Estándar de C++ sobre los contenedores estándar. Más adelante se discutirá en mayor detalle acerca de algunos aspectos claves de la representación e implementación de la clase *SFC*.

Operaciones de inserción y eliminación

Un conjunto clave dentro de las operaciones sobre un *diagrama SFC* son aquellas correspondientes a la inserción y eliminación de sus elementos fundamentales. El siguiente conjunto de funciones en la interfaz de la clase *SFC* brinda soporte para tales operaciones:

```
class SFC{
//...
public:
        // operaciones de inserción
        void insert(Step*);

        void insert(STAction*);
        void insert(SFCAction*);

        void insert(SingleTransition*);
        void insert(DivTransition*);
        void insert(ConvTransition*);
        void insert(SimultTransition*);

        // operaciones de eliminación
        void erase(Step*);
        void erase(step_iterator);

        void erase(Action*);
        void erase(st_action_iterator);
        void erase(sfc_action_iterator);

        void erase(SingleTransition*);
        void erase(DivTransition*);
        void erase(ConvTransition*);
        void erase(SimultTransition*);

        void erase(single_iterator);
        void erase(div_iterator);
        void erase(conv_iterator);
        void erase(simult_iterator);
//...
};
```

Los *pasos*, *acciones* y *transiciones* de un diagrama han de ser almacenados de algún modo dentro de la clase *SFC*; muy probablemente utilizando algún tipo de contenedor. Los contenedores estándar de C++ ofrecen excelente soporte, y fueron los utilizados para la implementación del prototipo.

Colocar directamente los objetos en los contenedores no es una buena opción. Un primer inconveniente consiste en que, tanto *pasos*, *acciones* como *transiciones* necesitan ser "referenciados" desde otros objetos en el diagrama. Por ejemplo, una acción necesita "conocer" el conjunto de pasos en los cuales aparece instanciada dentro del bloque de acciones – por razones de eficiencia, como se explicó con anterioridad. Por motivos similares, es bueno mantener para cada paso el conjunto de transiciones en los que el paso está involucrado. La cuestión clave es que, una forma conveniente y eficiente de referenciar los objetos es mediante la dirección física de los mismos. Si los objetos son colocados directamente en los contenedores, sería imposible referenciarlos mediante su dirección física. Ello se debe a que algunas operaciones sobre los contenedores estándar (o algún otro contenedor utilizado) pudieran ocasionalmente incurrir en la reasignación de memoria (*memory reallocation* [Str97]) para algunos de los elementos del contenedor. De ser así, sería necesario ir en búsqueda de alternativas para establecer tal asociación entre objetos dentro del diagrama, que conducirían a la replicación de objetos con la correspondiente ineficiencia en cuanto a espacio, y el uso de costosas operaciones para examinar la igualdad entre objetos.

Es muy conveniente que los elementos en el diagrama sean representados de forma única, y que sean identificados (referenciados) inequívocamente por su dirección física en memoria. De hecho, el diseño y la implementación de las clases anteriormente presentadas – Step, Action, SingleTransition, DivTransition, convTransition y SimultTransition – confían plenamente en que la clase SFC satisface esta propiedad. Son éstos aspectos fundamentales desde el punto de vista del implementador – y no del usuario – de la clase SFC y del conjunto de clases en general.

La solución para este problema consistió en colocar punteros a objetos en lugar de objetos dentro de los contenedores utilizados para representar los *pasos*, *acciones* y *transiciones* en la clase *SFC*. De esta forma se lidia de manera conveniente con los puntos señalados en el párrafo anterior. Esta crucial decisión desde el punto de vista de implementación tuvo, sin embargo, su repercusión en la interfaz y el modo de uso (*intended use* [Str97]) de la clase *SFC* y del conjunto de clases.

Una de las implicaciones consiste en que los iteradores ofrecidos por la interfaz de la case *SFC* iteran sobre punteros a elementos del diagrama, y no directamente sobre los punteros. Digamos, la secuencia *[steps_begin(), steps_end()[* permite iterar sobre objetos que son de tipo Step* – punteros a los pasos del diagrama. Por ejemplo:

```
void f(const SFC& sfc)
{
        for (SFC::step_iterator p = step_begin(); p != step_end(); ++p)
        {
                render((*p)->label());          //ok
                render((*p).label());           //error!
        }
}
```

Es posible definir iteradores que permitan la iteración directamente sobre los objetos, sin embargo se decidió evadir la carga de la implementación de clases que requieren ser cuidadosamente elaboradas. En definitiva, esto no impone un gravamen significativo sobre el usuario de la clase; errores en el uso como el anterior son capturados por el compilador.

Un segundo aspecto tiene que ver con el tipo de los parámetros utilizados en las operaciones de inserción de nuevos elementos en el diagrama. Se consideró la variante de utilizar referencias a objetos:

```
class SFC{
//...
public:
        // operaciones de inserción
        void insert(const Step&);

        void insert(const STAction&);
        void insert(const SFCAction&);

        void insert(const SingleTransition&);
        void insert(const DivTransition&);
        void insert(const ConvTransition&);
        void insert(const SimultTransition&);
//...
};
```

Sin embargo fue desechada, debido a que la posibilidad de que el objeto insertado tenga alcance local y sea eventualmente destruido obliga realizar operaciones de copia durante la inserción, con sus respectivas ineficiencias. Además, el usuario podría verse tentado a realizar cambios en un objeto utilizado en la inserción, sin que los mismos tengan efecto sobre el objeto realmente agregado al diagrama. Tales errores no podrían ser capturados por el compilador.

Para insertar un nuevo elemento en el diagrama, es necesario especificar su dirección física en memoria. Esto no deja de imponer sus propias restricciones acerca del uso de la clase. Los objetos cuya dirección física es especificada son insertados directamente en el conjunto de elementos correspondiente, de modo que es necesario que tales objetos hayan sido construidos en memoria permanente utilizando el operador *new*. Por otra parte, y muy importante, la clase *SFC* posee semántica de posesión (*ownership semantics* [Str97]) sobre los elementos del diagrama; es responsabilidad total de la clase la liberación de la memoria asignada a tales objetos mediante la invocación en el momento oportuno del operador *delete*, aun en el caso en que la operación de inserción sea fallida. Por consiguiente, es un error la invocación del operador *delete* sobre la dirección física de cualquiera de los elementos que conforman el diagrama. Tales errores tampoco pueden ser advertidos por el compilador. En el desarrollo futuro de un sistema para la edición de diagramas SFC pudiera votarse a favor de la utilización de referencias y no punteros a objetos, o pudieran incluirse ambas alternativas en la interfaz siempre que se alerte al usuario de la clase sobre el uso correcto de las mismas.

Por último, representar los conjuntos de elementos que componen el diagrama mediante punteros a objetos, ofrece la posibilidad al usuario de la clase SFC – y del conjunto de clases en general – de extender los tipos definidos a su conveniencia mediante el uso de la herencia, sin dejar de obtener los servicios ofrecidos por el conjunto de clases:

```
class UserStep: public Step{
        //...
};

void f(SFC& sfc)
{
        //...
        UserStep* s1 = new UserStep(/* ... */);
```

```
        sfc.insert(s1);
        //...
}
```

Otras operaciones

Finalmente, fueron incluidas operaciones simples de acceso al paso inicial del SFC:

```
class SFC{
//...
public:
        // otras  operaciones
        Step* start() const;
        void set_start(Step*);
        void set_start(step_iterator);
};
```

2.3.4.2. Invariante

Para toda clase concebida es conveniente establecer una invariante bien definida, verdaderamente útil, conveniente, y posible de mantener por la implementación. En ocasiones esta tarea resulta simple y sin mucho interés. No es éste el caso de la clase SFC. Existen puntos en la invariante definida para la misma que deben ser traídos a discusión.

Para que un objeto o instancia de la clase SFC se encuentre en un estado válido, debe cumplir – entre otros – los siguientes requerimientos:

1. Para cada paso p en el diagrama, el conjunto S_p de transiciones simples del diagrama para las cuales p es el paso anterior contiene a lo sumo un elemento.

   La necesidad de tal restricción está dada por lo siguiente: supóngase que para cierto paso p, el conjunto $S_p = \{t_1 = (p, c_1, p_1), t_2 = (p, c_2, p_2), \ldots , t_n = (p, c_n, p_n)\}$, de transiciones simples en las cuales p es el paso anterior, contiene más de un elemento. Entonces la representación gráfica de tal conjunto de transiciones correspondería a la de una única transición disyuntiva divergente $t = (p, \{(c_1, p_1), (c_2, p_2), \ldots, (c_n, p_n)\})$. Pero las transiciones disyuntivas divergentes son representadas

directamente como tal en la clase SFC; por consiguiente, en lugar de tal conjunto de transiciones simples, el SFC debería contener una transición disyuntiva divergente, a decir $t = (p, \{(c_1, p_1), (c_2, p_2), \ldots, (c_n, p_n)\})$.

2. Para cada paso *p* en el diagrama, el conjunto S'_p de transiciones simples del diagrama para las cuales *p* es el paso posterior contiene a lo sumo un elemento.

   La necesidad de tal restricción está dada por lo siguiente: supóngase que para cierto paso *p*, el conjunto $S'_p = \{t_1 = (p_1, c_1, p), t_2 = (p_2, c_2, p), \ldots, t_n = (p_n, c_n, p)\}$, de transiciones simples en las cuales *p* es el paso posterior, contiene más de un elemento. Entonces la representación gráfica de tal conjunto de transiciones correspondería a la de una única transición disyuntiva convergente $t = (\{(c_1, p_1), (c_2, p_2), \ldots, (c_n, p_n)\}, p)$. Pero las transiciones disyuntivas convergentes son representadas directamente como tal en la clase SFC; por consiguiente, en lugar de tal conjunto de transiciones simples, el SFC debería contener una transición disyuntiva convergente, a decir $t' = (\{(c_1, p_1), (c_2, p_2), \ldots, (c_n, p_n)\}, p)$.

3. Para cada paso *p* en el diagrama, el conjunto D_p de transiciones disyuntivas divergentes para las cuales *p* es el paso anterior contiene a lo sumo un elemento.

   La necesidad de tal restricción está dada por lo siguiente: supóngase que para cierto paso *p*, el conjunto $D_p = \{d_1, d_2, \ldots, d_n\}$, de transiciones disyuntivas divergentes en las cuales *p* es el paso anterior, contiene más de un elemento. Sean $P_1, P_2, \ldots, P_n$ los conjuntos de pasos posteriores en $d_1, d_2, \ldots, d_n$, respectivamente. Tales conjuntos de pasos han de ser disjuntos dos a dos, de otra forma el diagrama contendría desde el punto de vista semántico transiciones simples ambiguas (con el mismo paso anterior y posterior, pero probablemente no con la misma condición). Sea $P = P_1 \cup P_2 \cup \ldots \cup P_n = \{p_1, p_2, \ldots, p_k\}$, y sean $c_1, c_2, \ldots, c_k$ las condiciones asociadas a los pasos $p_1, p_2, \ldots, p_k$ en sus correspondientes transiciones disyuntivas divergentes en el conjunto D_p, respectivamente. Entonces la representación gráfica – y por consiguiente la representación en el diagrama – de tal conjunto de transiciones correspondería a la de una única transición disyuntiva divergente $t = (p, \{(c_1, p_1), (c_2, p_2), \ldots, (c_k, p_k)\})$.

4. Para cada paso *p* en el diagrama, el conjunto C_p de transiciones disyuntivas convergentes del diagrama para las cuales *p* es el paso posterior contiene a lo sumo un elemento.

   La necesidad de tal restricción está dada por lo siguiente: supóngase que para cierto paso *p*, el conjunto $C_p = \{c_1, c_2, \ldots, c_n\}$, de transiciones disyuntivas divergentes en las cuales *p* es el paso anterior, contiene más de un elemento. Sean $P_1, P_2, \ldots, P_n$ los conjuntos de pasos anteriores en $c_1, c_2, \ldots, c_n$, respectivamente. Tales conjuntos de pasos han de ser disjuntos dos a dos, de otra forma el diagrama contendría desde el punto de vista semántico transiciones simples ambiguas (con el mismo paso anterior y posterior, pero probablemente no con la misma condición). Sea $P = P_1 \cup P_2 \cup \ldots \cup P_n = \{p_1, p_2, \ldots, p_k\}$, y sean $c_1, c_2, \ldots, c_k$ las condiciones asociadas a los pasos $p_1, p_2, \ldots, p_k$ en sus correspondientes transiciones disyuntivas convergentes en el conjunto D_p, respectivamente. Entonces la representación gráfica – y por consiguiente la representación en el diagrama – de tal conjunto de transiciones correspondería a la de una única transición disyuntiva convergente $t = (\{(c_1, p_1), (c_2, p_2), \ldots, (c_k, p_k)\}, p)$.

5. Para cada par de transiciones *(s, d)* del diagrama, donde *s* es una transición simple y *d* es una transición disyuntiva divergente, el paso anterior en *s* no coincide con el paso anterior en *d*.

   Supóngase que existen en el diagrama una transición simple $s = (p, c, q)$ y una transición disyuntiva divergente $d = \{p, \{(c_1, p_1), (c_2, p_2), \ldots, (c_n, p_n)\}\}$ con el mismo paso anterior *p*. El paso *q* no debe coincidir con ninguno de los pasos $p_1, p_2, \ldots, p_n$; de lo contrario el diagrama contendría desde el punto de vista semántico transiciones simples ambiguas (con el mismo paso anterior y posterior, pero probablemente no con la misma condición). Pero entonces la representación gráfica – y por consiguiente la representación en el diagrama – de ambas transiciones correspondería a la de una única transición disyuntiva divergente $d' = (p, \{(c_1, p_1), (c_2, p_2), \ldots, (c_n, p_n), (c, q)\})$.

6. Para cada par de transiciones *(s, t)* del diagrama, donde *s* es una transición simple y *t* es una transición disyuntiva convergente, el paso posterior en *s* no coincide con el paso posterior en *t*.

   Supóngase que existen en el diagrama una transición simple $s = (q, c, p)$ y una transición disyuntiva divergente $t = \{\{(c_1, p_1), (c_2, p_2), \ldots, (c_n, p_n)\}, p\}$ con el mismo paso posterior *p*. El paso *q* no debe coincidir con ninguno de los pasos $p_1, p_2, \ldots, p_n$; de lo contrario el diagrama contendría

desde el punto de vista semántico transiciones simples ambiguas (con el mismo paso anterior y posterior, pero probablemente no con la misma condición). Pero entonces la representación gráfica – y por consiguiente la representación en el diagrama – de ambas transiciones correspondería a la de una única transición disyuntiva convergente $t' = (\{(c_1, p_1), (c_2, p_2), \ldots, (c_n, p_n), (c, q)\}, p)$.

No es este el conjunto completo de reglas que debe satisfacer una instancia de la clase *SFC*. Por ejemplo, si el diagrama contiene al menos un paso, el paso inicial debe estar definido.

El constructor de la clase debe asegurar que el objeto quede en un estado válido después de construido; las operaciones sobre el objeto – en particular las operaciones de inserción – deben ser cuidadosamente implementadas, de manera que luego de su invocación sobre una instancia de la clase en estado válido, la misma se permanezca en un estado igualmente válido. Ello condujo a la elaboración de algoritmos formales para ser utilizados en la implementación de tales operaciones con el fin de mantener la invariante de la clase *SFC* y del conjunto de clases en general.

2.3.5. Alternativas de diseño

En los epígrafes anteriores se hizo la discusión de las clases principales que conforman el diseño, lo cual incluyó la presentación de sus respectivas interfaces. Al mismo tiempo fueron analizadas en cada caso diferentes alternativas de diseño, sin embargo existen todavía algunos aspectos que no fueron considerados.

Aunque el diseño de clases está conformado por varias de ellas, se hizo hincapié en que las mismas fueron concebidas y deben ser vistas en su conjunto, debido a la estrecha relación existente entre ellas. Desde el punto de vista de la implementación, las clases pueden ser consideradas como un todo, en el cual las diferentes partes (clases) son definidas para lograr una mejor separación y organización del problema. Sin embargo, tal separación también permite presentar al usuario de la clase *SFC* una interfaz convenientemente tipada.

Por razones de simplicidad, fueron incluidas en la interfaz de las clases algunas funciones que, en cambio, no fueron concebidas para su utilización por el usuario de la clase, sino como artefacto para la

implementación de otras clases dentro del diseño. En otras palabras, existen operaciones entre aquellas que fueron presentadas que constituyen interfaz para el resto de las clases del diseño, pero no para el usuario del conjunto de clases. Por ejemplo, el conjunto de transiciones en las que un paso está involucrado es un detalle de implementación; tal conjunto de transiciones no es de incumbencia para usuario de las clases, y el mantenimiento del mismo es responsabilidad de la case SFC y del conjunto de clases en general.

Esta intención de uso fue directamente reflejada en el diseño de clases mediante la eliminación de tales funciones de la interfaz de las clases, si bien no de las clases mismas. Para lograr el necesario acceso a una de estas operaciones desde el resto de las clases, las mismas fueron declaradas como clases amigas (*friend classes*, [Str97]). Ello permite obtener soporte del compilador para prevenir el uso accidental o indebido de tales funciones:

```
class Action{
// ...
private:
        // operaciones
        void bind(const Step*);
        void bind(step_iterator);

        void unbind(const Step*);
        void unbind(step_iterator);

        // clases amigas
        friend class SFC;
        friend class Step;

public:
        // tipos miembros
        typedef tipo_dependiente_de_la_implementación  step_iterator;

        // iteradores
        step_iterator steps_begin() const;
        step_iterator steps_end() const;

        // funciones de acceso
        std::string label() const;
```

```
        void set_label(const std::string&);
};
```

Nótese que aunque el usuario de la clase pueda modificar directamente la etiqueta de una acción, la implementación podría (y debería) establecer algún mecanismo de chequeo para asegurar, por ejemplo, que si la acción forma parte de un *SFC*, la etiqueta no aparezca repetida en el mismo para otra acción. Otra alternativa sería remover esta operación de la clase *Action*, e incluir una con un fin similar en la interfaz de la clase *SFC*. Sin embargo, de esta forma el usuario no podría modificar la etiqueta de la acción hasta después de ser insertada en el diagrama.

Excepcionalmente estaría el usuario de la clase interesado en iterar sobre el conjunto de pasos en los cuales la acción aparece instanciada en el bloque de acciones, pero tampoco hay inconveniente en su uso, pues no modifican el estado del objeto.

En las clases para la representación de transiciones solamente fue excluida de la interfaz la función *bind_steps()*:

```
class SingleTransition{
//...
private:
        void bind_steps();

        // clases amigas
        friend class SFC;
public:
        // sin cambios, excepto en bind_steps()...
        //...
};

class DivTransition{
//...
private:
        void bind_steps();

        // clases amigas
        friend class SFC;
public:
        // sin cambios, excepto en bind_steps()...
```

```
        //...
};

class ConvTransition{
//...
private:
        void bind_steps();

        // clases amigas
        friend class SFC;
public:
        // sin cambios, excepto en bind_steps()...
        //...
};

class SimultTransition{
//...
private:
        void bind_steps();

        // clases amigas
        friend class SFC;
public:
        // sin cambios, excepto en bind_steps()...
        //...
};
```

Por último, la clase *Step* sufrió las correspondientes modificaciones:

```
class Step{
//...
private:
        // operaciones
        void bind_to(SingleTransition*);
        void bind_to(DivTransition*);
        void bind_to(ConvTransition*);
        void bind_to(SimultTransition*);

        void unbind(SingleTransition*);
        void unbind(DivTransition*);
```

```
        void unbind(ConvTransition*);
        void unbind(SimultTransition*);

        // clases amigas
        friend class SFC;
        friend class SingleTransition;
        friend class DivTransition;
        friend class ConvTransition;
        friend class SimultTransition;
public:
        // tipos miembros
        typedef tipo_dependiente_de la implementación single_iterator;
        typedef tipo_dependiente_de la implementación div_iterator;
        typedef tipo_dependiente_de la implementación conv_iterator;
        typedef tipo_dependiente_de la implementación simult_iterator;

        typedef tipo_dependiente_de la implementación block_iterator;
        typedef tipo_dependiente_de la implementación count_type;

        // iteradores
        single_iterator single_begin() const;
        single_iterator single_end() const;

        div_iterator div_begin() const;
        div_iterator div_end() const;

        conv_iterator conv_begin() const;
        conv_iterator conv_end() const;

        simult_iterator simult _begin() const;
        simult _iterator simult _end() const;

        block _iterator block_begin() const;
        block_iterator block_end() const;

        // constructor
        explicit Step(const std::string&);

        // operaciones sobre el bloque de acciones
        void insert_instance(const ActionInstance&);
        void insert_instance(block_iterator, const ActionInstance&);
```

```
        void erase_instance(block_iterator);
        void erase_instance(Action*);

        void clear_block();
};
```

La interfaz presentada con anterioridad para la clase SFC sí fue concebida para ser completamente utilizada por el usuario de la clase.

A propósito de la clase *SFC*: el diseño de la misma permite al usuario extender a conveniencia el conjunto de clases mediante el uso de la herencia a la vez que se obtienen los servicios básicos que provee la clase *SFC*. Por ejemplo, en una aplicación como el prototipo iniciado, en la que cada paso para su representación se asume localizado en un punto del plano, sería conveniente que como parte de la representación de cada paso, fueran incluidas las coordenadas cartesianas de su ubicación en el plano:

```
class MyStep: public Step{
//...
private:
        // coordenadas cartesianas en el plano
        float x;
        float y;
//...
};

void f(SFC& sfc)
{
        //...
        MyStep* s1 = new MyStep(/* ... */);
        sfc.insert(s1);
        //...
}
```

Si la clase SFC asemeja en gran manera a una clase contenedora (*containers*) desde cierto punto de vista, y fue concebida con la idea de permitir al usuario definir sus propios tipos para los elementos fundamentales de un diagrama SFC (pasos, acciones y transiciones) a conveniencia, ¿por qué no presentarla como una clase genérica al estilo de los contenedores estándar de C++? Por ejemplo:

```
template <class S, class STA, class SFCA,
          class STr, class DTr, class CTr,
          class SmTr>
class SFC{
//...
public:
        typedef     S       step_type;
        typedef     STA     st_action_type;
        typedef     SFCA    sfc_action_type;
        typedef     STr     single_tr_type;
        typedef     DTr     divergent_tr_type;
        typedef     CTr     convergent_tr_type;
        typedef     SmTr    simult_tr_type;
        //...

        insert(const step_type&);
//...
};
```

Esta alternativa fue rechazada debido a la complejidad y dificultad que impone primeramente sobre la necesaria comunicación e interrelación entre la clase *SFC* y el conjunto de clases ofrecidas por el usuario, y además sobre la implementación en general de la clase *SFC*. Ni siquiera se analizó con total profundidad si sería completamente factible la implementación de tal clase genérica.

2.4. Algoritmos usados en la implementación de las clases propuestas

Con anterioridad fueron discutidos los requerimientos fundamentales de la invariante de la clase *SFC*. El problema de establecer una invariante razonable para una clase no trivial está aparejado al problema de resolver cómo dicha invariante es mantenida por las funciones miembros de la clase. En el caso de la clase *SFC*, hubo algunas operaciones para las cuales surgió la necesidad de diseñar algoritmos con el fin de que la invariante de la clase (y del conjunto de clases en general) fuera mantenida una vez concluida la operación. En este epígrafe no sólo se describen tales algoritmos, sino que se abordan aspectos referentes a la representación utilizada en la implementación de la clase *SFC* y otras de las clases, que tienen que ver con la eficiencia de las operaciones mencionadas.

Entre las operaciones de la clase SFC, sólo ofrecen cierta complejidad en cuanto al mantenimiento de la invariante de la clase:

- las operaciones de inserción de transiciones (simples, disyuntivas divergentes, disyuntivas convergentes y simultáneas); y
- la eliminación de un paso en el diagrama.

El resto pueden ser consideradas operaciones relativamente simples.

2.4.1. Inserción de transiciones simples

El siguiente algoritmo fue diseñado para la inserción de transiciones simples en un diagrama SFC y ser empleado en la implementación de la función *SFC::insert(SingleTransition*)*:

Algoritmo de inserción de transiciones simples

Entrada: Una instancia *sfc* de la clase *SFC*, y una instancia *t* de la clase *SingleTransition*.

Salida: Instancia *sfc'* de la clase *SFC*, resultante de la inserción de la transición simple *t* en el diagrama *sfc*.

Método:

PASO 1. Hacer *sfc' = sfc*.

PASO 2. Buscar una transición simple *t'* en *sfc* tal que el paso anterior de *t'* coincida con el paso anterior de *t*. Si tal transición existe, ***continuar***; sino ir al **PASO 4**.

PASO 3. Sean *t = (a, c, p)* y *t' = (a, c', p')*. Si *p* y *p'* son pasos diferentes, entonces:

- Eliminar la transición *t'* de *sfc'*.
- Añadir una nueva transición disyuntiva divergente *dd = (a, {(c, p), (c',*

p')} al conjunto de transiciones disyuntivas divergentes del diagrama sfc'.

- Añadir la transición *dd* al conjunto de transiciones disyuntivas divergentes en las cuales están involucrados los pasos *p* y *p'*, respectivamente y ***retornar***.

En otro caso, ***abortar***.

PASO 4. Buscar una transición disyuntiva divergente *dd'* en *sfc'* cuyo paso anterior coincida con el paso anterior de *t*. Si tal transición existe, ***continuar***; sino ir al **PASO 6**.

PASO 5. Sea *t = (a, c, p)*. Insertar la rama *(c, p)* en la transición *dd'* y ***retornar***.

PASO 6. Buscar una transición simple *t'* en *sfc* tal que el paso posterior de *t'* coincida con el paso posterior de *t*. Si tal transición existe, ***continuar***; sino ir al **PASO 8**.

PASO 7. Sean *t = (a, c, p)* y *t' = (a', c', p)*. Si *a* y *a'* son pasos diferentes, entonces:

- Eliminar la transición *t'* de *sfc'* .
- Añadir una nueva transición disyuntiva convergente *dc = ({(c, a), (c', a')}, p)* al conjunto de transiciones disyuntivas divergentes del diagrama sfc'.
- Añadir la transición *dc* al conjunto de transiciones disyuntivas convergentes en las cuales están involucrados los pasos *p* y *p'*, respectivamente, ***retornar***.

En otro caso, ***abortar***.

PASO 8. Buscar una transición disyuntiva convergente *dc'* en *sfc'* cuyo paso posterior coincida con el paso posterior de *t*. Si tal transición existe, ***continuar***; sino ir al **PASO 10**.

PASO 9. Sea *t = (a, c, p)*. Insertar la rama *(c, p)* en la transición *dc'* y ***retornar***.

PASO 10. Sea *t = (a, c, p)*. Añadir *t* al conjunto de transiciones simples de *sfc'*, agregar *t* al conjunto de transiciones simples en las que están involucrados los pasos *a* y *p* respectivamente, y ***retornar***.

Este algoritmo – al igual que los restantes – asume que el diagrama de entrada satisface la invariante de la clase. Se trata entonces de realizar la operación de inserción (siempre que sea posible), de forma tal que el diagrama resultante cumpla con los requerimientos de la invariante de la clase *SFC*.

En consecuencia con lo anterior, en el PASO 2 se asume la existencia de a lo sumo una transición simple que cumple con la condición enunciada. Lo mismo ocurre en el PASO 6. Si se llega a la ejecución del PASO 3, nótese que la existencia de una transición simple con paso anterior a garantiza la no existencia de una transición disyuntiva divergente con paso anterior a, de modo que es seguro añadir la nueva transición. Algo semejante sucede en el PASO 7 con el paso posterior p. La demostración formal de que este y los restantes algoritmos mantienen la invariante de la clase *SFC* se deja al lector como ejercicio.

Las acciones que aparecen subrayadas (y sólo estas) son ejecutadas a través de operaciones en la interfaz de las clases. Por ejemplo, la eliminación de la transición simple en los pasos 3 y 7 es ejecutada a través de *SFC::erase()*. Sin embargo, no es correcto añadir una nueva transición simple en el PASO 10 mediante una invocación recursiva.

Por último, véase que no se tienen en cuenta detalles menores que resultan obvios en la invariante de la clase *SFC*, como por ejemplo, la pertenencia al diagrama de los pasos que forman parte de la transición a insertar. Una implementación completa debe incluir mecanismos de verificación con tal fin. Esta observación también es válida para el resto de los algoritmos presentados.

2.4.2. Inserción de transiciones disyuntivas divergentes

La inserción de transiciones disyuntivas divergentes resulta menos complicada. El siguiente algoritmo fue diseñado para la inserción de transiciones disyuntivas divergentes en un diagrama SFC y ser empleado en la implementación de la función *SFC::insert(DivTransition*)*:

Algoritmo de inserción de transiciones disyuntivas divergentes

Entrada: Una instancia *sfc* de la clase *SFC*, y una instancia *t* de la clase *DivTransition*.

Salida: Instancia *sfc'* de la clase *SFC*, resultante de la inserción de la transición disyuntiva divergente *t* en el diagrama *sfc*.

Método:

PASO 1. Hacer *sfc' = sfc*.

PASO 2. Buscar una transición simple *t'* en *sfc* tal que el paso anterior de *t'* coincida con el paso anterior de *t*. Si tal transición existe, ***continuar***; sino ir al **PASO 4**.

PASO 3. Sea *t = (a, c, p)*. Hacer lo siguiente:

- Eliminar la transición *t'* de *sfc'*.
- Añadir la nueva transición disyuntiva divergente *t* al conjunto de transiciones disyuntivas divergentes del diagrama *sfc'*.
- Para cada paso posterior *pp* de *t*, añadir la transición *t* al conjunto de transiciones disyuntivas divergentes en las cuales está involucrado *pp*.
- Insertar la rama *(c, p)* en la transición *t*, y ***retornar***.

PASO 4. Buscar una transición disyuntiva divergente *dd'* en *sfc'* cuyo paso anterior coincida con el paso anterior de *t*. Si tal transición existe, ***continuar***; sino ir al **PASO 6**.

PASO 5. Insertar en *dd'* el conjunto de ramas de *t*, y ***retornar***.

PASO 6. Añadir *t* al conjunto de transiciones disyuntivas divergentes de *sfc'*. Para cada paso posterior de *t*, así como para el paso anterior de *t*, añadir la transición *t* al conjunto de transiciones disyuntivas divergentes en las cuales el paso está involucrado; y ***retornar***.

Como en el caso previo, se supone que el diagrama de entrada ***sfc*** satisface la invariante de la clase. La nueva transición es insertada siempre que sea posible, de manera que el diagrama resultante ***sfc'*** cumpla también con los requerimientos de la invariante para la clase *SFC*.

En virtud de lo anterior, se asume en el PASO 2 la existencia de a lo sumo una transición simple en ***sfc*** que satisface la propiedad enunciada. Si se llega a la ejecución del PASO 3, nótese que la existencia de una transición simple en el PASO 2 con paso anterior igual al de *t*, garantiza la no existencia de una transición disyuntiva divergente con igual paso anterior. Por consiguiente, es seguro añadir la nueva transición *t* al diagrama toda vez que ha sido eliminada la transición simple. Igualmente, si se llega a la ejecución del PASO 6 del algoritmo, en tal punto queda garantizada la no existencia en el diagrama de transición simple o disyuntiva divergente alguna con igual paso anterior que *t,* por ende es seguro añadir *t* al conjunto de transiciones disyuntivas divergentes. La demostración formal de que este y los restantes algoritmos mantienen la invariante de la clase *SFC* se deja al lector como ejercicio.

Las acciones que aparecen subrayadas (y sólo estas) son ejecutadas a través de operaciones en la interfaz de las clases. No se tienen en cuenta detalles menores que resultan obvios en la invariante de la clase *SFC*, como por ejemplo, la pertenencia al diagrama de los pasos que forman parte de la transición a insertar. Una implementación completa debe incluir mecanismos de verificación con tal fin. Esta observación también es válida para el resto de los algoritmos presentados.

2.4.3. Inserción de transiciones disyuntivas convergentes

La estrategia aplicada para la inserción de transiciones disyuntivas convergentes es muy similar a la de inserción de transiciones disyuntivas divergentes. El siguiente algoritmo fue diseñado para la inserción de transiciones disyuntivas convergentes en un diagrama SFC y ser empleado en la implementación de la función *SFC::insert(ConvTransition*)*:

Algoritmo de inserción de transiciones disyuntivas convergentes

Entrada: Una instancia *sfc* de la clase *SFC*,
y una instancia *t* de la clase *ConvTransition*.

Salida: Instancia *sfc'* de la clase *SFC*, resultante de la inserción de la transición disyuntiva convergente *t* en el diagrama *sfc*.

Método:

PASO 1. Hacer *sfc' = sfc*.

PASO 2. Buscar una transición simple *t'* en *sfc* tal que el paso posterior de *t'* coincida con el paso posterior de *t*. Si tal transición existe, ***continuar***; sino ir al **PASO 4**.

PASO 3. Sea *t = (a, c, p)*. Hacer lo siguiente:

- Eliminar la transición *t'* de *sfc'*.
- Añadir la nueva transición disyuntiva convergente *t* al conjunto de transiciones disyuntivas convergentes del diagrama *sfc'*.
- Para cada paso anterior *pa* de *t*, añadir la transición *t* al conjunto de transiciones disyuntivas convergentes en las cuales está involucrado *pa*.
- Insertar la rama *(c, a)* en la transición *t*, y ***retornar***.

PASO 4. Buscar una transición disyuntiva convergente *dc'* en *sfc'* cuyo paso posterior coincida con el paso posterior de *t*. Si tal transición existe, ***continuar***; sino ir al **PASO 6**.

PASO 5. Insertar en *dc'* el conjunto de ramas de *t*, y ***retornar***.

PASO 7. Añadir *t* al conjunto de transiciones disyuntivas convergentes de *sfc'*. Para cada paso anterior de *t*, así como para el paso posterior de *t*, añadir la transición *t* al conjunto de transiciones disyuntivas convergentes en las cuales el paso está involucrado; y ***retornar***.

Se asume que el diagrama de entrada ***sfc*** satisface la invariante de la clase. La nueva transición es insertada siempre que sea posible, de manera que el diagrama resultante ***sfc'*** cumpla también con los requerimientos de la invariante para la clase *SFC*.

En virtud de lo anterior, se asume en el PASO 2 la existencia de a lo sumo una transición simple en ***sfc*** que satisface la propiedad enunciada. Si se llega a la ejecución del PASO 3, nótese que la existencia de una transición simple en el PASO 2 con paso posterior igual al de *t*, garantiza la no existencia de una transición disyuntiva convergente con igual paso posterior. Por consiguiente, es seguro añadir la nueva transición *t* al diagrama toda vez que ha sido eliminada la transición simple. Igualmente, si se llega a la ejecución del PASO 6 del algoritmo, en tal punto queda garantizada la no existencia en el diagrama de transición simple o disyuntiva convergente alguna con igual paso posterior que *t,* por ende es seguro añadir *t* al conjunto de transiciones disyuntivas convergentes. La demostración formal de que este y los restantes algoritmos mantienen la invariante de la clase *SFC* se deja al lector como ejercicio.

Las acciones que aparecen subrayadas (y sólo estas) son ejecutadas a través de operaciones en la interfaz de las clases. No se tienen en cuenta detalles menores que resultan obvios en la invariante de la clase *SFC*, como por ejemplo, la pertenencia al diagrama de los pasos que forman parte de la transición a insertar. Una implementación completa debe incluir mecanismos de verificación con tal fin. Esta observación también es válida para el resto de los algoritmos presentados.

2.4.4. Inserción de transiciones simultáneas

El algoritmo de inserción de transiciones simultáneas es realmente muy simple. Solamente es necesario verificar si existe en el diagrama alguna transición simultánea con igual conjunto de pasos anteriores e igual conjunto de pasos posteriores que la transición que se desea insertar.

La parte interesante radica en escoger una representación conveniente para las clases *SFC* y *SimultTransition*, de manera que esta operación pueda ser implementada con una eficiencia aceptable.

Como fue mencionado con anterioridad, los objetos correspondientes a los elementos fundamentales de un SFC contenidos dentro de un objeto de la clase SFC son inequívocamente identificables por sus direcciones físicas en memoria. En particular, ello es válido para el caso de las transiciones simultáneas. Sin embargo, al intentar insertar una nueva transición simultánea, no es posible determinar la equivalencia o no entre transiciones a través de la comparación de direcciones físicas, pues la nueva transición no pertenece aun al diagrama. Dos transiciones simultáneas son equivalentes a la hora de la inserción si y sólo si los conjuntos de pasos anteriores en ambas transiciones son equivalentes, y además los conjuntos de pasos posteriores en las mismas también son equivalentes.

Una forma de comprobar si existe transición simultánea en el diagrama equivalente a una nueva transición *t* que se desea insertar, consiste en recorrer exhaustivamente las transiciones simultáneas en el diagrama y comprobar, para cada transición simultánea *s*, la equivalencia los conjuntos de pasos anteriores y posteriores de *s* y *t*. Dejando a un lado los asuntos referentes a la comparación de conjuntos, esta alternativa es por sí sola muy ineficiente, por ejemplo, para el caso muy común en que no exista una transición simultánea en el diagrama que sea equivalente a la que se desea insertar. Esto condujo a la búsqueda de una mejor alternativa.

Una solución consiste en establecer un orden entre conjuntos de pasos – lo cual es posible, como se verá más adelante – de manera que las transiciones simultáneas en el diagrama sean mantenidas en orden con respecto al orden existente entre los conjuntos de pasos anteriores de las mismas. De esta manera, es posible encontrar el conjunto de transiciones en el diagrama cuyo conjunto de pasos anteriores es equivalente al de la transición que se desea insertar en $O(\log n)$ respecto al número de transiciones simultáneas en el diagrama; esto sin considerar la complejidad de las operaciones de comparación entre

conjuntos. Nótese que tal conjunto de transiciones puede en algunos casos ser vacío, pero también puede contener más de un elemento. De esta forma, sólo sería necesario chequear los conjuntos de pasos posteriores para dichas transiciones.

En virtud de lo anterior, se implementó una clase *simulttr_ptrcmp* para la comparación de transiciones simultáneas de acuerdo a sus conjuntos de pasos anteriores, de manera tal que la clase contenedora *std::multiset* de la Librería Estándar de C++ pudiera ser utilizada para representar el conjunto de transiciones simultáneas en la clase *SFC*:

```
class SFC{
private:
        // representación
        //...
        std::multiset<SimultTransition*, simulttr_ptr_cmp> simult_transitions;
public:
        //...
};

class simulttr_ptr_cmp: public std::binary_function<SimultTransition*,
                                                    SimultTransition*, bool>
{
public:
        bool operator()(const SimultTransition*, const SimultTransition*) const;
};
```

La comparación por defecto para el tipo *SimultTransition** ofrecida por la librería estándar no ofrece la semántica deseada. Nótese que se utilizó *std::multiset* y no *std::set* porque de hecho puede existir más de una transición simultánea con el mismo conjunto de pasos anteriores.

Otro problema tiene que ver con establecer un orden para conjuntos de pasos, y además con la eficiencia de la comparación de dos conjuntos de pasos; en particular, la comparación de conjuntos de pasos anteriores en transiciones simultáneas.

Existe $<_d$ un orden para las direcciones físicas de memoria. Esto permite contar con un orden para los pasos en un diagrama, dado que los mismos son identificados inequívocamente por sus direcciones físicas. El orden $<_p$ que proponemos para la comparación de conjuntos de pasos es el siguiente:

Orden para conjuntos de pasos

El orden $<_p$ para conjuntos de pasos se define de esta forma:

Sean A y B dos conjuntos de pasos arbitrarios, de n y m elementos respectivamente; y sean $\delta_a = a_1, a_2, ..., a_n$ y $\delta_b = b_1, b_2, ..., b_m$ secuencias conformadas a partir de todos los elementos de A y B, respectivamente, de manera tal que $a_1 <_d a_2 <_d ... <_d a_n$ y $b_1 <_d b_2 <_d ... <_d b_m$.

Se tiene que $A <_p B$ si y sólo si la secuencia δ_a es lexicográficamente menor que la secuencia δ_b respecto al orden $<_d$.

El orden $<_p$ es el utilizado por la clase *simulttr_ptrcmp*. La demostración formal de que este orden define un ordenamiento débil estricto (*strict weak ordering* [Str97]), según los requerimientos de la clase *std::multiset*, se deja al lector como ejercicio.

Visto esto, sería conveniente entonces mantener ordenados los pasos anteriores en una transición simultánea según el orden $<_d$, de manera que no hubiera necesidad de ordenar los elementos a la hora de la comparación de dos conjuntos; así la comparación entre dos conjuntos de pasos de n y m elementos respectivamente tendría una complejidad de $O(mín(n, m))$, versus $O(k \log k)$ si se necesita ordenar, donde $k = max(m, n)$. Pero precisamente la representación escogida para el conjunto de pasos anteriores en una transición simultánea permite mantener ordenados los pasos de esa forma. Además de ello, los pasos posteriores también son mantenidos en orden, lo cual hace más eficiente la comprobación final de la igualdad entre los conjuntos de pasos posteriores.

2.4.5. Eliminación de pasos

La operación de eliminación de pasos es interesante porque, con el fin de mantener la invariante establecida para la clase *SFC* y el conjunto de clases en general, es necesario realizar operaciones con las transiciones en las que el paso a eliminar está involucrado. En particular, no puede existir una transición en un diagrama SFC que contenga un paso anterior o posterior que no pertenezca al conjunto

de pasos del diagrama. Este aspecto no se hizo explícito cuando se presentó la invariante escogida por haberse considerado obvio.

El siguiente algoritmo fue diseñado para la eliminación de pasos en un diagrama SFC y ser empleado en la implementación de la función *SFC::erase(Step*)*:

Algoritmo de eliminación de pasos

Entrada: Una instancia *sfc* de la clase *SFC*, y una instancia *p* de la clase *Step*.

Salida: Instancia *sfc'* de la clase *SFC*, resultante de la eliminación del paso *p* del diagrama *sfc*.

Método:

PASO 1. Hacer *sfc' = sfc*.

PASO 2. Para cada transición simple *s* en *sfc* en la que *p* está involucrado, eliminar *s* del diagrama *sfc'*.

PASO 3. Para cada transición disyuntiva divergente *dd* en *sfc'* en la que *p* está involucrado, hacer lo siguiente:

- Si *p* es el paso anterior en *dd*, eliminar *dd* del conjunto de transiciones disyuntivas divergentes en *sfc'* y pasar a la próxima transición.
- Si *dd* tiene exactamente dos pasos posteriores, sea $dd = (a, \{(c, p), (c_1, p_1)\})$. Eliminar *dd* del conjunto de transiciones disyuntivas divergentes en *sfc'*, agregar al conjunto de transiciones simples de sfc' la nueva transición $(a, c1, p1)$ y pasar a la próxima transición.
- Eliminar en *dd* la rama en la cual *p* es el paso.

PASO 4. Para cada transición disyuntiva convergente *dc* en *sfc'* en la que *p* está involucrado, hacer lo siguiente:

- Si *p* es el paso posterior en *dc*, eliminar *dc* del conjunto de transiciones disyuntivas convergentes en *sfc'* y pasar a la próxima transición.
- Si *dc* tiene exactamente dos pasos anteriores, sea $dc = (\{(c, p), (c_1, p_1)\}, a)$. Eliminar *dc* del conjunto de transiciones disyuntivas convergentes en *sfc'*, agregar al conjunto de transiciones simples de *sfc'* la nueva transición (p_1, c_1, a) y pasar a la próxima transición.
- Eliminar en *dc* la rama en la cual *p* es el paso.

PASO 5. Para cada transición simultánea *sm* en *sfc'* en la que *p* está involucrado, hacer lo siguiente:

- Si *p* es el único paso anterior en la transición, o bien *p* es el único paso posterior, o bien *p* es uno de los dos pasos posteriores y la transición tiene un único paso anterior, o bien *p* es uno de los dos pasos anteriores y la transición tiene un único paso posterior, eliminar *sm* del conjunto de transiciones

simultáneas en *sfc'* y pasar a la próxima transición simultánea.
- Eliminar en *sm* el paso *p*.

PASO 6. Para cada acción *a* que aparece instanciada en el bloque de acciones de *p*, eliminar en *a* el paso *p* del conjunto de pasos en los que *a* está instanciada.

PASO 7. Remover el paso *p* del conjunto de pasos en *sfc'*.

Nótese en el PASO 2 que son a lo sumo dos las transiciones simples en las que un paso puede estar involucrado; de otra forma se estaría violando la invariante de la clase *SFC*.

Las transiciones simples en los que el paso a eliminar está involucrado son eliminadas del diagrama. Si el paso a eliminar constituye el paso anterior de una transición disyuntiva divergente, no queda otra alternativa que eliminar la transición del diagrama. Lo mismo ocurre si el paso a eliminar constituye el paso posterior de una transición disyuntiva convergente. Si por otro lado, en una transición disyuntiva divergente el paso a eliminar constituye uno de los dos únicos pasos posteriores, o si en una transición disyuntiva divergente el paso a eliminar es uno de los dos únicos pasos anteriores, entonces la transición disyuntiva divergente (o convergente, según el caso) pasaría a ser una transición simple. Finalmente, si en una transición simultánea es imposible eliminar el paso, la transición es eliminada del diagrama.

La eliminación de ramas siempre que sea posible parece ser una estrategia preferible antes que la eliminación de la transición completa. Si el usuario de la clase *SFC* desea eliminar la transición completamente y no tan sólo la rama correspondiente al paso, la clase *SFC* brinda operaciones para ello. Igualmente pudo haberse incluido una operación adicional para la eliminación de un paso en la cual fueran eliminadas todas las transiciones en las que un paso está involucrado.

La presentación de este algoritmo ayuda a comprender por qué, para implementar de manera eficiente la operación de eliminación de un paso en el diagrama – como se mencionó con anterioridad – es conveniente mantener para cada paso del diagrama el conjunto de transiciones de cada tipo en las que el mismo está involucrado.

Como en los casos anteriores, aquí no se tuvieron en cuenta aspectos menores, como por ejemplo, si el paso a eliminar constituye el paso inicial en el diagrama. Una implementación completa debe incluir mecanismos para lidiar con tales situaciones.

Las acciones que aparecen subrayadas (y sólo estas) son ejecutadas a través de operaciones en la interfaz de las clases. Al respecto, obsérvese que en los pasos 3 y 4 no es necesario utilizar la operación de inserción de la interfaz de la clase *SFC* a la hora de añadir la nueva transición simple; en ese punto ya es seguro agregar directamente la nueva transición (lo cual es más eficiente) sin posibilidad de que sea violada la invariante de la clase.

2.5. Algoritmos para la representación geométrica de elementos de SFC

El conceder al programador de PLCs (usuario final de la herramienta de edición) control total sobre la disposición de los pasos en el diagrama, impuso el reto de brindar una representación gráfica conveniente y comprensible para los elementos en el diagrama, en la medida de lo posible próxima a la representación usual.

La dificultad realmente radica en buscar una representación adecuada para las transiciones; el resto de los elementos del diagrama – pasos, acciones, y bloques de acciones – pueden ser representados sin mayor contratiempo. Por consiguiente, este epígrafe se circunscribe a la presentación de algoritmos para la representación geométrica de transiciones.

Los algoritmos propuestos a continuación fueron concebidos bajo el principio fundamental de que la representación gráfica obtenida para una transición permitiera imaginar o figurar la sintaxis de la misma sin dificultad o ambigüedades.

2.5.1. Preliminares

Antes de proceder a la presentación de los algoritmos diseñados, es preciso hacer algunas observaciones e introducir cierta notación.

Primeramente, indicar que la representación gráfica del diagrama y sus elementos se asume que será hecha sobre el plano, en el cual se encuentra situado un sistema de coordenadas cartesiano imaginario.

La representación gráfica de las transiciones que se pretende obtener se basa únicamente en la utilización de segmentos horizontales y verticales. De este modo, la representación de una transición consiste en un conjunto de segmentos en el plano. En todos los casos se trata de utilizar la menor cantidad de segmentos y de que los mismos tengan la menor longitud posible, siempre y cuando permitan ofrecer una representación razonable para la transición.

Los pasos, por su parte, son representados mediante rectángulos cuyos lados son igualmente paralelos a los ejes *x* e *y*. Se asume que cada paso *p* tiene una ubicación respecto al sistema de coordenadas. La ubicación no es más que un punto del plano, cuya abscisa y ordenada denotaremos por *x(p)* y *y(p)*, respectivamente. Así, la representación gráfica de *p* consiste un rectángulo con centro en *(x(p), y(p))* y lados de longitud fija ***LP*** (longitud del lado paralelo al eje *x*) y ***AP*** (longitud del lado paralelo al eje *y*).

Se analizó además el problema de poder precisar, a partir la representación gráfica de una transición y sus respectivos conjuntos de pasos, cuáles de ellos constituyen pasos anteriores y cuáles de ellos pasos posteriores en la transición. Después de analizar múltiples alternativas, se decidió que la solución con menos inconvenientes es la siguiente:

- Si un paso *p* en una transición *t* constituye un paso anterior, entonces existe un segmento vertical en la representación de *t* tal que uno de sus extremos yace sobre el segmento inferior del rectángulo que representa a *p*, y el otro extremo yace por debajo del mismo.
- Si un paso *p* en una transición *t* constituye un paso posterior, entonces existe un segmento vertical en la representación de *t* tal que uno de sus extremos yace sobre el segmento superior del rectángulo que representa a *p*, y el otro extremo yace por encima del mismo.

En ningún caso se trata de representar la pertenencia de un paso a una transición mediante segmentos horizontales con un extremo yaciente en uno de los lados verticales del rectángulo que sirve de representación al paso. Dicho de manera informal, las transiciones parecen “salir” de los pasos anteriores “por debajo” de los mismos, y “entrar” en los pasos posteriores “por encima” de estos.

Finalmente, si para cierta transición los pasos están ubicados de forma cercana a la disposición que toman en la representación usual descrita en el Capítulo 1, entonces el algoritmo debe ofrecer una representación que emule la representación usual correspondiente al tipo de transición en cuestión.

2.5.2. Transiciones simples

La representación gráfica de una transición simple es la más sencilla entre todas, y sin embargo requiere la consideración de varios casos particulares relacionados con la ubicación en el diagrama del paso posterior con respecto al paso anterior de la transición.

A groso modo, se trata de "conectar" el paso anterior con el paso posterior de la transición mediante segmentos contiguos siguiendo los principios enunciados previamente, y de manera tal que los segmentos utilizados para representar la transición no sólo no tengan intercepción con los rectángulos utilizados para representar los pasos de la transición, sino que se encuentren a una distancia prudencial de éstos (excepto aquellos con obligatorio contacto).

Además de las constantes ***LP*** y ***AP*** definidas anteriormente, se definen otras constantes utilizadas en éste y los restantes algoritmos, cuyo valor exacto depende de la implementación:

LC: longitud del segmento utilizado para representar una condición.

LR: longitud por defecto de los segmentos en la transición en los que uno de los extremos tiene contacto con uno de los rectángulos que representan los pasos en la transición.

SP: separación mínima permitida entre los segmentos de la transición y los rectángulos que representan los pasos en la misma.

A continuación es presentado formalmente el algoritmo diseñado para la representación de transiciones simples:

Algoritmo para la representación de transiciones simples

Entrada: Una transición simple $t = (a, c, p)$.

Salida: Un conjunto T de segmentos en el plano que constituye la representación gráfica correspondiente a t.

Método:

PASO 1. Hacer $T = \Phi$.

PASO 2. Si $y(a) - y(p) >$ ***AP***, ***continuar***; sino ir al **PASO 5**.

PASO 3. Si $x(a) = x(p)$, entonces:

- Hacer $T = T \cup \{((x(a),\ y(a) - \mathbf{AP}/2),\ (x(p),\ y(p) + \mathbf{AP}/2))\}$.
- Hacer

  $T = T \cup \{((x(a) - \mathbf{LC}/2,\ (y(a) + y(p))/2),\ (x(p) + \mathbf{LC}/2,\ (y(a) + y(p))/2))\}$.
- ***retornar***.

PASO 4. Construir T de la siguiente manera:

- Hacer $T = T \cup \{((x(a),\ y(a) - \mathbf{AP}/2),\ (x(a),\ (y(a) + y(p))/2))\}$.
- Hacer $T = T \cup \{((x(p),\ y(p) + \mathbf{AP}/2),\ (x(p),\ (y(a) + y(p))/2))\}$.
- Hacer $T = T \cup \{((x(a),\ (y(a) + y(p)/2),\ (x(p),\ (y(a) + y(p))/2))\}$.
- Hacer $T = T \cup \{(((x(a) + x(p))/2,\ (y(a) + y(p))/2 + \mathbf{LC}/2),\ ((x(a) + x(p))/2,\ (y(a) + y(p)/2 - \mathbf{LC}/2))\}$.
- ***retornar***.

PASO 5. Si $|x(a) - x(p)| <$ ***LP*** + ***SP***, ***continuar***; sino ir al **PASO 8**.

PASO 6. Si $x(p) < x(a)$, entonces:

- Hacer $T = T \cup \{((x(a),\ y(a) - \mathbf{AP}/2),\ (x(a),\ y(a) - \mathbf{AP}/2 - \mathbf{LR}))\}$.
- Hacer $T = T \cup \{((x(p),\ y(p) + \mathbf{AP}/2),\ (x(p),\ y(p) + \mathbf{AP}/2 + \mathbf{LR}))\}$.
- Hacer $T = T \cup \{((x(a),\ y(a) - \mathbf{AP}/2 - \mathbf{LR}),\ (x(p) - \mathbf{LP}/2 - \mathbf{SP},\ y(a) - \mathbf{AP}/2 - \mathbf{LR}))\}$.
- Hacer $T = T \cup \{((x(p),\ y(p) + \mathbf{AP}/2 + \mathbf{LR}),\ (x(p) - \mathbf{LP}/2 - \mathbf{SP},\ y(p) + \mathbf{AP}/2 + \mathbf{LR}))\}$.
- Hacer $T = T \cup \{((x(p) - \mathbf{LP}/2 - \mathbf{SP},\ y(p) + \mathbf{AP}/2 + \mathbf{LR}))\},\ (x(p) - \mathbf{LP}/2 -$

***SP**, y(a) - **AP**/2- **LR**))}.*

- Hacer *T = T U {(((x(p) - **LP**/2 - **SP** – **LC**/2, (y(a) + y(p))/2), ((x(p) - **LP**/2 - **SP** + **LC**/2, (y(a) + y(p))/2))}.*
- ***retornar***.

PASO 7. Construir *T* de la siguiente manera:

- Hacer *T = T U {((x(a), y(a) - **AP**/2), (x(a), y(a) - **AP**/2- **LR**))}*.
- Hacer *T = T U {((x(p), y(p) + **AP**/2), (x(p), y(p) + **AP**/2 + **LR**))}*.
- Hacer *T = T U {((x(a), y(a) - **AP**/2- **LR**), (x(p) + **LP**/2 + **SP**, y(a) - **AP**/2- **LR**))}*.
- Hacer *T = T U {((x(p), y(p) + **AP**/2 + **LR**), (x(p) + **LP**/2 + **SP**, y(p) + **AP**/2 + **LR**))}*.
- Hacer *T = T U {((x(p) + **LP**/2 + **SP**, y(p) + **AP**/2 + **LR**))}, (x(p) + **LP**/2 + **SP**, y(a) - **AP**/2- **LR**))}*.
- Hacer *T = T U {((x(p) + **LP**/2 + **SP** – **LC**/2, (y(a) + y(p))/2), ((x(p) + **LP**/2 + **SP** + **LC**/2, (y(a) + y(p))/2))}*.
- ***retornar***.

PASO 8. Construir *T* de la siguiente manera:

- Hacer *T = T U {((x(a), y(a) - **AP**/2), (x(a), y(a) - **AP**/2- **LR**))}*.
- Hacer *T = T U {((x(p), y(p) + **AP**/2), (x(p), y(p) + **AP**/2 + **LR**))}*.
- Hacer *T = T U {((x(a), y(a) - **AP**/2- **LR**), ((x(a) + x(p))/2, y(a) - **AP**/2- **LR**))}*.
- Hacer *T = T U {((x(p), y(p) + **AP**/2 + **LR**), ((x(a) + x(p))/2, y(p) + **AP**/2 + **LR**))}*.
- Hacer *T = T U {((x(a) + x(p))/2, y(p) + **AP**/2 + **LR**))}, ((x(a) + x(p))/2, y(a) - **AP**/2- **LR**))}*.
- Hacer *T = T U {((((x(a) + x(p))/2 – **LC**/2, (y(a) + y(p))/2), (((x(a) +*

*x(p))/2 + **LC**/2, (y(a) + y(p))/2))}.*

- ***retornar***.

La condición en el PASO 2 corresponde a si el paso anterior en la transición se encuentra “suficientemente por encima”. En tal caso, si ambos pasos están alineados verticalmente, se utilizan dos segmentos para representar la transición (PASO 3) conforme a la representación usual; de otro modo es necesario utilizar cuatro (PASO 4).

El resto de los casos corresponden a cuando el paso anterior se encuentra “por debajo” del paso posterior. La condición en el PASO 5 corresponde a si es posible incluir un segmento vertical entre ambos pasos para representar la transición; el caso negativo corresponde al PASO 6 y al PASO 7; el PASO 8 cubre el caso afirmativo.

La siguiente figura ayuda a una mejor comprensión de las posibles alternativas en cada paso del algoritmo (figura 14).

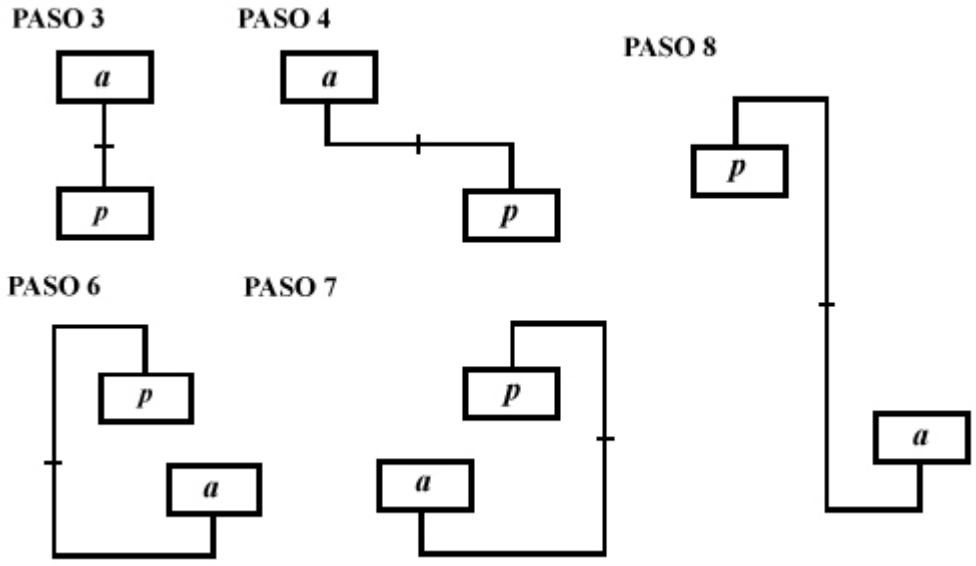

Figura 14. Representación obtenida para diferentes disposiciones relativas de los pasos en una transición simple.

2.5.3. Transiciones disyuntivas divergentes

Proponer un algoritmo para la representación de transiciones disyuntivas divergentes requirió de un mayor esfuerzo. Básicamente, la idea consiste en "conectar" los pasos involucrados en la transición de la manera siguiente:

1. Para cada paso en la transición, incluir en la representación un *conjunto de segmentos asociado al paso* que sirven para "conectar" al mismo con el resto de los pasos de la transición.

2. Se incluye en la representación un *segmento vertical distinguido,* que sirve para "conectar" los conjuntos de segmentos correspondientes a cada uno de los pasos de la transición.

3. El *conjunto de segmentos* asociado al paso anterior a consta de un único segmento vertical que "sale" por debajo del paso, de longitud fija ***LR*** (PASO 2 en el algoritmo).

4. El segmento horizontal mencionado en (2) yacerá sobre la recta *y = y(a) -* ***AP****/2 –* ***LR*** (l en el PASO 1 del algoritmo). Esto es importante para calcular el *conjunto de segmentos* asociado a cada paso posterior de la transición. Sin embargo, las coordenadas exactas de los extremos del segmento no son conocidas hasta después de computados dichos conjuntos de segmentos.

5. Para cada paso posterior p es calculado el *conjunto de segmentos asociado a* p, en dependencia de la posición de p respecto a la recta mencionada en (4) y al paso anterior a de la transición, de la siguiente manera:

   Si *p* está *completamente por debajo* de la recta l (PASO 3 A.), el conjunto de segmentos asociado a p consta solamente de dos elementos; el primero de ellos "sale" de la recta y "entra" en el rectángulo que representa a p, y el segundo simplemente corresponde a la condición asociada al paso. En el caso contrario (PASO 3 B.), el conjunto de segmentos consta de cuatro elementos y su cómputo es algo más complicado. Se toma en cuenta la posición relativa de p respecto a a, para tratar de que ninguno de los segmentos del conjunto intercepte al rectángulo que representa al paso anterior a, y que el *segmento horizontal* mencionado en (2) finalmente tenga la menor longitud posible.

Para cada paso posterior p, existe dentro del conjunto de segmentos asociados a p un *segmento vertical distinguido*, que sirve para "conectar" la recta l con el resto de los segmentos del conjunto. A medida que se calcula el conjunto de segmentos asociado a cada paso posterior, se computa además la ordenada del *segmento vertical distinguido más a la izquierda* y la ordenada del *segmento vertical más a la derecha*.

6. Finalmente, se añade a la representación el *segmento horizontal* que sirve para "conectar" los conjuntos de segmentos asociados a cada paso en la transición, el cual yace sobre la recta l y se extiende desde la ordenada del *segmento vertical distinguido más a la izquierda* hasta la ordenada del *segmento vertical distinguido más a la derecha* (PASO 4).

A continuación es presentado formalmente el algoritmo diseñado para la representación de transiciones disyuntivas divergentes:

Algoritmo para la representación de transiciones disyuntivas divergentes

Entrada: Una transición disyuntiva divergente $t = (a, R)$.

Salida: Un conjunto T de segmentos en el plano que constituye la representación gráfica correspondiente a t.

Método:

PASO 1. Hacer $T = \Phi$, $l = y(a) - \mathbf{LR} - \mathbf{AP}/2$, $minX = x(a)$, y $maxX = x(a)$.

PASO 2. Hacer $T = T \cup \{((x(a), y(a) - \mathbf{AP}/2), (x(a), y(a) - \mathbf{AP}/2 - \mathbf{LR}))\}$.

PASO 3. Para cada paso posterior p de t, hacer lo siguiente:

A. Si $y(p) < l - \mathbf{AP}/2$, entonces:

– Hacer $T = T \cup \{((x(p), l), (x(p), y(p) + \mathbf{AP}/2))\}$.

– Hacer $T = T \cup \{((x(p) - \mathbf{LC}/2, [l + y(p) + \mathbf{AP}/2]/2), (x(p) + \mathbf{LC}/2, [l +$

y(p) + ***AP****/2]/2))}*.

– Hacer *minX = min(minX, x(p))*, y *maxX = max(maxX, x(p))*.

– ***continuar*** con el próximo paso posterior de la transición.

B. sino:

– Si *x(a) – x(p) >* ***LP*** + ***SP***, o bien *x(a) – x(p) ≤ 0* y *x(p) – x(a) ≤* ***LP*** + ***SP***, entonces hacer *d =* ***LP*** + ***SP***, sino hacer *d =* ***–LP – SP***.

– Hacer *T = T U {((x(p) + d, l), (x(p) + d, y(p) +* ***AP****/2 +* ***LR****))}*.

– Hacer *T = T U {((x(p), y(p) +* ***AP****/2), (x(p), y(p) +* ***AP****/2 +* ***LR****))}*.

– Hacer *T = T U {((x(p) –* ***LC****/2, y(p) +* ***AP****/2 +* ***LR****/2), (x(p) +* ***LC****/2, y(p) +* ***AP****/2 +* ***LR****/2))}*.

– Hacer *T = T U {((x(p), y(p) +* ***AP****/2 +* ***LR****), (x(p) + d, y(p) +* ***AP****/2 +* ***LR****))}*.

– Hacer *minX = min(minX, x(p)+d)*, y *maxX = max(maxX, x(p)+d)*.

– ***continuar*** con el próximo paso posterior de la transición.

PASO 4. Hacer *T = T U {((minX, l), (maxX, l))}*.

En la siguiente figura se muestra la representación gráfica de dos transiciones disyuntivas divergentes obtenidas mediante la aplicación del algoritmo propuesto (figura 15). En la transición que aparece en la parte superior, los pasos se encuentran en una disposición próxima a la utilizada en la representación usual. Nótese entonces cómo la representación de la transición emula la representación usual para transiciones disyuntivas divergentes. En la parte inferior aparece la representación gráfica obtenida mediante la aplicación del algoritmo para una transición en la cual la locación relativa de los pasos es menos afortunada. Sin embargo, no hay dificultad en interpretar sin ambigüedad la sintaxis de la misma. Por supuesto, es posible imaginar casos extremos en los que la representación de la transición y del diagrama en general serían prácticamente ilegibles; es responsabilidad del programador de PLCs – quien dispondrá la ubicación de los pasos sobre el plano – la organización general del programa, en particular el

evadir tales casos siempre que sea posible, necesario o de su interés, así como el lidiar con el solapamiento de los objetos representados.

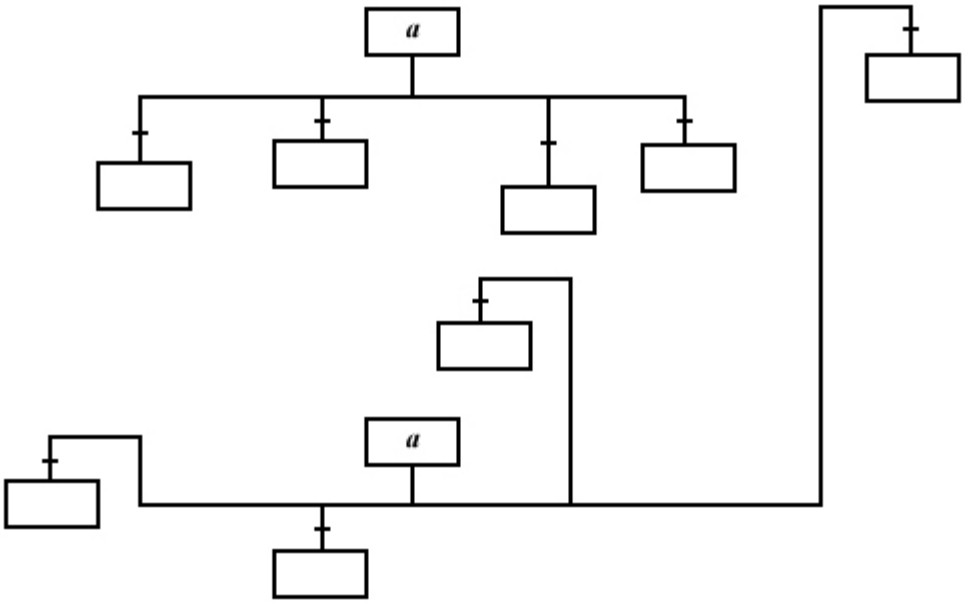

Figura 15. Representación obtenida para diferentes transiciones disyuntivas divergentes.

2.5.4. Transiciones disyuntivas convergentes

El algoritmo diseñado para obtener la representación gráfica de transiciones disyuntivas convergentes es muy similar al anterior para transiciones disyuntivas divergentes; por ende, a continuación solamente es presentada la definición formal del mismo:

Algoritmo para la representación de transiciones disyuntivas divergentes

Entrada: Una transición disyuntiva convergente $t = (R, p)$.

Salida: Un conjunto T de segmentos en el plano que constituye la representación gráfica correspondiente a t.

Método:

PASO 1. Hacer $T = \Phi$, $l = y(p) + \boldsymbol{LR} + \boldsymbol{AP}/2$, $minX = x(p)$, y $maxX = x(p)$.

PASO 2. Hacer $T = T \cup \{((x(p),\ y(p) + \boldsymbol{AP}/2),\ (x(p),\ y(p) + \boldsymbol{AP}/2 + \boldsymbol{LR}))\}$.

PASO 3. Para cada paso anterior a de t, hacer lo siguiente:

- Si $y(a) > l + \boldsymbol{AP}/2$, entonces:
  - Hacer $T = T \cup \{((x(a),\ l),\ (x(a),\ y(a) - \boldsymbol{AP}/2))\}$.
  - Hacer $T = T \cup \{((x(a) - \boldsymbol{LC}/2,\ [l + y(a) - \boldsymbol{AP}/2]/2),\ (x(a) + \boldsymbol{LC}/2,\ [l + y(a) - \boldsymbol{AP}/2]/2))\}$.
  - Hacer $minX = min(minX, x(a))$, y $maxX = max(maxX, x(a))$.
  - ***continuar*** con el próximo paso anterior de la transición.
- Si $x(p) - x(a) > \boldsymbol{LP} + \boldsymbol{SP}$, o bien $x(p) - x(a) \leq 0$ y $x(a) - x(p) \leq \boldsymbol{LP} + \boldsymbol{SP}$, entonces hacer $d = \boldsymbol{LP} + \boldsymbol{SP}$, sino hacer $d = -\boldsymbol{LP} - \boldsymbol{SP}$.
- Hacer $T = T \cup \{((x(a) + d,\ l),\ (x(a) + d,\ y(a) - \boldsymbol{AP}/2 - \boldsymbol{LR}))\}$.
- Hacer $T = T \cup \{((x(a),\ y(a) - \boldsymbol{AP}/2),\ (x(a),\ y(a) - \boldsymbol{AP}/2 - \boldsymbol{LR}))\}$.
- Hacer $T = T \cup \{((x(a) - \boldsymbol{LC}/2,\ y(a) - \boldsymbol{AP}/2 - \boldsymbol{LR}/2),\ (x(a) + \boldsymbol{LC}/2,\ y(a) - \boldsymbol{AP}/2 - \boldsymbol{LR}/2))\}$.
- Hacer $T = T \cup \{((x(a),\ y(a) - \boldsymbol{AP}/2 - \boldsymbol{LR}),\ (x(a) + d,\ y(a) - \boldsymbol{AP}/2 - \boldsymbol{LR}))\}$.
- Hacer $minX = min(minX, x(a)+d)$, y $maxX = max(maxX, x(a)+d)$.
- ***continuar*** con el próximo paso anterior de la transición.

PASO 4. Hacer $T = T \cup \{((minX, l), (maxX, l))\}$.

Las observaciones hechas para la representación de transiciones disyuntivas divergentes también son válidas o tienen su contraparte para el caso de la representación de transiciones disyuntivas convergentes.

2.5.5. Transiciones simultáneas

Resulta innecesario señalar que, dentro de los cuatro tipos de transiciones a representar, las transiciones simultáneas ofrecieron el mayor grado de dificultad.

En el caso de las transiciones simultáneas, existen tres alternativas distintas a la hora de la representación gráfica. Si un a transición simultánea contiene un único paso anterior (*transición simultánea divergente*), su representación es casi idéntica a la correspondiente a una transición disyuntiva divergente. Análogamente, si el conjunto de pasos posteriores de una transición simultánea contiene un solo elemento (*transición simultánea convergente*), su representación gráfica asemeja mucho a la de una transición disyuntiva convergente. Debido a tal similitud, a continuación solamente se aborda el caso de las transiciones *Rendezvous*.

En el diseño de clases no se tomó en cuenta reflejar directamente la distinción entre estos tres tipos de transiciones; esto es, no se consideró la definición de tipos separados para representar directamente los conceptos de *transición simultánea divergente*, *transición simultánea convergente* y *transición Rendezvous*, respectivamente. Sin embargo, sí fueron concebidos tipos diferentes para transiciones simples, disyuntivas divergentes y disyuntivas convergentes por razones de eficiencia a la hora de la representación gráfica de las mismas – si bien desde el punto de vista de la semántica de un SFC no existe tal necesidad. El motivo de haber definido un único tipo dentro del diseño de clases para representar transiciones simultáneas responde a que la necesaria distinción entre los tres casos

mencionados previamente a la hora de la representación gráfica puede hacerse de manera muy eficiente, mediante una simple inspección del número de pasos anteriores y/o de pasos posteriores en la transición.

A grandes rasgos y de manera informal, la estrategia del algoritmo propuesto consiste en lo siguiente:

1. “Conectar” el conjunto de pasos anteriores de la transición de manera muy similar a como se hizo para el caso de las transiciones disyuntivas convergentes, mediante el cómputo del conjunto de segmentos asociado a cada paso anterior (PASOS 2, 3, y 4 en el algoritmo). A diferencia de las transiciones disyuntivas convergentes, en lugar de un *segmento horizontal* para “conectar” todos los conjuntos de segmentos, aparecen dos *segmentos horizontales paralelos*, muy cercano el uno del otro, separados a una distancia constante ***SS*** (PASO 5). Las rectas horizontales sobre la cuales yacen tales segmentos paralelos son previamente calculadas en función de la locación de todos los pasos que intervienen en la transición (PASO 1).
2. “Conectar” igualmente el conjunto de pasos posteriores de la transición de manera muy similar a como se hizo para el caso de las transiciones disyuntivas divergentes, mediante el cómputo del conjunto de segmentos asociado a cada paso posterior (PASOS 6, 7, y 8 en el algoritmo). A diferencia de las transiciones disyuntivas divergentes, en lugar de un *segmento horizontal* para “conectar” todos los conjuntos de segmentos, aparecen dos *segmentos horizontales paralelos*, muy cercano el uno del otro, separados a una distancia constante ***SS*** (PASO 9). Las rectas horizontales sobre la cuales yacen tales segmentos paralelos son previamente calculadas en función de la locación de todos los pasos que intervienen en la transición (PASO 1).
3. “Conectar” ambos pares de segmentos paralelos mediante un conjunto apropiado de segmentos. Los pares de segmentos paralelos son computados de manera que el par correspondiente a los pasos anteriores se encuentra “por encima” del par correspondiente al conjunto de pasos posteriores, a la distancia constante ***LR***.

   Si ambos pares de segmentos no *colapsan verticalmente*, entonces son necesarios cuatro segmentos, incluyendo aquel que representa la condición en la transición (PASO 10). En caso de que los pares *colapsen verticalmente*, solamente se necesita un par de segmentos, convenientemente colocados (PASOS 11, 12, 13, 14, 15 y 16 en el algoritmo).

A continuación es presentado formalmente el algoritmo diseñado para la representación de *transiciones simultáneas Rendezvous*:

Algoritmo para la representación de transiciones simultáneas

Entrada: Una transición simultánea $t = (A, c, P)$, en la cual tanto A como P contienen al menos dos elementos.

Salida: Un conjunto T de segmentos en el plano que constituye la representación gráfica correspondiente a t.

Método:

PASO 1. Hacer $T = \Phi$, $mY = \frac{\sum_{p \in A \cup P} y(p)}{\#A + \#P}$, $l_a = mY + \mathbf{LR}/2 + \mathbf{SS}$, $l_p = mY - \mathbf{LR}/2 - \mathbf{SS}$.

PASO 2. Hacer $mX_a = \frac{\sum_{p \in A} x(p)}{\#A}$.

PASO 3. Hacer $minX_a = mX_a$, $maxX_a = mX_a$.

PASO 4. Para cada paso anterior a de t, hacer lo siguiente:

- Si $y(a) > l_a + \mathbf{AP}/2$, entonces:
  - Hacer $T = T \cup \{((x(a), l_a), (x(a), y(a) - \mathbf{AP}/2))\}$.
  - Hacer $minX_a = min(minX_a, x(a))$, y $maxX_a = max(maxX_a, x(a))$.
  - ***continuar*** con el próximo paso anterior de la transición.
- Si $mX_a - x(a) > \mathbf{LP}/2 + \mathbf{SP}$, o bien $mX_a - x(a) \leq 0$ y $x(a) - mX_a \leq \mathbf{LP}/2 + \mathbf{SP}$, entonces hacer $d = \mathbf{LP}/2 + \mathbf{SP}$, sino hacer $d = -\mathbf{LP}/2 - \mathbf{SP}$.
- Hacer $T = T \cup \{((x(a) + d, l_a - \mathbf{SS}), (x(a) + d, y(a) - \mathbf{AP}/2 - \mathbf{LR}))\}$.
- Hacer $T = T \cup \{((x(a), y(a) - \mathbf{AP}/2), (x(a), y(a) - \mathbf{AP}/2 - \mathbf{LR}))\}$.

- Hacer $T = T \cup \{((x(a), y(a) - \mathbf{AP}/2 - \mathbf{LR}), (x(a) + d, y(a) - \mathbf{AP}/2 - \mathbf{LR}))\}$.
- Hacer $minX_a = min(minX_a, x(a)+d)$, y $maxX_a = max(maxX_a, x(a)+d)$.
- ***continuar*** con el próximo paso anterior de la transición.

PASO 5. Hacer $T = T \cup \{((minX_a, l_a), (maxX_a, l_a)), ((minX_a, l_a - \mathbf{SS}), (maxX_a, l_a - \mathbf{SS}))\}$.

PASO 6. Hacer $mX_p = \dfrac{\sum_{p \in P} x(p)}{\#P}$.

PASO 7. Hacer $minX_p = mX_p$, $maxX_p = mX_p$.

PASO 8. Para cada paso posterior p de t, hacer lo siguiente:

- Si $y(p) < l_p - \mathbf{AP}/2$, entonces:
  - Hacer $T = T \cup \{((x(p), l_p), (x(p), y(p) + \mathbf{AP}/2))\}$.
  - Hacer $minX_p = min(minX_p, x(p))$, y $maxX_p = max(maxX_p, x(p))$.
  - ***continuar*** con el próximo paso posterior de la transición.
- Si $mX_p - x(p) > \mathbf{LP} + \mathbf{SP}$, o bien $mX_p - x(p) \leq 0$ y $x(p) - mX_p \leq \mathbf{LP} + \mathbf{SP}$, entonces hacer $d = \mathbf{LP}/2 + \mathbf{SP}$, sino hacer $d = -\mathbf{LP}/2 - \mathbf{SP}$.
- Hacer $T = T \cup \{((x(p) + d, l_p + \mathbf{SS}), (x(p) + d, y(p) + \mathbf{AP}/2 + \mathbf{LR}))\}$.
- Hacer $T = T \cup \{((x(p), y(p) + \mathbf{AP}/2), (x(p), y(p) + \mathbf{AP}/2 + \mathbf{LR}))\}$.
- Hacer $T = T \cup \{((x(p), y(p) + \mathbf{AP}/2 + \mathbf{LR}), (x(p) + d, y(p) + \mathbf{AP}/2 + \mathbf{LR}))\}$.
- Hacer $minX_p = min(minX_p, x(p)+d)$, y $maxX_p = max(maxX_p, x(p)+d)$.
- ***continuar*** con el próximo paso posterior de la transición.

PASO 9. Hacer $T = T \cup \{((minX_p, l_p), (maxX_p, l_p)), ((minX_p, l_p + \mathbf{SS}), (maxX_p, l_p + \mathbf{SS}))\}$.

PASO 10. Si $maxX_p < minX_a$, o bien $maxX_a < minX_p$, entonces:

- Hacer $T = T\ U\ \{(([minX_a + maxX_a]/2,\ l_a$ - **SS** $),\ ([minX_a + maxX_a]/2,\ mY\ ))\}$.

- Hacer $T = T\ U\ \{(([minX_p + maxX_p]/2,\ l_p +$ **SS** $),\ ([minX_p + maxX_p]/2,\ mY\ ))\}$.

- Hacer $T = T\ U\ \{(([minX_a + maxX_a]/2,\ mY\ ),\ ([minX_p + maxX_p]/2,\ mY\ ))\}$.

- Hacer $T = T\ U\ \{(([minX_a + maxX_a + minX_p + maxX_p]/4,\ mY -$ **LC** $),\ ([minX_a + maxX_a + minX_p + maxX_p]/4,\ mY +$ **LC** $))\}$.

- ***retornar***.

PASO 11. Si $minX_a \geq minX_p$ y $maxX_a \leq maxX_p$, hacer $conX = (minX_a + maxX_a)/2$ e ir al **PASO 15**.

PASO 12. Si $minX_p \geq minX_a$ y $maxX_p \leq maxX_a$, hacer $conX = (minX_p + maxX_p)/2$ e ir al **PASO 15**.

PASO 13. Si $minX_a \geq minX_p$, hacer $conX = (minX_a + maxX_p)/2$ e ir al **PASO 15**.

PASO 14. Hacer $conX = (minX_p + maxX_a)/2$.

PASO 15. Hacer $T = T\ U\ \{((conX,\ l_a$ - **SS** $),\ (conX,\ l_p +$ **SS**$))\}$.

PASO 16. Hacer $T = T\ U\ \{((conX -$ **LC**$,\ mY\ ),\ (conX +$ **LC**$,\ mY))\}$.

En la siguiente figura (figura 16) se muestra la representación obtenida para dos transiciones simultáneas diferentes, luego de la aplicación del algoritmo anterior. En la transición que aparece en la parte superior izquierda, los pasos aparecen dispuestos en forma próxima a la utilizada para la representación usual. Nótese entonces cómo la representación de la transición emula la representación usual para transiciones simultáneas Rendezvous. En la parte inferior derecha, en contraste, aparece la representación gráfica obtenida para una transición en la cual la locación relativa de los pasos es menos afortunada. Sin embargo, es fácil figurar la sintaxis de la misma sin ambigüedad alguna. Por supuesto, es posible imaginar casos extremos en los que la representación de la transición y del diagrama en general serían

prácticamente ilegibles; es responsabilidad del programador de PLCs – quien dispondrá la ubicación de los pasos sobre el plano – la organización general del programa, en particular el evadir tales casos siempre que sea posible, necesario o de su interés, así como el lidiar con el solapamiento de los objetos representados.

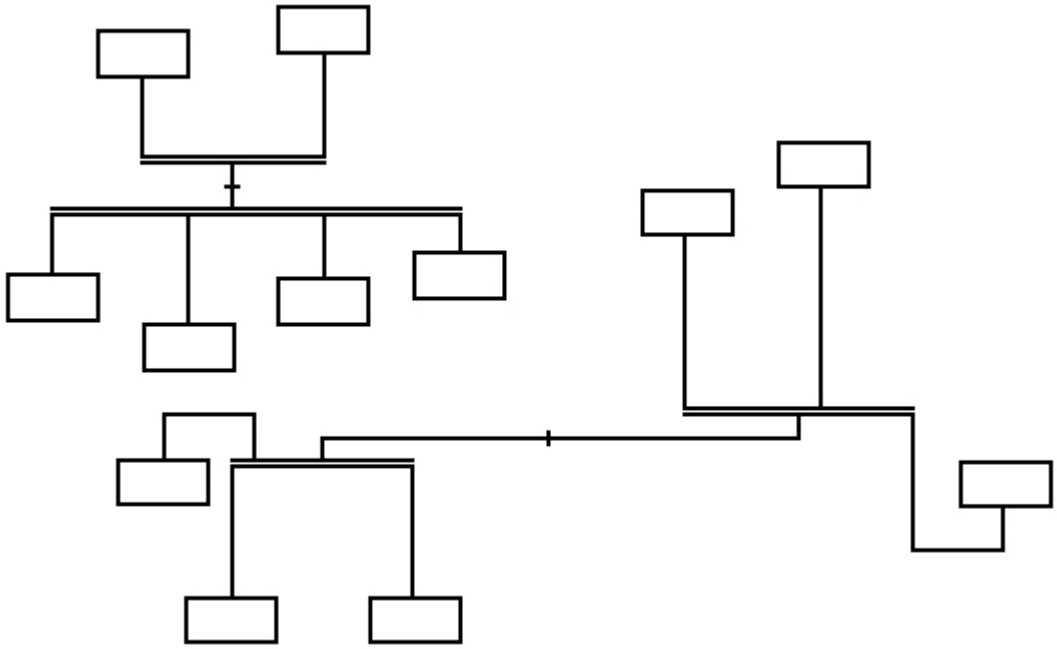

Figura 16. Representación obtenida para diferentes transiciones simultáneas.

2.6. Desarrollo de un prototipo funcional

Como parte del trabajo concebido, se dieron los primeros pasos en el desarrollo de un prototipo funcional de editor gráfico de programas SFC, haciendo uso de las ideas, herramientas de desarrollo, algoritmos y técnicas de programación expuestas hasta el momento. Por motivo del tiempo disponible no hubo posibilidad de dar culminación al mismo; sin embargo, sí se dieron algunos pasos significativos a los cuales se hace referencia en este epígrafe.

La implementación un prototipo con las características deseadas requiere el acometido de las siguientes tareas fundamentales:

1. Implementación y puesta a punto de las clases propuestas.
2. Implementación de las operaciones de navegación por el diagrama (zoom y desplazamiento) a partir de primitivas brindadas por OpenGL.
3. Extensión de la plataforma de aplicación MDI ofrecida por la MFC en correspondencia con los propósitos específicos de la aplicación.
4. Diseño e implementación de la interfaz gráfica de usuario.
5. Implementación de la selección de los objetos representados en el diagrama.

De las mismas, se dio cumplimiento cabal a las tareas 1 y 2. Se trabajó parcialmente en la realización de las tareas 3 y 4, mientras que no fue posible trabajar en la tarea 5.

La implementación y puesta a punto de las clases propuestas en el epígrafe 2.3 incluyó el desarrollo de un conjunto de clases auxiliares utilizadas como parte de la implementación. Además, llevó implícita la implementación de los algoritmos propuestos en el epígrafe 2.4 y de otros algoritmos de menor complejidad que no fueron descritos.

Por su parte, la implementación de las operaciones de navegación sobre el diagrama requirió hacer artificiosas manipulaciones con la matriz de modelación (*model matrix*), la matriz de proyección (*projection matrix*) y el *viewport* (ver [DavNei] y [San96]) para lograr el *zoom* a través del *scroll* del ratón, y el desplazamiento al estilo de la herramienta “mano”, muy común en los productos de Adobe.

Las tareas 3 y 4 guardan una estrecha relación entre sí. La extensión de la plataforma MDI en correspondencia con los propósitos específicos de la aplicación incluye la necesidad de sobrescribir convenientemente un gran número de métodos virtuales de algunas de las clases de la MFC. Como parte de la implementación de la interfaz gráfica de usuario es preciso implementar los algoritmos geométricos para la representación gráfica de los elementos de SFC descritos en el epígrafe 2.5. De estos, hasta el momento sólo ha sido implementada la representación de transiciones simples.

Por último, se hace preciso trabajar en la concepción de una estrategia para la selección de los objetos (elementos de SFC o partes de estos) representados en el diagrama. La misma dependerá en gran medida de la estrategia de representación gráfica presentada.

Conclusiones

El presente trabajo concluye con el cumplimiento cabal de los objetivos propuestos. En particular, fueron alcanzados los siguientes resultados:

1. Se hizo un estudio crítico y valorativo acerca de los problemas fundamentales asociados al desarrollo de una herramienta para la edición gráfica de diagramas SFC.
2. Se propusieron herramientas de desarrollo adecuadas para la construcción de una herramienta con tal fin.
3. Se diseñó un conjunto de conveniente de clases para representar los conceptos en el dominio de la aplicación. Las mismas fueron implementadas totalmente y puestas a punto.
4. Fueron diseñados algunos algoritmos necesarios para la implementación de las operaciones sobre un diagrama SFC.
5. Fue concebida exitosamente toda una estrategia para la representación gráfica de programas SFC. En particular, se diseñaron algoritmos formales para la representación geométrica de los elementos de SFC.
6. Haciendo uso de las herramientas de desarrollo propuestas, el conjunto de clases implementado y los algoritmos diseñados, se inició el desarrollo de un prototipo de editor gráfico de programas SFC sobre la plataforma Windows.

Estos aportes establecen las bases para el desarrollo posterior de una herramienta para la edición gráfica de programas en lenguaje SFC.

Recomendaciones

A partir de los resultados obtenidos en esta tesis, se estima conveniente hacer las siguientes recomendaciones para el trabajo futuro:

1. Concluir el desarrollo del prototipo funcional.
2. Desarrollar una herramienta para la edición gráfica de programas SFC, utilizando la metodología de desarrollo RUP.
3. Utilizar los aportes de esta tesis en el desarrollo de tal herramienta.
4. Incluir en la herramienta la generación del código en lenguaje ST correspondiente a un diagrama SFC, para su posterior procesamiento por el compilador con que se cuenta hasta el momento [Tru06].
5. Trabajar en el diseño e implementación como un todo de un entorno para la programación de PLCs, con soporte para la composición de programas utilizando los cuatro lenguajes definidos por la norma IEC-1131, más elementos de SFC.

Bibliografía

[Bec04] Beckhoff Industrial Electronics: “TwinCAT Information System”, 2004. ftp://ftp.beckhoff.com/Software/TwinCAT/InfoSystem/1033/install/TcInfoSys.exe

[DavNei] Davis, Tom; Neider, Jackie; Woo, Mason: “OpenGL Programming Guide”, Addison-Wesley Publishing Company.

[Haa04] Haase, Friedrich: “OATs 2.0 Online Manual”, Non-commercial version, 2004.

[Huu03] Huuck, Ralf: “Software Verification for Programmable Logic Controllers”, Universität zu Kiel, 2003.

[IEC93] IEC: “International Standard IEC 1131-3”, First Edition 1993-03.

[IEC96] IEC SC65B/WG7/Task Force 3: “International Electrotechnical Commission. Technical Committee No. 65: Industrial-Process Measurement and Control. Sub-Committee 65B: Devices”, 1996-09.

[Jac04] Jack, Hugh: “Automating Manufacturing System with PLCs”, 2004. http://claymore.engineer.gvsu.edu/~jackh/books/plcs/chapters/

[JohTie] John, Karl-Heinz; Tiegelkamp, Michael: “IEC 61131-3: Programming Industrial Automation Systems”, Springer-Verlag (ISBN 3-540-67752-6), s/a.

[Lew98] Lewis, R. W.: “Programming Industrial Control Systems using IEC 1131-3”, Revised Edition, The Institution of Electrical Engineers (ISBN 0 85296 950 3), 1998.

[Mal99] Maler, Oded. “On the Programmin of Industrial Computers”. 1999. http://plcopen.org/pc2/oded_malerl.htm

[Mer07] Mercury Computer Systems: “Version 6 of Open Inventor™, 2007.

[MSD05] Microsoft® Company: “Microsoft Developer Network for Visual Studio 2005”, 2005.

[San96] San Lee,Hock:“OpenGL SuperBible”, 1996.

[Sco96] Scowen, Roger: “ISO/IEC 14977 : 1996(E)” (final draft), 1996. http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf

[Str97] Stroustrup, Bjarne: “The C++ Programming Languaje”, Third Edition, 1997.

[Tru06] Trujillo, Rolando; de Rojas, Antonio: “Compilador para el Lenguaje ST”, Universidad de La Habana, 2006-06.