UNIVERSIDAD AUTONOMA DE NUEVO LEON
FACULTAD DE INGENIERIA MECANICA Y ELECTRICA
DIVISION DE ESTUDIOS DE POSTGRADO

COORDINACION DE ACTIVIDADES EN SISTEMAS DE MANUFACTURA FLEXIBLES

POR

ING. CARLOS ALONSO CAMACHO RAMIREZ

T E S I S

EN OPCION AL GRADO DE MAESTRO EN
CIENCIAS DE LA ADMINISTRACION
CON ESPECIALIDAD EN SISTEMAS

SAN NICOLAS DE LOS GARZA, N. L. ABRIL DE 1999

COORDINACION DE ACTIVIDADES EN SISTEMAS
DE MANUFACTURAS FLEXIBLES

C. A. C. R.

UNIVERSIDAD AUTONOMA DE NUEVO LEON
FACULTAD DE INGENIERIA MECANICA Y ELECTRICA
DIVISION DE ESTUDIOS DE POSTGRADO

COORDINACION DE ACTIVIDADES EN SISTEMAS DE MANUFACTURA FLEXIBLES

POR

ING. CARLOS ALONSO CAMACHO RAMIREZ

T E S I S

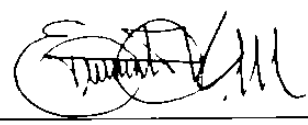
EN OPCION AL GRADO DE MAESTRO EN
CIENCIAS DE LA ADMINISTRACION
CON ESPECIALIDAD EN SISTEMAS

SAN NICOLAS DE LOS GARZA, N. L. ABRIL DE 1999

UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN
FACULTAD DE INGENIERÍA MECÁNICA Y ELÉCTRICA
DIVISIÓN DE ESTUDIOS DE POSTGRADO

Los miembros del comité de tesis recomendamos que la tesis COORDINACION DE ACTIVIDADES EN SISTEMAS DE MANUFACTURA FLEXIBLES, realizada por el Ing. Carlos Alonso Camacho Ramirez, sea aceptada para su defensa como opción al grado de Maestro en Ciencias de la Administración con especialidad en Sistemas.

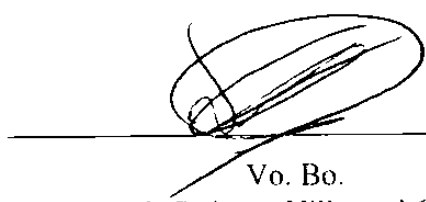
El Comité de Tesis



Asesor
Dr. Luis Ernesto López Mellado

Coasesor
Dr. José Luis Martínez Flores

Coasesor
Dra. Ada Margarita Alvarez Socarrás



Vo. Bo.
M.C. Roberto Villarreal Garza
División de Estudios de Postgrado

San Nicolás de los Garza, Nuevo León, a 15 Abril de 1999

DEDICATORIAS

A Dios . . .

Por que aún me permite seguir viviendo.

A mis Padres . . .

Hilaria Ramírez Romero y Adolfo Camacho Camacho (†), quienes siempre me mostraron el trabajo y la superación con sus ejemplos.

A mis hermanos . . .

Por su apoyo y comprensión en los momentos difíciles.

Especialmente a mi Esposa e hijos . . .

Patty, por tu cariño, comprensión y apoyo que me has brindado en todo momento.

Carlos Eduardo y Diana Lizeth, por ser mi razón de vivir y por los momentos que les he quitado para dedicárselo a este trabajo.

AGRADECIMIENTOS

Deseo expresar mi sincero agradecimiento al Dr. Luis Ernesto López Mellado, asesor de esta tesis, por su valiosa ayuda y acertada guía para el desarrollo del presente trabajo.

A los coasesores Dra. Ada Margarita Álvarez Socarrás y Dr. José Luis Martínez Flores, por la revisión y recomendaciones para poder concluir esta tesis.

Al Consejo Nacional de Ciencia y Tecnología por el apoyo económico para la realización de mis estudios.

Al Doctorado en Ingeniería de Sistemas y a la Facultad de Ingeniería Mecánica y Eléctrica de la Universidad Autónoma de Nuevo León por permitirme el uso del equipo e instalaciones.

A la Dirección General de Institutos Tecnológicos y al Consejo del Sistema Nacional de Educación Tecnológica por la licencia beca-comisión que me otorgaron para la realización del postgrado.

También deseo agradecer una manera muy especial a mi familia por la paciencia y apoyo incondicional que siempre me han brindado.

Agradezco también a todos mis compañeros del Doctorado en Ingeniería de Sistemas por su amistad y compañía brindados durante esta etapa tan valiosa para ambos.

RESUMEN

Carlos Alonso Camacho Ramírez **Fecha de Graduación: Abril, 1999**

Universidad Autónoma de Nuevo León

Facultad de Ingeniería Mecánica y Eléctrica

Título del estudio: COORDINACIÓN DE ACTIVIDADES EN SISTEMAS DE MANUFACTURA FLEXIBLES

Número de páginas: 118 **Candidato para el grado de Maestría en Ciencias de la Administración con especialidad en Sistemas**

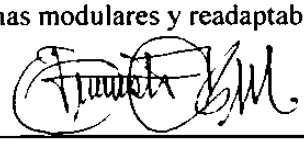
Área de estudio: Sistemas

Propósito y Método del estudio: El propósito principal de esta investigación es proponer una guía de desarrollo para coordinar las actividades desempeñadas en un Sistema de Manufactura Flexible (FMS de sus siglas en inglés). El software de supervisión y control para estos sistemas es complejo y su obtención para una tarea determinada no es evidente, dada la explosión combinatoria de las situaciones que se pueden presentar.

El control de actividades en un FMS puede ser llevada a cabo de forma secuencial o de manera concurrente. Aquí nos enfocamos a la sincronización secuencial y considerando que las actividades se ejecutan en una situación normal.

El método propuesto se descompone principalmente en dos etapas: la primera es el modelado del software de control el cual está basado en conocimiento, pues se obtienen las reglas de producción. La segunda etapa consiste de la implementación, la cual utiliza la Programación Orientada a Objetos para construir el controlador; aquí también se incluye la descomposición de la tarea general del sistema en subtareas.

Contribuciones y conclusiones: La guía para desarrollar controladores de FMS ayuda a coordinar las tareas que se ejecutan en estos sistemas, la cual fue probada implementando algunos ejemplos de células de ensamble. También se comprobó que el modelado de software de control permite construir bases de conocimiento condensadas y modulares para especificar las tareas de ensamble. Además, se confirmó que la Programación Orientada a Objetos (POO) permite acortar el ciclo para desarrollar software de control de los FMS, obteniendo programas modulares y readaptables.

FIRMA DEL ASESOR: 

Dr. Luis Ernesto López Mellado

TABLA DE CONTENIDO

Capítulo **Página**

1. INTRODUCCIÓN 1

1.1 Motivación 1

1.2 Justificación 2

1.3 Objetivos 4

1.4 Organización de la tesis 4

2. SISTEMAS DE MANUFACTURA FLEXIBLES 6

2.1 Introducción 6
2.1.1 Automatización 6
2.1.2 Contexto de los Sistemas de Manufactura Flexibles 9

2.2 Definición de Sistemas de Manufactura Flexibles (FMS) 11
2.2.1 Ejemplo de una célula de ensamble 11

2.3 La coordinación en los FMS 13
2.3.1 Jerarquía de Control 14

2.4 Resumen 15

3. SISTEMAS BASADOS EN CONOCIMIENTO 17

3.1 Inteligencia Artificial y Sistemas Expertos 17
3.1.1 Sistemas basados en conocimiento (Knowledge Based Systems) . 18
3.1.2 Componentes de un Sistema Basado en Conocimiento 19

3.2 Representación del conocimiento 21

Capítulo **Página**

3.3 Lenguajes computacionales para los SBC 26

3.4 Resumen 29

4. INTRODUCCIÓN A PROGRAMACIÓN ORIENTADA A OBJETOS 30

4.1 Introducción 30
4.1.1 Diferencias entre POO y Programación tradicional 31

4.2 Conceptos básicos de la Orientación a Objetos 34
4.2.1 Objeto 34
4.2.2 Atributos 34
4.2.3 Métodos 35
4.2.4 Clases 35
4.2.5 Abstracción y encapsulación de datos 36
4.2.6 Herencia 37
4.2.7 Ejemplares (Instances) 38
4.2.8 Mensaje 38
4.2.9 Polimorfismo 39
4.2.10 Analogías entre POO y Programación tradicional 39

4.3 Programación Orientada a Objetos con C++ 40

4.4 Resumen 41

5. GUÍA PARA EL DESARROLLO DE CONTROLADORES DE FMS 42

5.1 Introducción 42

5.2 Coordinación de actividades en FMS 44
5.2.1 Control de secuencias 45

5.3 Guía de desarrollo para coordinar actividades en un FMS 47
5.3.1 Contexto de la modelación 47
5.3.2 Guía de desarrollo para modelar FMS 53
5.3.3 Modelando una célula de ensamble 58

5.4 Resumen 61

Capítulo Página

6. IMPLEMENTACION DE EJEMPLOS 63

6.1 Introducción 63

6.2 Ejemplo 1 63

6.3 Ejemplo 2 70

6.4 Resumen 75

7. CONCLUSIONES Y RECOMENDACIONES 76

7.1 Conclusiones 76

7.2 Trabajos futuros 77

APÉNDICE A. PROGRAMACIÓN ORIENTADA A OBJETOS EN C++ 79

APÉNDICE B. CÓDIFICACION DEL EJEMPLO 1 93

APÉNDICE C. CÓDIFICACION DEL EJEMPLO 2 104

REFERENCIAS 116

LISTA DE FIGURAS

Figura **Página**

2.1 Relación de los tipos de automatización 8
2.2 Célula de ensamble 13
2.3 Jerarquía de control para sistemas de manufactura 15
3.1 Esquema de bloques de un Sistema Basado en Conocimiento..... 20
3.2 Ejemplo de red semántica 23
3.3 Ejemplo de frames 24
4.1 Relación entre POO y C++ 32
4.2 Diferencias entre programación tradicional y POO 33
4.3 Representación de una clase usando frames 35
4.4 Un ejemplo de clase 36
4.5 Analogía entre POO y programación tradicional 40
5.1 Estructura de una célula de trabajo 46
5.2 Clasificación funcional de los componentes.....54
5.3 Representación con grafo..... 56
5.4 Célula de ensamble simple 58
5.5 Clasificación de célula de la figura 5.4 59
5.6 Grafo de la figura 5.4 60
5.7 Subtarea 1 de figura 5.6 60
5.8 Subtarea 2 de figura 5.6 60
6.1 Célula de ensamble del ejemplo 1 64
6.2 Orden de ensamblaje de ejemplo 1 65
6.3 Grafo de la célula de ensamble del ejemplo 1 66
6.4 Subtareas de ejemplo 1 67-68

Figura **Página**

6.5 Orden de ensamblaje del ejemplo 2 70
6.6 Grafo de la célula de ensamble del ejemplo 2 71
6.7 Subtareas del ejemplo 2 71-72

CAPÍTULO 1

INTRODUCCIÓN

1.1 Motivación

La empresa de manufactura, considerada como un sistema, está formada por muchas y variadas funciones; pero sin duda alguna, todas estas funciones están directamente influenciadas por el proceso mismo de manufactura.

Es por esto que el proceso de manufactura representa una parte fundamental de la empresa manufacturera, y por ende, es necesario buscar y/o desarrollar nuevas y mejores tecnologías en el contexto de la manufactura, tales como:

- Métodos para evaluar los sistemas de manufactura existentes
- Mejores políticas de producción
- Control total de la calidad
- Integración de manufactura por computadora
- Métodos para modelar y simular los sistemas de manufactura
- Automatización de los sistemas de ensamble/manufactura
- Flexibilidad en el proceso de manufactura

Por lo anterior, existe la necesidad de enfocar nuestra atención a los diferentes sistemas de ensamble/manufactura que podemos encontrar en las empresas de la actualidad. Más específicamente a la manera en que se coordinan y controlan las actividades de la función de manufactura.

Además, esto implica el requerimiento de un método para modelar de una manera adecuada y eficiente los sistemas de manufactura. Dentro de los diferentes tipos de sistemas de manufactura que requieren de un método para modelarse y coordinar sus actividades de ensamble están los llamados Sistemas de Manufactura Flexible (o FMS por sus siglas en inglés de Flexible Manufacturing System).

1.2 Justificación

Un sistema de ensamble/manufactura consta de varios componentes de naturaleza y conducta diversa, como pueden ser robots, máquinas procesadoras, bandas transportadoras, etc.. Estos elementos interactuan entre sí y desempeñan varias actividades para satisfacer los requerimientos específicos de producción.

La coordinación de las actividades de los componentes que conforman un sistema de manufactura puede estar a cargo de un sistema de control computarizado, el cual integra una gran cantidad de información de diferentes fuentes. Debido a la complejidad de las tareas, el controlador es usualmente sintetizado usando diferentes técnicas.

Además, el software de supervisión y control es complejo y su obtención para la ejecución de una tarea dada no es evidente, dada la explosión combinatoria de las situaciones que se presentan.

En el trabajo desarrollado por López-Mellado [11], se propone un método para la síntesis de controladores en tiempo real de sistemas de ensamble. Este trabajo utiliza el enfoque basado en el conocimiento (knowledge-based), donde se muestra que el proceso de toma de decisiones del control de tareas de ensamble, puede ser expresado por un conjunto de reglas que generan los comandos necesarios a ejecutarse. El método propuesto consiste principalmente en el modelado de sistemas de ensamble y en la construcción de bases de conocimiento, obteniendo como resultado un software condensado, modular, legible y fácil de modificar.

El mismo autor propone en 1997 [13] un método para la síntesis de software de control de Sistemas de Manufactura Flexibles. Este trabajo, el cual es una adaptación del anterior, trata de cómo construir de una manera sistemática programas basados en objetos que son utilizados para controlar la secuenciación de operaciones que se tienen que seguir en un sistema de ensamble, considerando que todas las operaciones son ejecutadas en un funcionamiento normal. Este método consiste principalmente en:
a) la modelación de la célula de ensamble.
b) la especificación de las tareas a ejecutarse.

Por otra parte, el enfoque Orientado a Objetos (Object-based) ha servido de soporte en muchas disciplinas y se ha usado para el modelado y simulación de todo tipo de sistemas, entre ellos los sistemas de ensamble/manufactura.

La Programación Orientada a Objetos permite una definición rápida y natural de objetos, clases y métodos acortando el ciclo de desarrollo de software de control, obteniendo programas modulares y readaptables.

Este trabajo de tesis, trata de formalizar un método para coordinar y controlar las actividades de sistemas de manufactura flexibles utilizando un

enfoque basado en el conocimiento (knowledge-based) y un enfoque orientado a objetos.

1.3 Objetivos

El objetivo principal de la tesis es proponer y formalizar una guía de desarrollo que ayude a la obtención de software de control para Sistemas de Manufactura Flexibles (FMS).

El enfoque está basado en conocimiento; combina el uso de reglas de producción para modelar los procesos de toma de decisión controlando las actividades de los componentes del sistema de manufactura flexible.

El método propuesto permite la construcción de software de control modular y condensado usando lenguajes orientados a objetos.

Esta tesis presenta y propone una guía de desarrollo para modelar y auxiliar en el control de Sistemas de Manufactura Flexibles.

1.4 Organización de la tesis

El trabajo que aquí se presenta está dividido en siete capítulos, cuyo contenido se menciona a continuación.

En el segundo capítulo se da una introducción a los sistemas de manufactura, incluyendo los tipos de automatización que existen para estos sistemas. Además se presenta una definición de los Sistemas de Manufactura

Flexibles, así como el contexto de los mismos. Incluye también algunos ejemplos de este tipo de sistemas.

En el tercer capítulo se presenta la relación que existe entre la Inteligencia Artificial y los Sistemas Basados en Conocimiento, enfocándonos principalmente, a las formas más comunes de representar el conocimiento. También se incluyen algunos lenguajes computacionales que son utilizados para programar sistemas basados en conocimiento o sistemas expertos.

En el cuarto capítulo encontramos una introducción a la filosofía Orientada a Objetos, incluyendo los conceptos básicos y algunos ejemplos para visualizar la manera como pueden ser implementados en lenguaje C++.

En el quinto capítulo aparece la guía de desarrollo que se propone para modelar los Sistemas de Manufactura Flexibles, ejemplificando los pasos que se tendrían que seguir para modelar una célula de ensamble sencilla. La guía es descompuesta principalmente en dos etapas:
a) modelado del software de control, el cual está basado en conocimiento, y
b) la implementación, que utiliza la Programación Orientada a Objetos.

En la sexta parte de la tesis se presenta la manera como se implementó y probó el método propuesto. Aquí básicamente aparecen dos ejemplos de células de ensamble.

Finalmente, se presentan las conclusiones y las investigaciones futuras que se sugieren para mejorar el trabajo presentado en esta tesis.

En los apéndices aparece una introducción al lenguaje C++ y el código de dos ejemplos implementados en la versión 3.1 de Borland C++.

CAPÍTULO 2

SISTEMAS DE MANUFACTURA FLEXIBLES

2.1 Introducción

En las empresas industriales actuales, los sistemas de manufactura y la automatización son dos tecnologías que están estrechamente relacionadas. Es por ello que no se puede hacer referencia a una, sin mencionar a la otra tecnología. A continuación presentamos algunos conceptos básicos de la automatización, para posteriormente pasar al contexto de los Sistemas de Manufactura Flexibles.

2.1.1 Automatización

En un contexto industrial podemos definir la automatización como una tecnología que está relacionada con el empleo de sistemas mecánicos, eléctricos y electrónicos, la cual está basada en computadoras para la

operación y control de la producción. Ejemplos de esta tecnología son: líneas de transferencia, máquinas de montaje mecanizado, sistemas de control realimentado (aplicados a los procesos industriales), máquinas-herramienta con control numérico y robots, entre otros.

Existen tres clases amplias de automatización que se aplican a los procesos industriales [4], las cuales son:

- Automatización fija,
- Automatización programable y
- Automatización flexible.

La automatización fija es adecuada para diseñar equipos especializados para procesar el producto con alto rendimiento y con elevadas tasas de producción. Los problemas encontrados aquí son: primero, que al ser el costo de inversión inicial elevado, si el volumen de producción resulta ser más bajo que el previsto, los costos unitarios se harán también más grandes que los considerados en las previsiones. Segundo, que el equipo está especialmente diseñado para obtener un tipo de producto, por lo que, para productos con cortos ciclos de vida el empleo de esta automatización representa un gran riesgo.

La automatización programable se utiliza cuando el volumen de producción es relativamente bajo y hay una diversidad de productos a obtener. En este caso el equipo de producción está diseñado para ser adaptable a variaciones en la configuración del producto. Esta característica de adaptabilidad se realiza haciendo funcionar el equipo bajo el control de un programa de instrucciones preparado especialmente para el producto dado. El programa se introduce en el equipo de producción y este último realiza la secuencia particular de operaciones de procesamiento (o montaje) para obtener el producto. En términos de economía, el costo del equipo

programable puede repartirse entre un gran número de productos, aún cuando sean diferentes. Gracias a la característica de programación y a la adaptabilidad resultante del equipo, muchos productos diferentes y únicos en su género pueden obtenerse económicamente en pequeños lotes.

La automatización flexible, es una categoría existente entre automatización fija y automatización programable y también es conocida como "Sistemas de manufactura flexibles" (o FMS del inglés Flexible Manufacturing Systems). La experiencia adquirida hasta ahora con este tipo de automatización indica que es más adecuado para el rango de producción de volumen medio[4], como se ilustra en la figura 2.1.

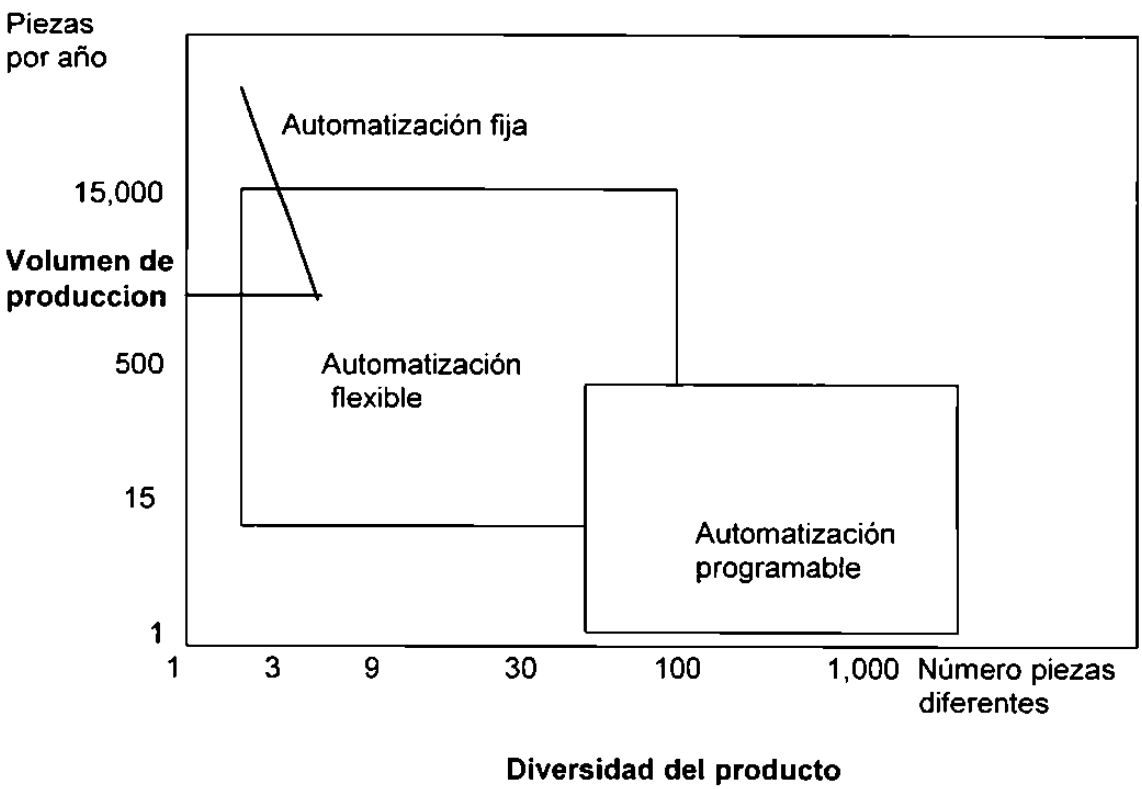


Figura 2.1. Relación de los tipos de automatización

Los sistemas flexibles tienen algunas de las características de la automatización fija y de la automatización programable. Debe programarse para diferentes configuraciones de productos, pero la diversidad de las configuraciones suele estar más limitada que para la automatización programable. Los sistemas automatizados flexibles suelen estar constituidos por una serie de estaciones de trabajo que están interconectadas por un sistema de almacenamiento y manipulación de materiales. Una computadora central se utiliza para controlar las diversas actividades que se producen en el sistema, encaminando las diversas piezas a las estaciones adecuadas y controlando las operaciones programadas en las diferentes estaciones.

Una de las características que distingue a la automatización flexible de la automatización programable es que con la automatización flexible diferentes productos pueden obtenerse al mismo tiempo en el mismo sistema de manufactura. Con la automatización programable los productos se obtienen en lotes; cuando se completa un lote, el equipo se programa para procesar el siguiente lote.

Lo anterior significa que la automatización flexible tiene un nivel de versatilidad que no está disponible en la automatización programable pura. Esto significa que pueden obtenerse productos en un sistema flexible en lotes si ello fuera deseable, o varios estilos de productos diferentes pueden mezclarse en el sistema. La potencia de cálculo de la computadora de control es lo que posibilita esta versatilidad.

2.1.2 Contexto de los Sistemas de Manufactura Flexibles

El objetivo general de un sistema de manufactura cualquiera, es el de convertir materia prima en producto terminado que tenga valor en el mercado. La tarea de coordinar todas las actividades requeridas para lograr este objetivo

es extremadamente compleja y se convierte en un problema de procesamiento de información [28].

En el ciclo de transformación de un proceso típico de manufactura existe una característica específica de interés para este tipo de sistemas, la cual es la planeación y control de las actividades que se ejecutan en un proceso de ensamble/manufactura.

La planeación y control de los sistemas de manufactura consiste en determinar la secuencia de operaciones individuales de proceso y ensamble requeridos para producir el producto terminado. Aquí se determinan todas las operaciones y recursos de máquina y de labor asociados a cada componente del sistema.

Por todo lo anterior, es necesario que enfoquemos nuestra atención a los diferentes sistemas de ensamble/manufactura que podemos encontrar en las empresas de la actualidad, más específicamente, a la manera en que se coordinan y controlan todas las actividades del proceso de ensamble para obtener la producción requerida de las partes terminadas.

Dentro de los diferentes tipos de sistemas de manufactura que se mencionaron en el apartado anterior y, que utilizan la automatización como parte fundamental de su proceso, se encuentran los Sistemas de Manufactura Flexibles. Estos sistemas son representativos de las tendencias actuales en la automatización de sistemas de producción de escala media; en los cuales uno o más de sus componentes podrían ser programados para alcanzar los requerimientos de producción.

Los Sistemas de Manufactura Flexible son la columna vertebral que se presentará en este escrito; y nos enfocaremos a ellos de aquí en adelante, por lo que a continuación se presenta una definición común de estos sistemas.

2.2 Definición de Sistemas de Manufactura Flexibles (FMS)

Los Sistemas de Manufactura Flexibles son aquellos sistemas que consisten de la integración computarizada de varios componentes de naturaleza y conducta diversa, como pueden ser robots, bandas transportadoras, sensores, máquinas procesadoras, etc., los cuales interactúan entre sí y desempeñan varias actividades para satisfacer los requerimientos específicos de producción.

Actualmente, los Sistemas de Manufactura Flexible están diseñados de una manera modular para combinar células de ensamble y de manufactura en las cuales las piezas de trabajo son movidas entre células por medio de bandas o vehículos guiados automáticamente. Cada célula consta de máquinas-herramientas, robots programables y medios de almacenamiento [8].

Entre los ejemplos más comunes de Sistemas de Manufactura Flexibles podemos encontrar una célula de ensamble robotizada, como el ejemplo que se muestra a continuación.

2.2.1 Ejemplo de una célula de ensamble

Este ejemplo el cual fue dado por Arjona-Suarez y López-Mellado en [8], consiste en una célula de ensamble robotizada que tiene los siguientes componentes:

- un robot simple,
- una banda transportadora,
- una mesa rotatoria con cuatro sitios para el almacenaje temporal de partes,
- una mesa de ensamble con dos sitios,
- una mesa para productos terminados,

- un sistema de visión (cerca de la banda transportadora hay una cámara de TV),
- varios sensores simples (celdas infrarroja y micro-switchs).

La tarea general consiste del ensamble de partes diferentes de acuerdo a un modelo establecido con anterioridad, las cuales conformarán el producto terminado.

Las actividades que ejecuta cada uno de los componentes de la célula son las siguientes:

1. Por la banda transportadora llegan las partes a ser ensambladas en forma aleatoria. En la mesa de ensamble se lleva a cabo el ensamble de las partes.

2. La mesa de almacén rotatoria es para guardar temporalmente las partes.

3. En la mesa de producto terminado se colocan los ensambles finales.

4. El robot se encarga de ensamblar las partes y de mover los productos terminados o partes entre los componentes de la célula.

Una representación gráfica de la célula de ensamble se muestra a continuación en la figura 2.2.

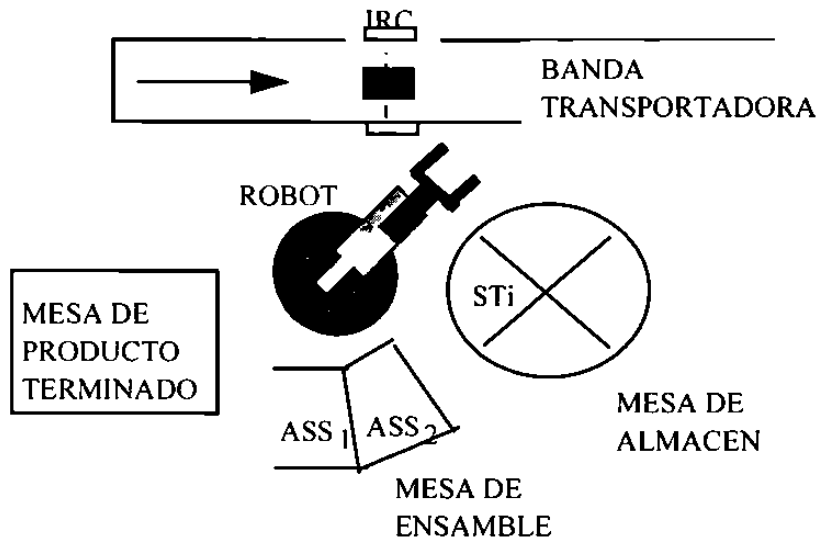


Figura 2.2. Célula de Ensamble

2.3 La coordinación en los FMS

La coordinación de las actividades de los componentes que conforman un sistema de manufactura puede estar a cargo de un sistema de control computarizado, el cual integra una gran cantidad de información de diferentes fuentes.

Debido a la complejidad de las tareas, el controlador es usualmente sintetizado usando técnicas de planeación. Además, el software de supervisión y control es complejo y su obtención para la ejecución de una tarea dada no es evidente, dada la explosión combinatoria de las situaciones que se presentan.

Varios autores coinciden con la jerarquía de control para sistemas discretos de manufactura que se muestra a continuación.

2.3.1 Jerarquía de Control

En la figura 2.3 se muestra la jerarquía de control que es utilizada para atacar la coordinación de actividades en los FMS.

En este esquema, la función general de control es descompuesta en varias capas en el cual el tiempo de respuesta es corto en niveles bajos. Las capas se definen enseguida:

- El nivel más bajo de la figura contiene los controladores locales de los dispositivos físicos de la célula (robots, máquinas, sensores, etc.).

- El nivel de "coordinación de tareas" ó nivel célula, se maneja y supervisa las actividades de los controladores locales involucrados en una célula, generando comandos pertinentes de acuerdo a eventos emitidos del nivel de control local.

- El nivel "Planeación de tareas" genera la estrategia del controlador para la célula de ensamble según las especificaciones del nivel de planeación de la producción.

- En la capa superior se genera un programa de producción y de abastecimientos basándose en los pronósticos de ventas, estándares de producción ó capacidad de la célula de ensamble.

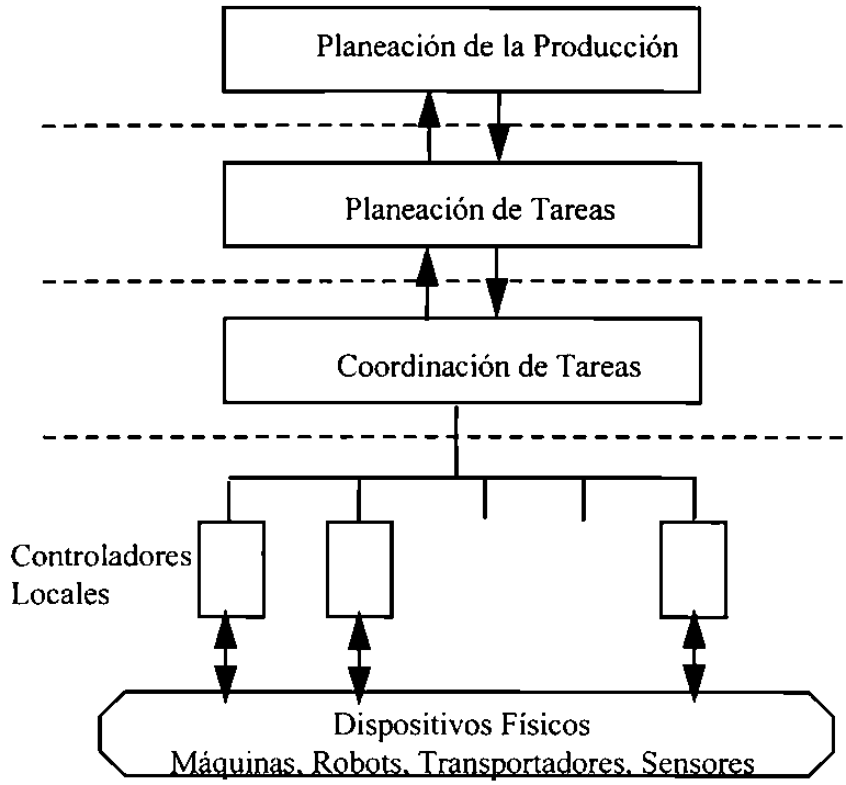


Figura 2.3. Jerarquía de control para sistemas de manufactura

2.4 Resumen

Se presentó una definición de los Sistemas de Manufactura Flexibles, así como una jerarquía de control usada por varios autores para coordinar este tipo de sistemas.

En este trabajo se trata con el nivel de Coordinación de las actividades, es decir, nos enfocamos en la secuenciación de las tareas, considerando principalmente situaciones de ejecución normal.

La programación de tareas en un FMS es un trabajo difícil que crece proporcionalmente a la complejidad de las tareas; así que las herramientas de

modelado de tareas son muy útiles para especificar las actividades a desarrollarse en un sistema y para analizar su comportamiento.

El objetivo fundamental de esta tesis es la coordinación de actividades en un Sistema de Manufactura Flexible utilizando técnicas de representación del conocimiento y la Programación Orientada a Objetos.

CAPÍTULO 3

SISTEMAS BASADOS EN CONOCIMIENTO

3.1 Inteligencia Artificial y Sistemas Expertos

La Inteligencia Artificial es la capacidad de una computadora para imitar las acciones del cerebro humano y para ello hace uso de los datos que le son proporcionados, los cuales representan el conocimiento.

Lo que más se aproxima a la reproducción de la inteligencia artificial son los llamados sistemas expertos, ya que se les considera como la primera aplicación práctica de la inteligencia artificial. Por ello, a continuación presentamos la definición y algunas características de estos sistemas.

3.1.1 Sistemas basados en conocimiento (Knowledge Based Systems)

Un Sistema Experto o Sistema Basado en el Conocimiento (SBC), es un conjunto de programas computacionales que son capaces, mediante la aplicación de conocimientos, de resolver problemas en un área determinada del conocimiento.

Características de los Sistemas Basados en el conocimiento

- Un sistema experto es una clase de Sistemas Basados en el Conocimiento.

- Un SBC tiene una limitación en cuanto al conocimiento que puede contener, tanto por el tamaño de memoria en que se almacena, como por el tiempo requerido para su procesamiento; por lo que se utiliza en un campo muy reducido.

- Las dos funciones básicas de un SBC son: la resolución en base a un conocimiento y la comunicación de este conocimiento al usuario.

- El conocimiento en un SBC tiene que estar en forma de unidades elementales del mismo, de tal manera que puedan relacionarse unas con otras y que nos permitan conocer cuál de ellas ha actuado, cuándo y por qué.

- La táctica en un SBC para la resolución de problemas se suele representar de varias formas, de las cuales las más utilizadas son: el uso de coeficientes de certeza y el empleo de metarreglas.

- Los SBC tienen que se flexibles, es decir poder modificar el conocimiento que tienen almacenado de una forma sencilla y sin que ello afecte al resto del sistema.

- La flexibilidad de un SBC lo convierte en un programa muy rentable y productivo al poderse adaptar a las necesidades, criterios, políticas, etc. de cada situación.

3.1.2 Componentes de un Sistema Basado en Conocimiento

Mientras en un programa tradicional sus componentes conforman una misma unidad, esto es,

DATOS+ALGORITMOS+CONTROL+ENTRADAS/SALIDAS=PROGRAMA;

en un SBC estos elementos son independientes unos de otros y forman unidades separadas, una forma representativa es:

(BASE DE HECHOS, BASE DE CONOCIMIENTOS, MOTOR DE INFERENCIA, ENTRADA/SALIDA) = SBC

Un esquema de bloques de un SBC se presenta en la figura 3.1, donde podemos ver los elementos que lo conforman prosiguiendo con la definición de cada uno de ellos.

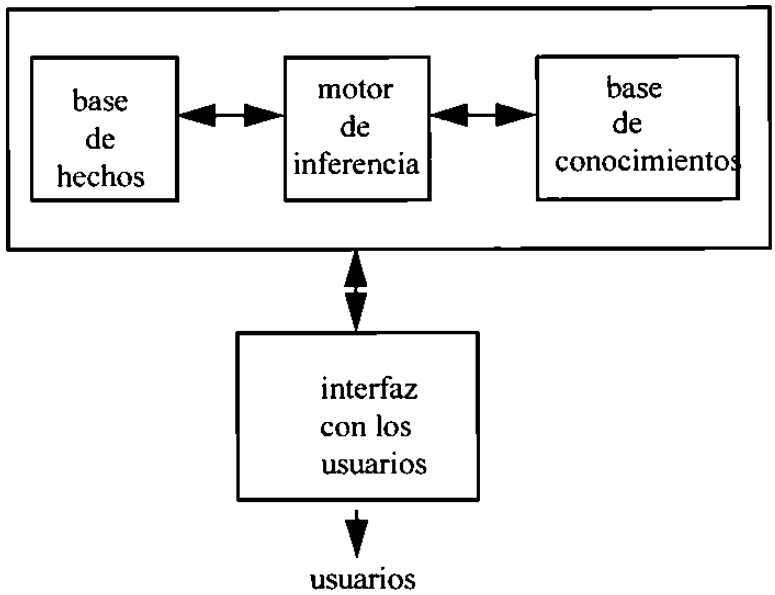


Figura 3.1. Esquema de bloques de un SBC

De la figura anterior podemos observar los siguientes elementos:

Motor de Inferencia (Inference Engine)

Es el sistema de control del Sistema Experto, que construye de una forma dinámica las soluciones. Selecciona, decide, interpreta y aplica el conocimiento de la Base de Conocimientos sobre la Base de Hechos con el fin de obtener la solución buscada.

A los motores de inferencia que emplean conocimiento representado en forma de reglas también se les llama intérprete de reglas.

Base de Conocimientos

Es la parte de un Sistema Experto que contiene el conocimiento del dominio en el cual es competente el programa.

El conocimiento en la Base de Conocimientos tiene que ser representado con el fin de que pueda incluirse en el sistema.

La Base de Conocimientos debe ser desde el punto de vista lógico completa y coherente, y desde el punto de vista funcional debe ser rápida, modular, fácil de desarrollar y mantener.

Base de Hechos

Es el conjunto de información que forman el universo del Sistema Experto y en base a ellos, y mediante la Base de Conocimientos, el Sistema llega a la solución.

3.2 Representación del conocimiento

Una de las partes fundamentales que conforman a un Sistema Experto, es sin duda la Base de Conocimientos. Por lo que de aquí en adelante enfocamos nuestro interés en este concepto, y mas especificamente, en la manera de representar el conocimiento.

Dentro de la Inteligencia Artificial existen diferentes formas que se utilizan para representar el conocimiento, por lo que a continuación presentamos algunas de las técnicas más comunes para tal modelación.

Durante mucho tiempo la psicología cognoscitiva ha investigado las formas de representar el conocimiento que más se aproximen al del cerebro humano. Basándose en estos estudios, la ingeniería del conocimiento, ha creado diversos sistemas de representación que implementados sobre una computadora actúan según los modelos teóricos elaborados por la psicología cognoscitiva [5].

Existen múltiples formas de representar el conocimiento en una computadora, y cada una de ellas representa alguna ventaja sobre las demás [5,6,7,24,25,26]. Las formas más importantes de representar el conocimiento son:

1. Redes semánticas ("semantics nets")
2. Marcos ("Frames")
3. Reglas de producción ó reglas ("Rules")

Las cuales se presentan a continuación:

1. Redes Semánticas

Las redes semánticas o redes asociativas son representaciones del conocimiento que consisten en nodos y arcos. Los nodos normalmente representan objetos, conceptos o situaciones y los arcos representan las relaciones entre nodos, estas relaciones pueden ser de herencia o de descripción. Los nodos y los arcos se etiquetan con descripciones de lenguajes simples. Como ejemplo, la siguiente información se muestra en una red semántica que aparece en la figura 3.2:

Juan es un estudiante.
Juan es un aficionado a la robótica.
Los aficionados a la robótica utilizan juegos.
Los aficionados a la robótica pueden ser estudiantes.
Los estudiantes pueden ser aficionados a la robótica.

Se pueden deducir muchas inferencias respecto de Juan, de los estudiantes y de los aficionados a la robótica investigando la red.

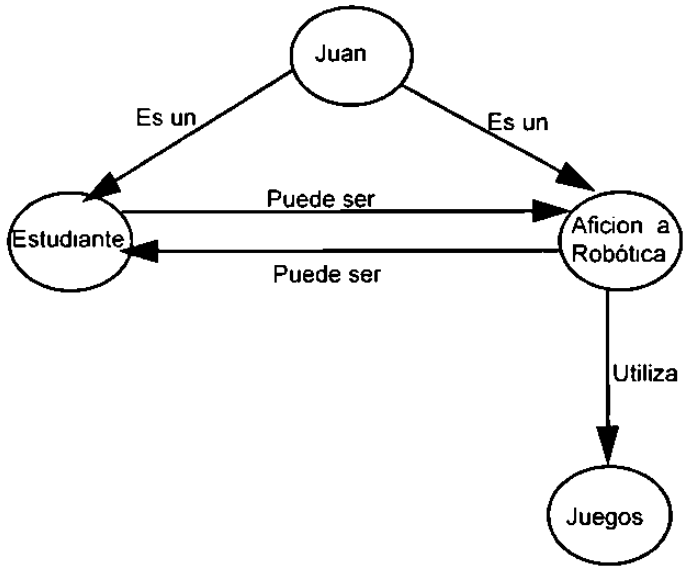


Figura 3.2 Ejemplo de red semántica

Las ventajas de las redes semánticas son su potencia a la hora de definir relaciones y su especial adaptación a sistemas interactivos.

Las desventajas de las redes semánticas son su poca flexibilidad, lo que dificulta las modificaciones.

2. Marcos

El concepto de "marcos" es la traducción más próxima del término original "frames", o también los podemos encontrar en las bibliografías como guiones (del inglés "scripts"), trama o cuadro. Aquí utilizaremos el nombre original de frame.

Los frames combinan las características de las reglas y de las redes semánticas, son modulares por naturaleza y admiten la representación del conocimiento en forma declarativa o procedimental.

Los frames están compuestos por un nombre y por una serie de slots, que también se denominan casilleros, campos de información o variables. Los slots, pueden guardar ciertas relaciones de herencia entre ellos y contener valores o procedimientos para calcular estos valores, o también reglas para inferirlos. Un frame puede presentar uno de los dos estados siguientes:

- Modelo, prototipo o marco general, que contiene la estructura del frame y los valores por defecto.

- Particularizado, instanciado o lleno, en el cual se han sustituido algunos valores por defecto, por valores particulares. Los frames particularizados deben incluir una referencia al modelo del cual procede.

A continuación se muestran 2 ejemplos de frames, entre los que existe una relación de herencia puesto que encino es "hijo" de árbol, y a su vez árbol lo es de planta.

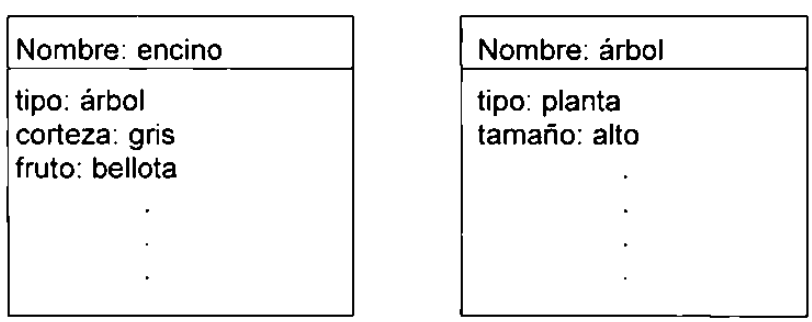


Figura 3.3 Ejemplo de frames

El uso de frames está especialmente indicado para la representación de estereotipos, en los cuales se van rellenando con información que distingue a cada caso. Los marcos pueden emplearse fácilmente para bases de datos relacionales [5,26].

3. Reglas de Producción

La representación del conocimiento en forma de reglas de producción fue tomada de la propuesta por Post en 1943 (reglas de reescritura en grámaticas). La regla es la forma más común de representar el conocimiento, debido a su gran sencillez y a que es la formulación más inmediata del principio de causalidad (causa-efecto) [5,24,25].

La regla está formada por una o más causas (o condiciones) que al cumplirse provocan la ejecución de uno o más efectos (acciones).

Los sistemas de producción son técnicas que almacenan la información en la forma de elementos llamados "producciones", las cuales tienen la siguiente forma general:

SI (condiciones) **ENTONCES** (conclusiones o acciones)

donde conclusiones puede referirse a la creación de un nuevo hecho válido, mientras que acciones se refiere a transformación de un hecho. A continuación se muestra un ejemplo:

A. **SI** la persona es estudiante de computación **ENTONCES** usa Internet.

B. **SI** la persona está leyendo esta tesis **ENTONCES** es estudiante de computación.

Si encontráramos a Eduardo leyendo esta tesis, entonces sabríamos que él es, de hecho, un usuario de Internet.

Los sistemas de producción se utilizan para dividir el problema en tareas sucesivamente más manejables. Son típicos de la construcción de Sistemas Expertos. Proporcionan sistemas diseñados uniformemente utilizando la

construcción SI - ENTONCES, como se mostró en el ejemplo anterior. Además permiten modularidad, de forma que cada regla no tiene efecto directo sobre otras reglas.

Existen algunas restricciones de sintaxis de las reglas de producción, y entre ellas están las denominadas cláusulas de Horn, que consisten en que:

1. Sólo existe un conclusión por regla.
2. La conclusión no puede aparecer negada.

Además, existen un conjunto de reglas que normalmente se les denomina estratégicas, que consisten en la ejecución de algoritmos y procedimientos para el control del proceso.

Las ventajas que presentan las reglas de producción son su carácter declarativo, su sencillez, su uniformidad que permite la representación de conocimiento como de metaconocimiento, su independencia y su modularidad al ser fácilmente agrupables. Además que, en los lenguajes de programación tradicionales las reglas se pueden escribir de forma directa.

Desafortunadamente una de las desventajas que presentan las reglas de producción es que cuando crece el número de reglas se vuelve mas lento el proceso, provocando que sean ineficaces.

3.3 Lenguajes computacionales para los SBC

A continuación se presenta una clasificación de los lenguajes computacionales que se pueden utilizar para programar Sistemas Expertos.

Según la información dada por Sáanchez y Beltrán [5], estos lenguajes se pueden agrupar en las siguientes clases:

- Lenguajes imperativos
- Lenguajes funcionales
- Lenguajes orientados al objeto
- Lenguajes declarativos

Lenguajes imperativos

Los lenguajes imperativos, son aquellos en los que el control del programa pasa siempre a la siguiente línea del programa salvo que se le ordene lo contrario.

El programador debe marcar en el programa, el flujo de la ejecución de las instrucciones.

Los lenguajes imperativos o procedimentales que se utilizan con mayor frecuencia son:

- BASIC, por su sencillez,
- PASCAL, por su estructura,
- C por su portabilidad y alto rendimiento.

Lenguajes funcionales

Los lenguajes funcionales, también llamados aplicativos, son aquellos en que el flujo del programa viene marcado por las necesidades que aparecen al evaluar una función.

El programador sólo tiene que indicar el orden de evaluación de las funciones, pero no es necesario que se preocupe de referir su posición física dentro del programa.

Esta filosofía ya existe en los lenguajes de tipo imperativo, en los que aparecen instrucciones que permiten estructuras de control funcionales, como pueden ser subrutinas o procedimientos.

Los lenguajes funcionales más conocidos son LISP y LOGO.

Lenguajes orientados al objeto

Los lenguajes orientados a los objetos se caracterizan porque no existe distinción entre los procedimientos y los datos.

Los programas están formados por los objetos que son a la vez los datos y los procedimientos (también llamados métodos).

La estructuración del programa en objetos es la base fundamental de su gran modularidad.

Las acciones que realiza el programa son realmente mensajes que se envían los objetos. Cada objeto interpreta el mensaje que le llega.

Un objeto es la particularización de una clase heredando en este proceso las propiedades de la clase que lo ha originado. Debido al proceso generativo de los objetos, éstos están estructurados jerárquicamente en clases y subclases.

Ejemplos de estos lenguajes son SMALLTALK y C++.

Lenguajes declarativos

Los lenguajes declarativos son aquellos en los que solamente hay que indicarle al programa el objetivo que queremos demostrar, especificando en el programa el universo sobre el que debe demostrarlo y las reglas que puede utilizar, es decir se especifica el "qué" pero no el "cómo".

Los lenguajes declarativos más conocidos son PROLOG y OPS

3.4 Resumen

Lo que podemos obtener de este capítulo para propósitos de la tesis son, los sistemas basados en conocimiento, más específicamente las diferentes maneras de representar el conocimiento. En nuestro caso utilizaremos las reglas de producción; las cuales servirán para tomar decisiones al momento de coordinar las actividades en un sistema de ensamble.

La guía de desarrollo que proponemos da la pauta para construir bases de conocimiento condensadas y modulares para la especificar las tareas de ensamble.

CAPÍTULO 4

INTRODUCCIÓN A PROGRAMACIÓN ORIENTADA A OBJETOS

4.1 Introducción

El enfoque Orientado a Objetos ha servido de soporte en muchas disciplinas y se ha utilizado para el modelado y simulación de todo tipo de sistemas, entre ellos los sistemas de ensamble [8,9,10]. Por ello en este capítulo se pretende dar una introducción y presentar por medio de ejemplos esta filosofía, con el objetivo de que nos sirva como base para aplicarla a la guía de desarrollo que estamos proponiendo.

Aquí se tratan los conceptos fundamentales de la Programación Orientada a Objetos y cómo se pueden implementar los mismos utilizando el lenguaje computacional llamado C++ [3,23]. Si ha manejado el lenguaje C pero no conoce el C++, se recomienda acudir al apéndice donde se muestran algunas características que el C++ le agregó al lenguaje C.

El concepto Orientado a Objetos (OO) se basa en la idea natural de que el mundo está formado por objetos, por lo que la resolución de algún problema utilizando esta filosofía, se realiza en términos de objetos.

La Programación Orientado a Objetos (POO) está teniendo una gran utilidad, y consiste en toda una metodología para el diseño de programas. Esta programación toma las mejores ideas de la programación estructurada y las combina con nuevos conceptos, ayudando así, a tener una nueva visión de la tarea de programación.

La POO permite descomponer fácilmente un problema en subgrupos de partes relacionadas, y después traducir estos subgrupos en unidades llamadas objetos.

Un lenguaje se dice que está basado en objetos, si soporta objetos como característica fundamental del lenguaje.

Antes de iniciar con los conceptos de Orientación a Objetos presentamos las diferencias con respecto a la programación estructurada.

4.1.1 Diferencias entre POO y Programación tradicional

Al hablar de la programación tradicional, nos referimos a la programación modular y estructurada. Esta se caracteriza por representar las acciones a realizar mediante algoritmos que a su vez se implementan con módulos (funciones), a las cuales se les aplican las estructuras de datos adecuadas y se obtienen los programas. Por ello se exige que el diseño del algoritmo y la estructura de datos sean los adecuados [1,2].

En la POO, la clave es encontrar características comunes entre entidades y crear estructuras de datos (objetos) que capturen estas características, permitiendo posteriormente relacionar los elementos con características comunes.

En la POO los elementos de sus programas se pueden relacionar de un modo jerárquico.

Los términos básicos utilizados en la POO con el lenguaje C++ son: Objeto, Clase y Método. En la figura 4.1 se muestra la relación de estos términos y su equivalente en C++ [3,23].

Término de POO	Equivalencia en C++
Clase	Tipo
Objeto	Variable
Método	Función

Figura 4.1. Relación entre POO y C++

Observaciones de la figura 4.1:

- Los objetos son modelos de computadora de un objeto físico.

- Las clases son los tipos de objeto.

- Los métodos son las acciones que se pueden ejecutar con un objeto, y en C++ se realizan con funciones, existiendo una por cada método.

En el esquema tradicional de programación, los datos son considerados de manera separada a las funciones, esto nos lleva a tener por un lado los datos y por otro a las operaciones, como se muestra en la figura 4.2 a.

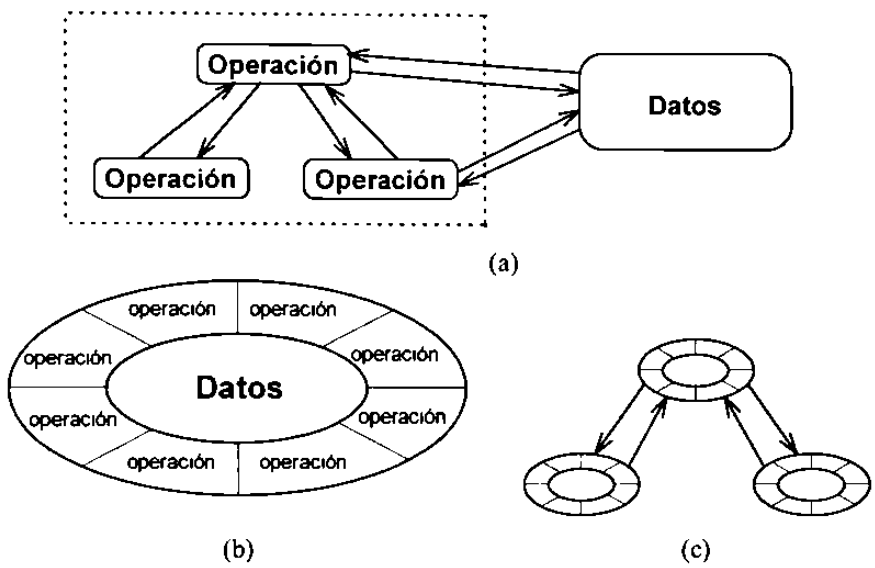


Figura 4.2. Diferencias entre la programación tradicional y la POO:
(a) Programación Tradicional. (b) y (c) Programación Orientada a Objetos

De forma distinta, la POO considera a los datos y a las operaciones que actúan de manera directa sobre esos datos, como una sola entidad a la cual se llama objeto (figura 4.2 b). Es cierto que los objetos realizan la misma función que las operaciones solas, la diferencia es que ahora las operaciones saben directamente a qué datos se refieren, sin necesidad de especificárselos por medio de parámetros.

Cabe mencionar que los datos contenidos en un objeto pueden ser a su vez objetos o algún tipo de referencia a otros objetos, comunicándose entre sí por medio de mensajes, como se muestra en la figura 4.2c.

La POO trata de hacer un mapeo natural y casi directo de la descripción y conceptualización de un problema hacia su representación en computadora.

4.2 Conceptos básicos de la Orientación a Objetos

A continuación se presenta un conjunto de definiciones que nos ayudarán a comprender un poco más la filosofía de la orientación a objetos.

4.2.1 Objeto

Un objeto es una entidad lógica que contiene datos (atributos) y un código asociado (métodos) que manipula estos datos.
Además, existen algunos puntos que se deben agregar a la definición anterior:
Un objeto se caracteriza por tener una identidad propia, un estado interno (formado por los valores de sus atributos en un momento dado) y un comportamiento.
Un objeto se refiere a una cosa que es identificable por los usuarios del sistema, ya sea tangible o abstracta.

4.2.2 Atributos

Los atributos definen las variables o tipos de datos propios del objeto, que nos proporcionan información que necesitamos conocer del mismo para resolver un problema o ejecutar un proceso. Por ejemplo, los atributos de un automóvil pueden ser: su velocidad máxima, aceleración promedio, rendimiento, etc., si lo que queremos conocer se relaciona con estas cantidades.

4.2.3 Métodos

Los métodos, entregan información relacionada con los atributos, parámetros y/o constantes; la información puede ser resultado de la aplicación de una fórmula o de algún proceso a dichos datos. Siguiendo con el ejemplo, un método podría ser el cálculo de la velocidad en un instante t, donde t sería un parámetro para dicho método.

Los métodos pueden también modificar los valores de los atributos, cambiando así el estado interno del objeto. En caso de monitorear la cantidad de combustible del automóvil por medio de un atributo, puede implantarse un método para actualizar dicha variable en un instante t.

4.2.4 Clases

Las clases son como una plantilla (o esqueleto) que contiene toda la información referente a un tipo de objeto como: métodos, atributos y las clases de quienes hereda[1]. Una forma práctica para representar clases es mostrada en la figura 4.3, este tipo de representación se conoce como frames o marcos.

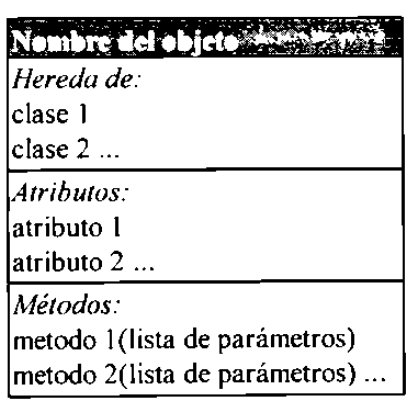


Figura 4.3 Representación de una clase usando frames

[1] El concepto de Herencia es tratado más adelante.

Este tipo de representación encierra toda la información referente al objeto y se asemeja a una declaración en algún lenguaje de programación como el C++.

La figura 4.4 presenta un ejemplo comparativo, usando frames y la sintaxis de C++.

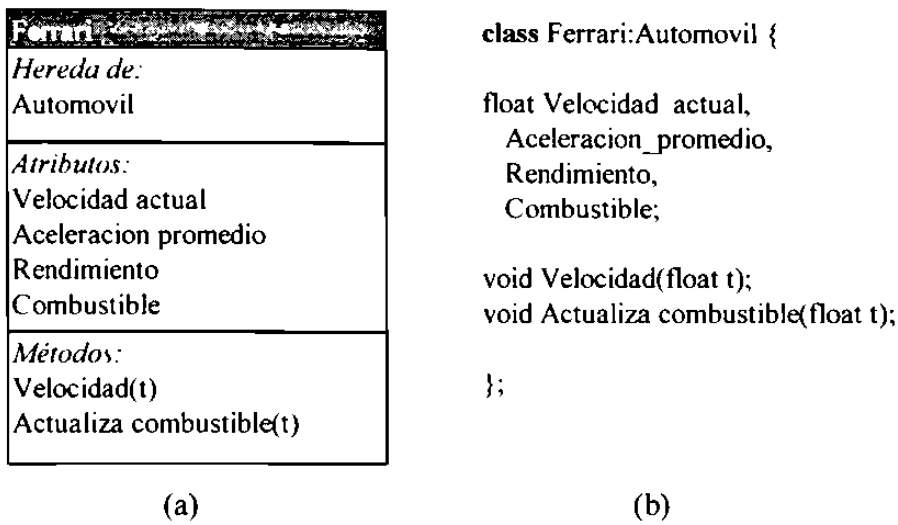


Figura 4.4 Un ejemplo de clase: (a) Frames (b) C++.

4.2.5 Abstracción y encapsulación de datos

Un tipo de datos abstracto (TDA) es un modelo de software que especifica un conjunto permisible de valores para los datos y un conjunto de operaciones permisibles que se pueden proporcionar en ellos. Por ejemplo, si el programador quiere utilizar un entero, sólo tiene que declararlo (Pascal o C), y utilizar las operaciones que ya están definidas para enteros (suma, resta, etc.), sin que tenga que conocer la forma en que el lenguaje de programación realiza el manejo interno. Es decir, esto se encuentra oculto y fuera de alcance para el programador (llamado encapsulación u ocultamiento de datos).

Mediante la encapsulación, los datos y los procedimientos que los manipulan permanecen juntos, esto limita el ámbito y visibilidad de los procedimientos y funciones a una región altamente localizable del sistema de software. Este hecho facilita la depuración y mantenimiento de programas.

En resumen, la encapsulación es una técnica usada por los lenguajes orientados a objetos que restringe el acceso a métodos y atributos, evitando que los datos de un objeto puedan ser modificados por otros objetos que no tengan permiso para hacerlo. La forma más común de representar la encapsulación es por medio de las clases.

4.2.6 Herencia

Al existir una clase ya definida, se puede definir otra clase (llamada clase derivada) a partir de la primera. Esto es, la clase derivada hereda las características de la primera clase. Lo anterior nos ayuda a reutilizar el código existente; y es que si no existiese la herencia, tendríamos que volver a definir métodos y atributos para objetos similares. En los lenguajes orientados a objetos, se definen dos tipos de herencia que son: la herencia simple y la múltiple[2].

Herencia simple

La herencia simple consiste en heredar los atributos y métodos de una sola clase a la que llamaremos base o padre. La clase resultante contendrá los atributos y métodos del padre además de los propios.

---

[2] No todos los lenguajes orientados a objetos soportan la herencia múltiple, por ejemplo Smalltalk V, solo soporta herencia simple

Herencia múltiple

La herencia múltiple es una extensión de la simple, ésta permite a una clase tener más de una clase base o padre. Así, podemos seleccionar un conjunto de clases para armar nuestra nueva clase, esto es análogo a querer armar un auto, nosotros podemos seleccionar las llantas de cierta marca, el estéreo, la tapicería, y entonces tener un auto hecho a la medida.

4.2.7 Ejemplares (Instances)

Antes de poder utilizar los objetos estos deben ser creados, ya que la clase es tan sólo una definición de las características y capacidades del objeto, no conforma al objeto en sí. Cuando creamos objetos específicos, decimos que estamos creando ejemplares o individuos de objetos. Así pues, el ejemplar de una clase es un objeto especifico de ésta. Como muestra, en la clase planeta un ejemplar de ésta sería el planeta Mercurio, otro Venus, etc., cada uno de estos ejemplares contiene datos diferentes, por lo que podemos afirmar que son individuos distintos, a pesar de provenir de la misma clase.

4.2.8 Mensaje

Cuando se requiere conocer o afectar el estado de un objeto, es preciso mandarle un mensaje[3]; este mensaje activará un método que bien puede reportar su estado o tomar decisiones para cambiarlo, si es necesario. Por ejemplo, para que el objeto automóvil cambie su velocidad podría decírsele

[3] En la práctica, algunos lenguajes orientados a objetos permiten que el estado de un objeto pueda ser modificado, accediendo a sus atributos en forma directa.

(usando la sintaxis de C++): automovil.velocidad(35.00), lo que cambiaría su velocidad a 35 Km./hora.

El concepto de mensaje, es similar a la llamada de una función en una programación tradicional, sólo que en esta última no existe un objeto asociado a dicha función. Este concepto añade naturalidad a la programación, podemos llegar a imaginar que en realidad, damos ordenes a objetos o que podemos comunicarnos con ellos, dejando a un lado lo rígido de la programación tradicional.

4.2.9 Polimorfismo

El Polimorfismo, es una de las capacidades de los lenguajes OO, que permite al programador usar el mismo nombre para definir a diferentes métodos que realizan tareas similares, pero que reciben parámetros distintos.

El fin del Polimorfismo es permitir el uso de un nombre para especificar una clase de acción general.

El polimorfismo se puede visualizar cuando sobrecargamos a las funciones ó a los operadores.

4.2.10 Analogías entre POO y Programación tradicional

Para comprender mejor los conceptos antes citados podemos usar la siguiente analogía mostrada en la figura 4.5, entre la programación orientada a objetos y la programación tradicional:

Programación Orientada a Objetos	Programación Tradicional
clase	tipo de datos
instancia	variable
instanciar	declarar variables
método	función ó procedimiento
mensaje	llamada a una función
objeto	datos + procedimientos

Figura 4.5 Analogía entre POO y Programación tradicional

4.3 Programación Orientada a Objetos con C++

Es necesario utilizar un lenguaje de programación que cumpla con los propósitos que se pretenden dar al presente trabajo. Por lo que usaremos el lenguaje C++, ya que permite la programación orientada a objetos.

Como se mencionó anteriormente, los sistemas de manufactura están conformados por varios elementos de conducta diversa. Por lo tanto, se usará la programación orientada a objetos, ya que nos ofrece facilidades para definir nuevos tipos de datos (clases), y crear tantos objetos de ese tipo como se requieran, lo cual nos permite simular a los componentes de estos sistemas.

Uno de los lenguajes computacionales Orientados a Objetos (OO) que han tenido bastante aceptación por los programadores es el C++.

En este trabajo utilizamos el lenguaje Borland C++ versión 3.1 para la simulación de Sistemas de Manufactura Flexibles, por lo que en el apéndice A titulado "Programación Orientada a Objetos en C++" se presenta la manera

como pueden ser aplicados los conceptos básicos de Orientación a Objetos utilizando el lenguaje C++. Favor de acudir a este apéndice para más detalles.

4.4 Resumen

La Programación Orientada a Objetos (POO) permite una definición rápida y natural de objetos, clases y métodos, acortando el ciclo de desarrollo de software de control y obteniendo programas modulares y readaptables, como lo mostró López/Mellado en [13].

Para el caso de la guía de desarrollo que se propone, utilizaremos el enfoque Orientado a Objetos para el modelado y simulación de los sistemas de ensamble.

CAPÍTULO 5

GUÍA PARA EL DESARROLLO DE CONTROLADORES DE FMS

5.1 Introducción

Como se ha visto en los capítulos anteriores, el sistema de control de una célula de ensamble/manufactura se ocupa de la coordinación de las actividades de todos los elementos que la componen.

Una serie de opciones de hardware están disponibles para satisfacer los requisitos de la coordinación de la célula de manufactura. Estas opciones pueden ser entre otras, controladores lógicos programables (PLC: Programmable Logic Controllers) o computadoras digitales. La decisión de cual opción usar depende de la complejidad y tamaño de la célula.

En casos donde una computadora es el controlador de la célula de trabajo, se debería utilizar o bien en conjunto con un controlador lógico programable, o como un sustituto de éste [4].

Los controladores programables son dispositivos que tienen ciertas limitaciones en cuanto al procesamiento de datos y a lenguajes de programación, que dan a la computadora una ventaja en aplicaciones que necesitan estas capacidades. Algunos ejemplos de tipos de aplicaciones que podrían favorecer al uso de computadoras para el control de célula de trabajo incluirían las siguientes:

- Casos en los que existen varias células, cuyas operaciones se deben coordinar y además cantidades grandes de datos se deben comunicar entre ellas.

- Células en las cuales el problema de detección y recuperación de errores constituye una parte importante de la codificación que se debe programar para la operación de la célula de trabajo.

- Cuando algunos productos diferentes se hacen sobre la misma línea de producción automatizada, las operaciones en las diferentes estaciones se tienen que coordinar y secuenciar adecuadamente.

- En casos donde las líneas de producción se utilizan para operaciones de ensamblaje, los diversos tamaños y estilos de piezas componentes se deben clasificar y adaptar al modelo particular que está siendo ensamblado en cada estación de trabajo a lo largo de la línea.

Las diferencias entre los computadoras digitales y los controladores programables son principalmente en aplicación más que en tecnología básica.

Algunas de las aplicaciones tienen requisitos para los cuales una computadora digital es el método más apropiado de control de la célula de trabajo, tales como los sistemas de ensamble flexibles, a los cuales hemos estado haciendo referencia a lo largo de la tesis.

En el apartado que se presenta a continuación exponemos la manera en cómo se puede llevar a cabo la coordinación de actividades en Sistemas de Manufactura.

5.2 Coordinación de actividades en FMS

Para poder coordinar las actividades en los sistemas de ensamble/manufactura existen algunas opciones entre las que destacan:

a) Que las actividades se ejecuten de forma secuencial, esto es una después de la otra.
b) Que las actividades se lleven a cabo de manera concurrente, donde una o más actividades se pueden realizar al mismo tiempo.

Independientemente de la opción que se utilice, ya sea que algunas actividades en una célula de ensamble/manufactura se produzcan en forma secuencial, o bien de manera simultánea, se requiere un método para controlar y sincronizar estas diversas tareas, y ése es el propósito fundamental del coordinador de actividades de la célula de ensamble según la jerarquía mostrada en el punto 2.3.

El control de la célula de trabajo puede ser realizado, por un computador independiente o por un controlador programable. Durante la operación, el controlador comunica señales al equipo en la célula y recibe señales desde dicho equipo. Es decir, mediante la comunicación en todos los sentidos, las diferentes actividades en la célula se realizan en la secuencia adecuada [4].

5.2.1 Control de secuencias

El control de las secuencias de operaciones es la misión principal del coordinador de la célula de ensamble durante su funcionamiento regular. Este puede incluir las siguientes funciones de coordinación:

- Controlar el orden que tienen que seguir las actividades en la célula de trabajo.
- Controlar y sincronizar actividades simultáneas.
- Tomar decisiones para proseguir o detener el ciclo de trabajo, cuando ocurren fallas con los componentes de la célula.
- Toma de decisiones para parar o retardar el ciclo de trabajo, basado en sucesos que ocurran en la célula.

Estas funciones ocurren durante la operación normal de la célula de trabajo. A continuación se presenta un ejemplo, el cual tiene como finalidad demostrar la importancia de controlar la secuencia de actividades en la células de ensamble durante la operación regular.

La célula consiste de un robot, una máquina-herramienta que opera sobre un ciclo automático, y una banda transportadora para entregar las piezas a trabajar en la máquina. Las piezas acabadas se colocan en una banda transportadora de salida. En la figura 5.1 se muestra la estructura de la célula y el ciclo de trabajo consiste en la siguiente secuencia de actividades:

1. El robot toma la pieza de trabajo de la banda transportadora de entrada (la pieza es colocada en una posición conocida por el robot).
2. El robot carga la pieza en la máquina-herramienta para que sea trabajada.
3. La máquina-herramienta comienza su ciclo de trabajo automático.
4. Cuando la máquina-herramienta finaliza el ciclo de trabajo, el robot descarga la máquina y coloca la pieza terminada en la banda transportadora de salida.
5. El robot se mueve otra vez al punto de partida inicial, para recoger la siguiente pieza que entre al sistema.

En la operación de la célula de la figura 5.1, casi todas las actividades ocurren secuencialmente. La única excepción significativa es que el transportador de entrada probablemente operaría de forma continua para dejar las piezas en la célula durante el ciclo de trabajo. El objetivo del coordinador de la estación de trabajo en este ejemplo sería el asegurar que las actividades ocurren en la secuencia correcta y que cada paso en la secuencia se ha completado antes de que se inicie el siguiente. Por ejemplo, el comienzo del ciclo de trabajo ejecutado por la máquina-herramienta no debe ocurrir hasta que el robot halla dejado la pieza de trabajo en la misma.

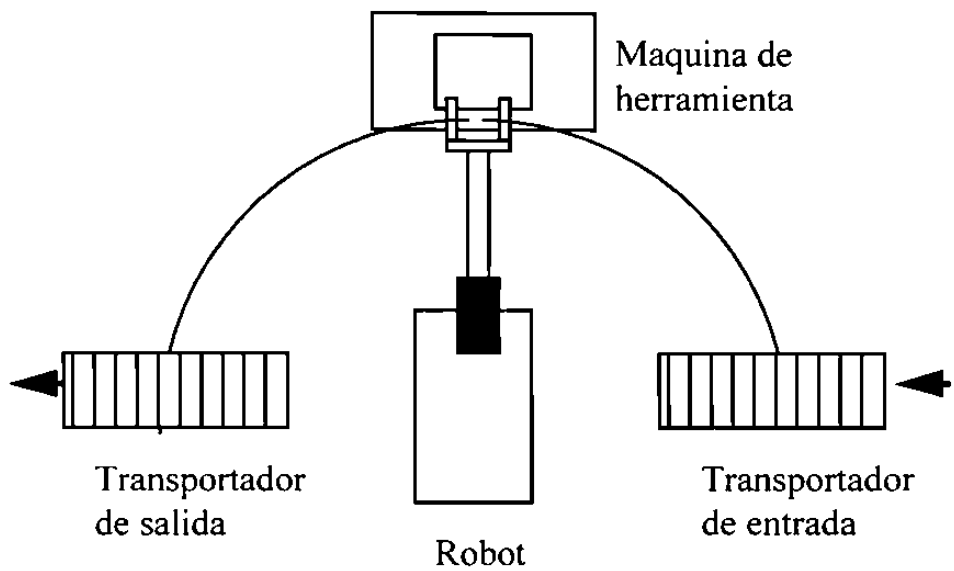


Figura 5.1 Estructura de una célula de trabajo

Como se mencionó anteriormente, las actividades también pueden ocurrir de manera simultánea.

Se pueden lanzar los comandos en forma secuencial pero las operaciones se desarrollan en forma simultánea, es decir, no es necesario esperar que una operación termine para lanzar la otra.

En cuanto a la guía de desarrollo que se está proponiendo en la presente tesis para determinar el orden de las operaciones en un FMS, se utilizará la forma secuencial, esto es, una actividad después de la otra.

A continuación se presenta la guía de desarrollo propuesta para la modelación de Sistemas de Manufactura Flexibles (FMS).

5.3 Guía de desarrollo de coordinación de actividades en FMS

En este apartado aparece el método propuesto para la coordinación de actividades en Sistemas de Manufactura Flexibles, el cual hemos dividido en tres partes: Primeramente presentamos el contexto sobre el cual vamos a trabajar. Posteriormente se da una guía de desarrollo para la modelación de un FMS, donde se muestran los pasos específicos a seguir. Por último, aplicamos este proceso a un ejemplo simple.

5.3.1 Contexto de la modelación

Primeramente, para propósitos de demostración es necesario seleccionar algún sistema que sea representativo de los FMS, en este caso hemos elegido las células de ensamble para simular tales sistemas.

En la guía de desarrollo se incluye una discretización del espacio de trabajo, una clasificación de componentes y una descripción del flujo de partes, como se mostrará más adelante con algunos ejemplos.

Algunos de los elementos clave que utilizaremos en el proceso de modelación son los siguientes conceptos:

Sitio

Un sitio se define como el lugar de algún componente del sistema, donde las piezas o partes pueden quedar en una posición estable. Por ejemplo una mesa de ensamble, una mesa de almacenaje, la mano de un robot, lugar en la banda transportadora que puede ser accesada por el robot para recoger una pieza, etc. [11,12].

Efector

Es el elemento que ejecuta alguna operación sobre las piezas o tarea específica dentro del sistema de ensamble, como ejemplos podemos tener:
El efector "robot" lleva a cabo las funciones de recoger o depositar una pieza en una mesa de almacén.
El efector "banda transportadora" introduce las piezas a la célula de ensamble.

Sensor

Obtiene información sobre el entorno y sobre las piezas: presencia, reconocimiento, inspección, localización de piezas y de herramientas, etc.

De una manera muy general, el método propuesto se descompone principalmente en dos etapas, las cuales presentamos a continuación:

A) Modelación del software de control el cual está basado en conocimiento, pues se obtienen los marcos (frames) y las reglas de producción.

B) La implementación, la cual utiliza la Programación Orientada a Objetos para construir el controlador. Aquí también se incluye la descomposición de la tarea general del sistema en subtareas.

A continuación presentamos algunas características sobresalientes dentro de cada una de estas etapas:

A) Modelación del software de control

La complejidad creciente de los sistemas de manufactura involucra recurrir a un modelo de referencia de soporte de decisión durante los diversos pasos del ciclo de vida de sistemas: diseño, evaluación, planificación y explotación. Esos modelos de referencia no son otra cosa más que modelos de simulación [9].

El problema del modelado de tareas en un FMS ha sido atacado usando métodos analíticos y numéricos. El modelado analítico se ha realizado usando principalmente teoría clásica de colas [14,15,16]. El modelado numérico ha sido realizado usando principalmente el enfoque basado en eventos por medio de Redes de Petri [17,18].

Según el criterio presentado en documentos anteriores [9], una herramienta eficiente de modelado y simulación debe ofrecer:

- Un modelo simple construido de manera que exista una relación estrecha entre los objetos simulados y los objetos reales.
- Una transparencia en la incorporación del conocimiento físico dentro del modelo.
- Modularidad, o bien la capacidad para cambiar dinámicamente la conducta de los objetos y el modelo lógico.
- Un ambiente genérico que no sea otro modelo o lenguaje de computadora.

El enfoque que se presenta aquí está basado en conocimiento. Este combina el uso de marcos y reglas de producción para implementar el proceso de toma de decisiones, controlando las actividades de los componentes del sistema de ensamble de acuerdo a eventos asíncronos los cuales representan

situaciones esperadas o inesperadas (pero predecibles). El método que se utiliza da la pauta para construir bases de conocimiento condensadas y modulares que sirvan para especificar las tareas de ensamble.

Se muestra que el proceso de toma de decisiones involucrando un control de tiempo real de tareas de ensamble puede ser expresado por un conjunto de reglas de producción que generan los comandos apropiados de acuerdo a un contexto de ejecución dado. Este contexto involucra información acerca del estado de la tarea y eventos relevantes del entorno.

La metodología propuesta ayuda a obtener sistemáticamente los contenidos de las bases de conocimiento de un modelo del sistema de ensamble; el contexto de ejecución es estructurado en marcos y reglas de despache que pueden ser implementados en un sistema de reglas. Estas técnicas de representación del conocimiento son incluidas en la mayoría de las herramientas de desarrollo de sistemas basados en conocimiento.

Los sistemas de reglas (producciones) que se propusieron como conjuntos de reglas de reescritura en gramáticas formales son ahora una técnica de representación del conocimiento ampliamente usada en aplicaciones de Inteligencia Artificial [11]. La idea básica de este formalismo está basada en la noción de Reglas de Producción.

En un sistema de reglas de producción se controlan las actividades del sistema según una estrategia ya definida. Este opera en tres pasos:

a) Evaluación de la parte antecedente de las reglas con respecto a la Base de Hechos (mostrada anteriormente en figura 3.1) actual. Las reglas cuya parte antecedente es igual es seleccionada poniéndole una "marca".
b) Se selecciona una de las reglas, de entre las reglas marcadas según el criterio dado.

c) Ejecución de las acciones expresadas en la parte consecuente de la regla seleccionada.

Estos tres pasos de cada ciclo son llamados matching (acoplamiento), resolución de conflicto, y acción [11].

Entre las cualidades de los sistemas de reglas pueden ser mencionadas las siguientes:

a) La modularidad: las producciones en la Base de Reglas pueden ser agregadas, borradas ó modificadas independientemente.
b) La uniformidad de la Base de Reglas simplifica la revisión o comprensión del conocimiento expresado.

Las principales desventajas de los sistemas de reglas, señaladas por los usuarios, son:

a) la dificultad para expresar conocimiento algorítmico, y
b) la ineficiencia del motor de inferencia (intérprete) cuando la Base de Reglas es larga.

B) Implementación Orientada a Objetos

El propósito de esta parte es la especificación de las tareas a ejecutarse, donde se describe la manera en cómo determinar las actividades de ensamble a ejecutarse. Primeramente se presentan las razones por las cuales se utiliza el enfoque OO.

El enfoque OO [1,2] ha servido de soporte en muchas disciplinas y se ha usado para el modelado y simulación de todo tipo de sistemas, entre ellos los sistemas de ensamble [8, 9, 10].

La Programación Orientada a Objetos permite una definición rápida y natural de objetos, clases y métodos acortando el ciclo de desarrollo de software de control, obteniendo programas modulares y readaptables.

Los FMS son sistemas de eventos discretos y es posible seleccionar entre varios lenguajes computacionales de automatización especializados para implementar los modelos. Cada uno de estos lenguajes está orientado a un propósito particular y tiene su propio grado de complejidad [13]. En nuestro caso, hemos optado por utilizar el lenguaje computacional llamado Borland C++, versión 3.1.

Un FMS es visto como un sistema dinámico de eventos discretos en el cual los cambios de estado ocurren solo en momentos relevantes de tiempo. Así, si observamos el estado del sistema solo en esos momentos podemos tener una descripción completa de la conducta del sistema [8].

Para especificar un sistema de eventos discretos es necesario definir: las entidades (objetos, recursos) que lo constituyen (que usualmente tiene asociado uno o más atributos), las relaciones entre entidades y los cambios que podrían ocurrir en los valores de los atributos de las entidades así como las condiciones que generan esos cambios.

En la filosofía OO, el concepto de **sitio** que se definió en el punto 5.3.1, es representado por instancias de objetos y clases [1,3,19,20,21]; y las operaciones se simulan por métodos que se incluyen en los objetos [19,20,23] que están asociadas a los **efectores**.

La sincronización de operaciones de ensamble que utilizaremos es llevada a cabo para propósitos de simulación, secuencialmente, pero en caso de una aplicación de control en tiempo real, los comandos deben ser ejecutados concurrentemente. Por lo que, una manera de implementar el control de las

actividades es definiendo conjuntos de reglas, de las cuales sólo una regla puede ser ejecutada a la vez; esto se puede llevar a cabo descomponiendo las tareas de ensamble en subtareas.

Cada manejador de subtarea puede tener asociado recursos tales como sitios, efectores, sensores, y otros. Por lo tanto, la disponibilidad de cada recurso debe ser verificada, esto por algún método que prueba si un recurso compartido está o no disponible para algún manejador.

La manera que utilizamos para representar el modelo es por medio de un grafo dirigido. En el mismo, los sitios se asocian a los vértices, los cuales se representan como óvalos y están unidos por arcos dirigidos indicando el flujo de las actividades.

5.3.2 Guía de desarrollo para modelar FMS

Enseguida se muestran los pasos que se llevan a cabo en la guía propuesta:

1. Identificar los componentes del sistema

Consiste en localizar la ubicación de cada uno de los componentes del sistema de ensamble y clasificarlos desde el punto de vista funcional. Esto es, agruparlos de tal manera que en cada grupo estén todos los componentes que tienen funciones similares. Por ejemplo en un grupo quedarían todos los efectores, en otro los sitios, y en un último se agrupan todas las operaciones que pueden ejecutar los efectores (ver figura 5.2)

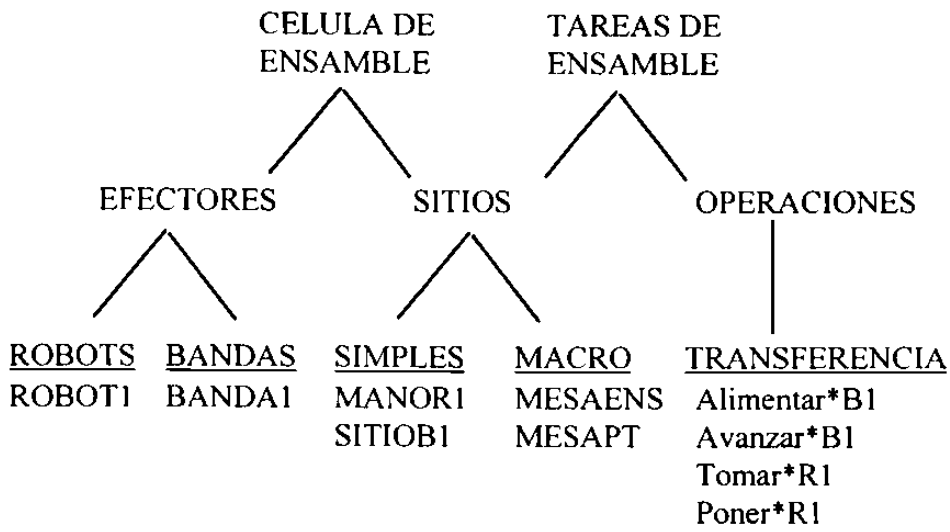


Figura 5.2 Clasificación funcional de los componentes

2. *Definir el orden de los ensambles*

- Aquí es necesario identificar cuáles son las piezas que pueden entrar al sistema para ser ensambladas y, qué se debe hacer con las piezas que no pueden ser procesadas (llamadas piezas no válidas).

- Definir el orden en que las piezas que entran al sistema deberán ser ensambladas. Debido a que pueden existir varias mesas de ensamble, es necesario definir a qué mesa le corresponde cada una de las piezas que llegan a la célula.

- Identificar a qué efectores del sistema les corresponde ejecutar cada una de las operaciones, de acuerdo al orden de ensamble predefinido con anterioridad.

3. *Especificación de la tarea y el flujo del material*

La especificación de una tarea consiste en construir un grafo dirigido (también llamado digrafo) que represente el flujo de las piezas dentro del sistema (según lo especificado en el punto anterior), así como las actividades que se llevan a cabo en el proceso. El digrafo debe ser construido tomando en cuenta las siguientes consideraciones:

- Los sitios existentes en la célula de ensamble serán representados por los nodos del grafo. Con un círculo se representan los sitios simples mientras que los macrositios se representan con un cuadrado.

- Las operaciones ejecutadas por los efectores, necesarias para transferir las piezas desde un sitio a otro o bien las operaciones para modificar las propiedades de la parte que está en el sitio, deberán ser representadas por los arcos que unen a los nodos. Se debe etiquetar cada arco con la operación que se lleva a cabo y quién la ejecuta. Algunos ejemplos de operaciones pueden ser las siguientes:
  - ✓ Introducir las piezas para que sean ensambladas (por algún efector).
  - ✓ Identificación de una pieza o inspección.
  - ✓ Modificar las propiedades de la pieza que se encuentra en un sitio (que un robot ensamble una pieza que tiene en su pinza sobre otra que está en la mesa de ensamble).
  - ✓ Transferir piezas entre sitios según el flujo ya predefinido, (que un robot tome la pieza de la banda transportadora y la coloque en la mesa de almacén).
  - ✓ Sacar las piezas ya ensambladas (producto terminado) del sistema (por algún efector).

Un ejemplo sencillo de representación por medio de grafo se muestra en la figura 5.3. Aquí se describen las operaciones de "tomar" y "poner", que

son ejecutadas por un robot. Aquí podemos visualizar tres sitios que están involucrados: CONV2, que representa el lugar en la banda transportadora donde el robot puede accesar para tomar la pieza; GRIP1, que modela el sitio asociado a la pinza del robot; y TAB1, que es el sitio de la mesa donde el robot pone la pieza.

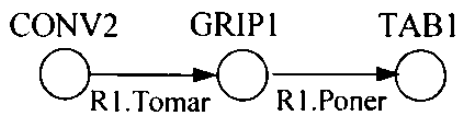


Figura 5.3 Representación con grafo

4. *Representación basada en objetos*

La secuenciación de operaciones se lleva a cabo ejecutando un conjunto de reglas de producción derivadas del grafo realizado en el punto anterior. Existen tantas reglas como arcos en el grafo. Para el ejemplo de la figura 5.3, aparecerían dos reglas: Una que determina cuando el robot debe tomar una pieza de CONV2, y otra que indica cuando poner la pieza en TAB1.

En cada regla se involucran las condiciones del sitio en cuestión, así como la información necesaria para que se pueda ejecutar la operación. Esto quiere decir que para que el robot de la figura 5.3 pueda "tomar" la pieza de la banda transportadora, tendrían que cumplirse las siguientes condiciones:

- ✓ Que en la banda transportadora existiera una pieza
- ✓ Que la pieza fuera válida
- ✓ Que la pinza del robot estuviera vacía
- ✓ Que exista un espacio vacío en la mesa para poder colocar la pieza

Si alguna de las condiciones anteriores no se cumple, entonces el robot no tomaría la pieza.

Una manera de implementar el control de las actividades es definir conjuntos de reglas, de las cuales sólo una regla puede ser ejecutada a la vez; esto se puede llevar a cabo descomponiendo las tareas de ensamble en subtareas, como se menciona a continuación.

Descomposición de tareas

Para descomponer la tarea general de ensamble en subtareas, el punto clave es considerar a todos los sitios que son controlados por el mismo efector. En el grafo dirigido, los límites de una subtarea son los sitios que están antes y después de un sólo efector, por lo que el número de subtareas es igual al número de efectores que existan. Para la figura 5.3 existe una sola subtarea, que es ejecutada por el efector R1, el cual tiene un sitio llamado GRIP1 que representa la pinza del robot.

Ya que se tienen todas las subtareas, cada una de ellas generará un conjunto de reglas, donde cada regla corresponde a cada una de las posibles operaciones que puede ejecutar un efector. Debido a que no todas las reglas se pueden cumplir al mismo tiempo, sólo una regla puede ser aplicada a la vez. Esto quiere decir que, por ejemplo en la figura 5.3, el robot puede estar tomando la pieza de la banda transportadora o bien poniendo la pieza en la mesa, pero no ejecutando ambas operaciones.

Consideraciones de la programación:

Debe existir una clase Manager, la cual se encarga de controlar y manejar la tarea completa, esto es, a todas las subtareas. Cada conjunto de reglas de una subtarea se implementa como un método de la clase Manager.

El manejador de la tarea completa envía los mensajes a las instancias de manejadores asociadas a las subtareas; esto se puede llevar a cabo para

propósitos de simulación, secuencialmente, pero en caso de una aplicación de control en tiempo real, los mensajes deben ser hechos concurrentemente.

Cada manejador de subtarea puede tener asociado recursos tales como sitios, efectores, sensores, y otros. Por lo tanto, la disponibilidad de cada recurso debe ser verificada, esto por algún método que prueba si un recurso compartido está o no disponible para algún manejador.

5.3.3 Modelando una célula de ensamble

Consideremos el ejemplo del sistema que se muestra en la figura 5.4. A continuación se muestra la manera como se aplicarían los pasos mencionados en el apartado 5.3.2 para este ejemplo en particular.

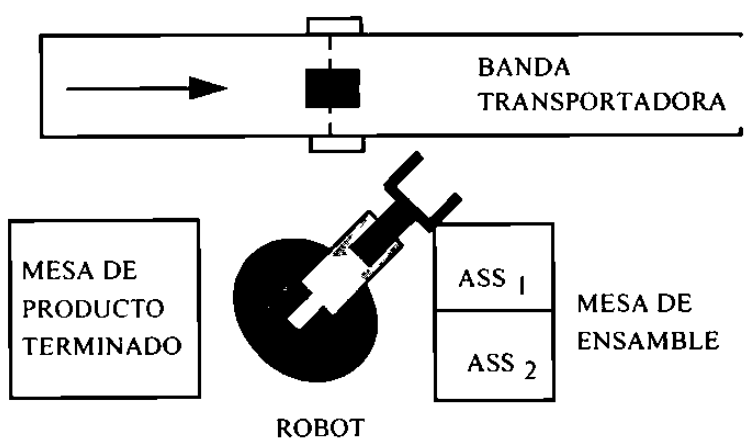


Figura 5.4 Célula de ensamble simple

1. Clasificando a cada uno de los componentes, quedarían de la manera como se muestran en la figura 5.5.

2. El flujo a seguir por los componentes de la célula es el siguiente:

a) La banda transportadora introduce las piezas para que sean procesadas (BANDA1).
b) El robot (ROBOT1) toma las piezas de la banda y las coloca en la mesa de ensamble o máquina para que sean procesadas.
c) Cuando termina el proceso, el robot toma la parte y la coloca en la mesa de producto terminado (MESAPT).

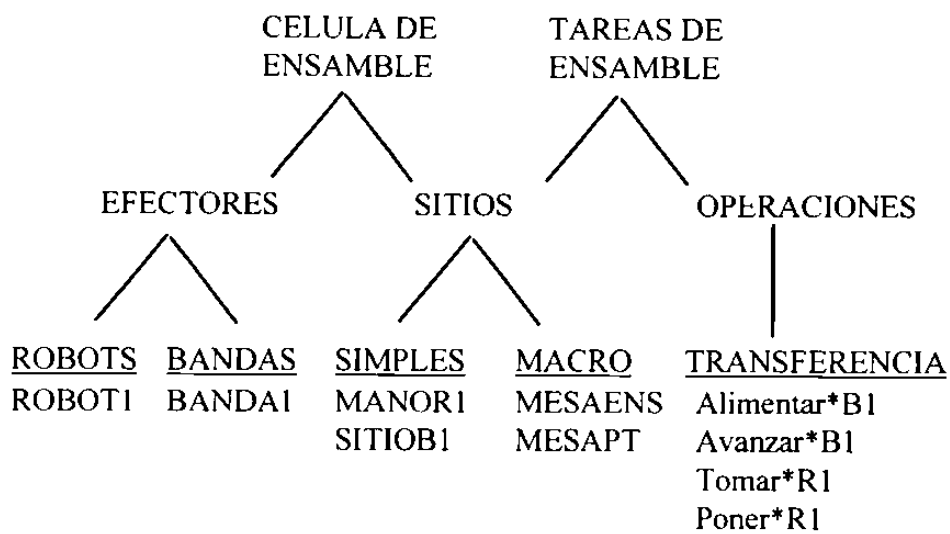


Figura 5.5 Clasificación de célula de la figura 5.4

3. El grafo correspondiente al ejemplo quedaría como se muestra en la siguiente la figura.

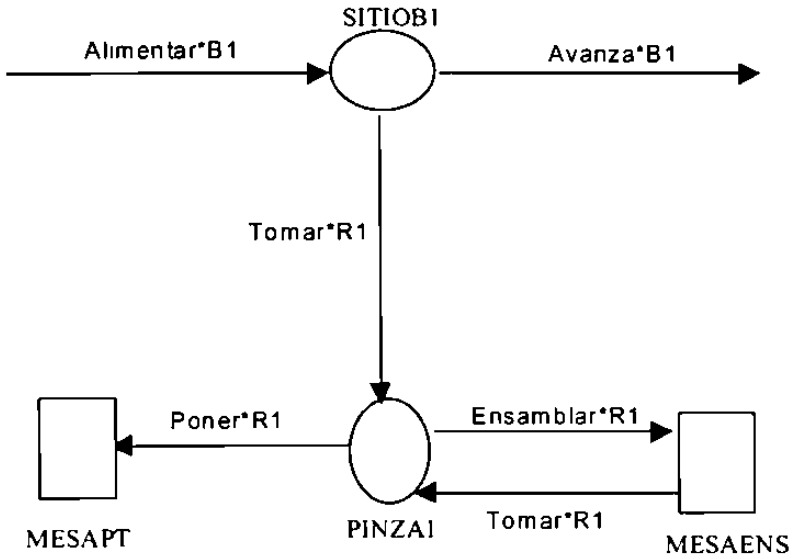


Figura 5.6 Grafo de la figura 5.4

4. Debido a que en el ejemplo aparecen dos efectores, entonces existirán también dos subtareas. Las subtareas 1 y 2 se muestran en las figuras 5.7 y 5.8.

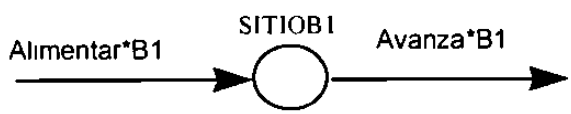


Figura 5.7 Subtarea 1 de figura 5.6

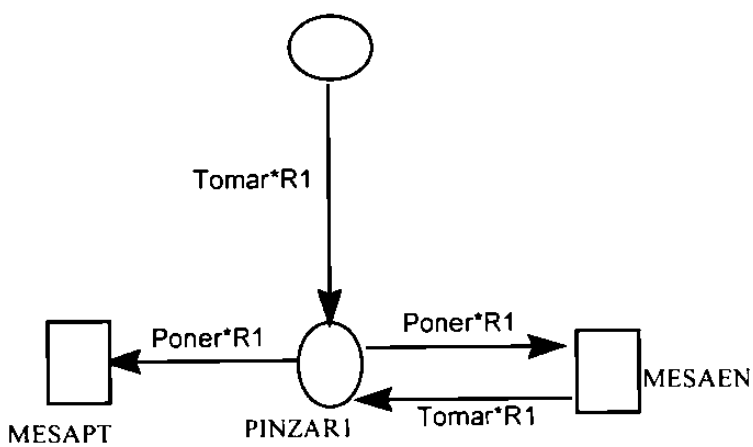


Figura 5.8 Subtarea 2 de figura 5.6

Una vez que tenemos las subtareas se puede pasar a la implementación del programa, tomando en cuenta las consideraciones de programación mencionadas anteriormente en 5.3.2. Por ejemplo, algunas de las reglas implementadas en C++ quedarían de la siguiente manera:

```
If ( (PINZAR1.vacio()) && (SITIOB1.!vacio()) )
{
   R1.tomar( SITIOB1) );
}

If ( (PINZAR1 !vacio()) && (MESAEN.vacio()) )
{
   R1.poner( SITIOB1) );
}
```

5.4 Resumen

En este capítulo se presentó la guia de desarrollo que se propone para modelar los Sistemas de Manufactura Flexibles, los cuales fueron ejemplificados por células de ensamble.

Mostramos que la secuencia de actividades puede ser llevada a cabo de forma secuencial o de manera concurrente, enfocándonos a la sincronización secuencial y considerando que las actividades se ejecutaban en una situación normal.

La guía de desarrollo propuesta se descompuso en dos etapas, las cuales son.

A) Modelación del software de control el cual está basado en conocimiento, pues se obtienen las reglas de producción.

B) La implementación, la cual utiliza la Programación Orientada a Objetos para construir el controlador, aquí también se incluye la descomposición de tareas.

También se ejemplificaron los pasos que se tendrían que seguir para modelar una célula de ensamble sencilla, concluyendo con la manera en cómo tendrían que implementarse las reglas de producción en el lenguaje computacional C++.

CAPÍTULO 6

IMPLEMENTACION DE EJEMPLOS

6.1 Introducción

En esta parte se presenta la manera en que fue implementada y probada la guía de desarrollo propuesta. Básicamente se muestran los pasos que se siguieron para implementar la simulación de los controladores de algunas células de ensamble, los cuales se escribieron en el lenguaje Borland C++ versión 3.1.

6.2 Ejemplo 1

La célula de ensamble que modela el ejemplo 1 está formado por los siguientes elementos:

- 1 Banda transportadora por donde entran las piezas a ser ensambladas.
- 1 Sensor (SENS) de visión ubicado en la banda de entrada, que identifica si llegan piezas.
- 1 Mesa de almacenamiento con 4 lugares o sitios disponibles.

- 2 Mesas de ensamble (ENS1, ENS2) con 2 sitios cada una.
- 1 Robot (R1) que ensambla y/o almacena las piezas que llegan por la banda de entrada.
- 1 Robot llamado R2, que toma el producto terminado de la mesa de ensamble ENS1 y lo pone en la banda transportadora de salida.
- 1 Robot llamado R3, que toma el producto terminado de la mesa de ensamble ENS2 y lo pone en la banda transportadora de salida.
- 1 Banda transportadora de salida (B2) donde se pone el producto terminado.

En la figura 6.1 se muestra el esquema de la célula de ensamble del ejemplo.

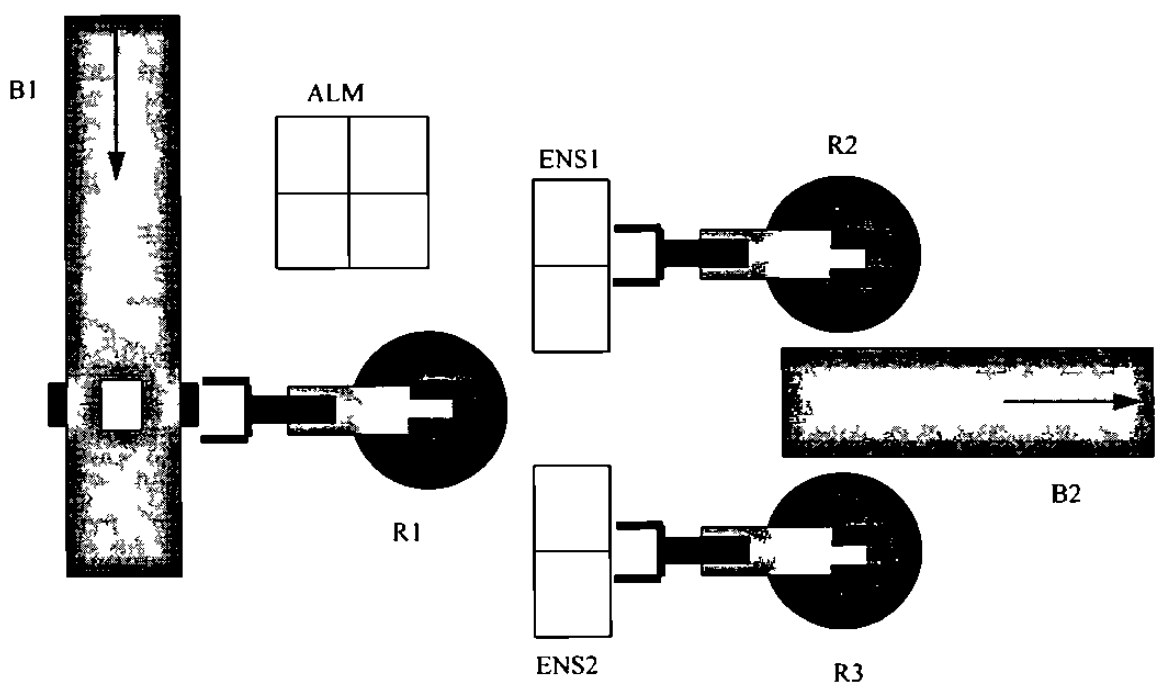


Figura 6.1 Célula de ensamble del ejemplo 1

Este sistema lleva a cabo el ensamble de acuerdo a la tarea que se describe a continuación:

- El ensamblado consiste del montaje de cuatro piezas, las cuales se deberán apilar una sobre la otra. Por cuestiones de simplicidad identificaremos a las piezas como A, B, C y D. En las dos mesas de ensamble deben ser apiladas de la forma como se muestra en la figura 6.2.

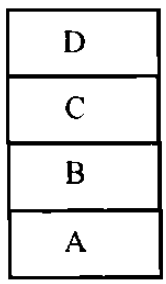


Figura 6.2 Orden de ensamblaje ejemplo 1

- Las piezas llegan por la banda transportadora de entrada (B1), en un orden cualquiera; además de las cuatro piezas mencionadas en el punto anterior, puede llegar cualquier otra pieza.
- Cuando se detecta la llegada de una pieza por medio del sensor SENS (ver figura 6.1), es detenida la banda transportadora de entrada B1 para que sea identificada la pieza.
- Una vez detectada la presencia de una pieza en la banda transportadora, se procede a la identificación de la misma (la pieza es reconocida para saber si pertenece al contexto de ensamble, es decir, si es A, B, C ó D).
- Si la pieza identificada no pertenece al contexto de ensamble, o no se puede ensamblar, o bien no se puede almacenar; entonces el sistema de control le indica a la banda transportadora B1 que avance, para sacar la pieza del sistema.
- Si la pieza que llega es la que continúa para ser ensamblada (en ENS1 ó ENS2), o bien puede ser almacenada en la mesa de almacén (ALM) el robot R1 la toma; si no, la deja pasar indicándole a la banda B1 que avance.

- Si no llega pieza, el sistema de control verifica si en la mesa de almacén (ALM) existe una pieza que pueda ser ensamblada (en ENS1 o ENS2). Si éste es el caso la toma y la ensambla.
- El robot R2 descarga los montajes completos de ENS1 y los deposita en la banda transportadora de salida (B2).
- El robot R3 descarga el producto terminado de ENS2 y lo deposita en la banda de salida (B2).

El grafo que representa a la célula de ensamble quedaría como se muestra a continuación:

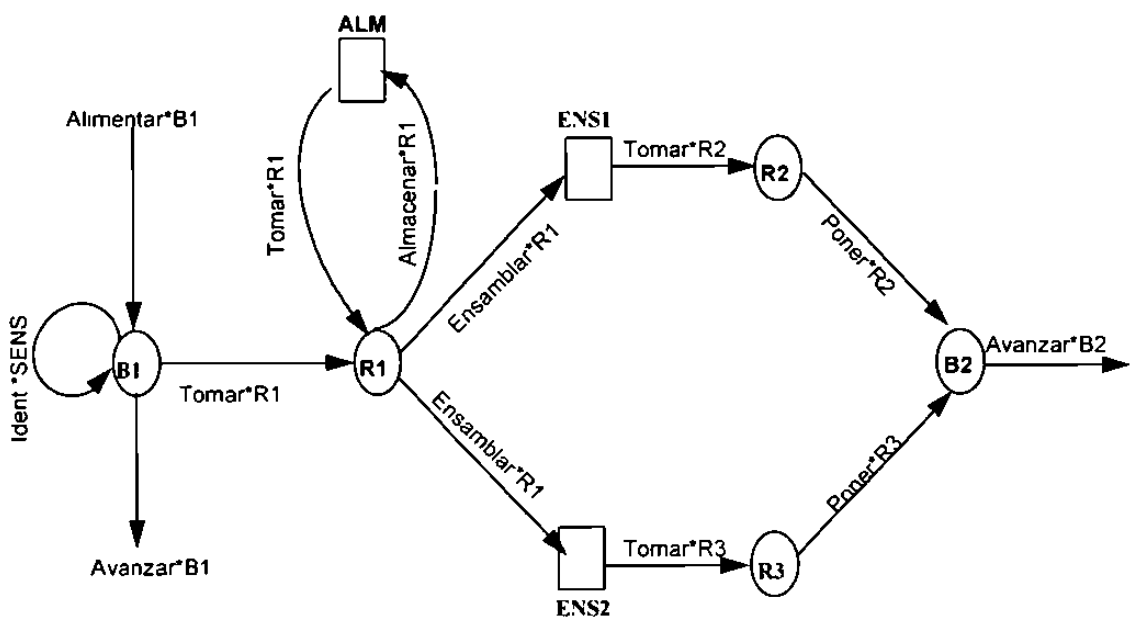
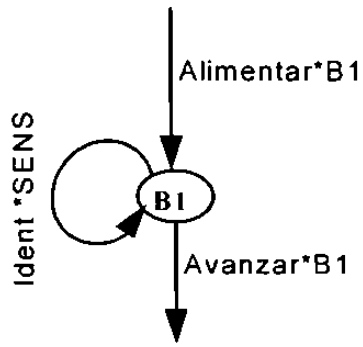
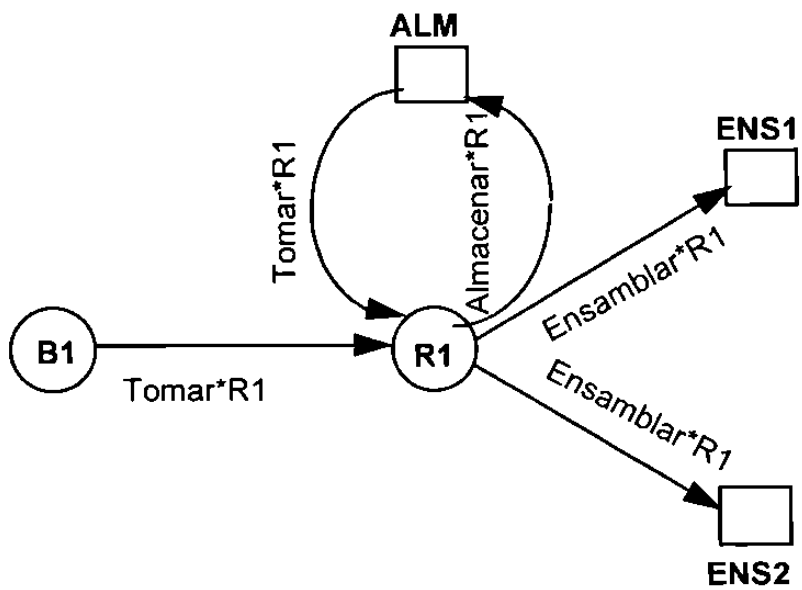


Figura 6.3 Grafo de la célula de ensamble del ejemplo 1

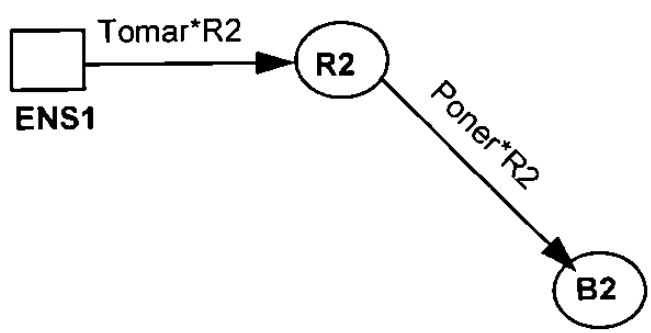
Del grafo mostrado en la figura 6.3 se obtuvieron las subtareas que se muestran en la figura 6.4.



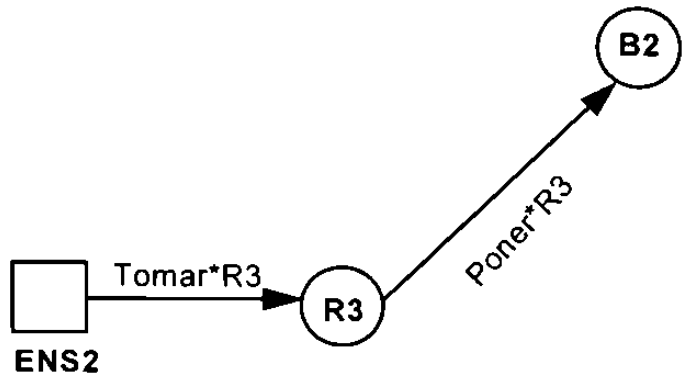
Subtarea 1



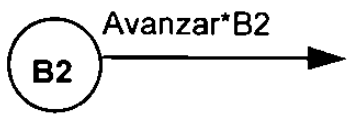
Subtarea 2



Subtarea 3



Subtarea 4



Subtarea 5

Figura 6.4 Subtareas de ejemplo 1

De la figura 6.4 se pueden generar, de una manera simbólica, las reglas de producción para controlar la célula, las cuales se muestran a continuación. El código para esta célula de ensamble se detalla en el apéndice B.

SUBTAREA 1:

Si (En BANDA1 hay una PIEZA **) entonces**:

BANDA1 alimenta la PIEZA hacia el SENSOR

Si (SENSOR está desocupado y BANDA1 tiene PIEZA **) entonces**:

SENSOR Identifica la PIEZA

Si ((PIEZA no es válida) o (PIEZA no se puede ensamblar en ENS1 o ENS2) o (PIEZA no se puede almacenar en ALM) **) entonces**:

BANDA1 avanza para sacar la PIEZA del sistema

SUBTAREA 2:

Si ((BANDA1 tiene PIEZA válida) y (PIEZA se puede ensamblar en ENS1 o ENS2) o (PIEZA puede almacenarse en ALM) y (PINZA1 esta desocupada) **) entonces**:

ROBOT1 toma la PIEZA

Si ((PINZA1 tiene PIEZA) y (PIEZA se puede ensamblar en ENS1) **) entonces**:

ROBOT1 ensambla la PIEZA en ENS1

Si ((PINZA1 tiene PIEZA) y (PIEZA se puede ensamblar en ENS2) **) entonces**:

ROBOT1 ensambla la PIEZA en ENS2

Si (PINZA1 tiene PIEZA) y (PIEZA no se puede ensamblar en ENS1 o ENS2) y (hay espacio en el ALM) **) entonces**:

ROBOT1 almacena la PIEZA en ALM

Si (hay PIEZA en ALM que se pueda ensamblar en ENS1 **) entonces**:

ROBOT1 toma la PIEZA del ALM y la ensambla en ENS1

Si (hay PIEZA en ALM que se pueda ensamblar en ENS2 **) entonces**:

ROBOT1 toma la PIEZA del ALM y la ensambla en ENS2

SUBTAREA 3:

Si ((hay PRODUCTO TERMINADO en la mesa ENS1) y (PINZA2 esta vacía) **) entonces**:

ROBOT2 toma el PRODUCTO TERMINADO de ENS1

Si ((PINZA2 tiene pieza) y (BANDA2 está vacía) **) entonces**:

ROBOT2 pone el PRODUCTO TERMINADO en BANDA2

SUBTAREA 4:

Si ((hay PRODUCTO TERMINADO en la mesa ENS2) y (PINZA3 esta vacía) **) entonces**:

ROBOT3 toma el PRODUCTO TERMINADO de ENS2

Si ((PINZA3 tiene pieza) y (BANDA2 está vacía) **) entonces**:

ROBOT3 pone el PRODUCTO TERMINADO en BANDA2

SUBTAREA 5:

Si (BANDA2 tiene PIEZA **) entonces**:

BANDA2 avanza

6.3 Ejemplo 2

El ejemplo que aquí se presenta es una variante del anterior y la célula de ensamble tiene dos mesas de ensamblaje y en cada una de ellas se deben montar cuatro piezas, las cuales se deberán apilarse una sobre otra en la mesa correspondiente. El orden en que deberán ser apiladas se muestra en la figura 6.5.

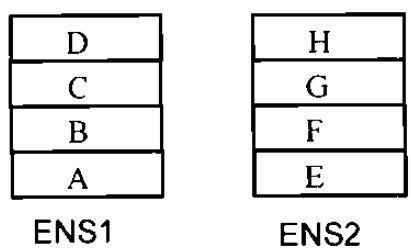


Figura 6.5 Orden de ensamblaje del ejemplo 2

El grafo de la célula de ensamble se presenta en la figura 6.6.

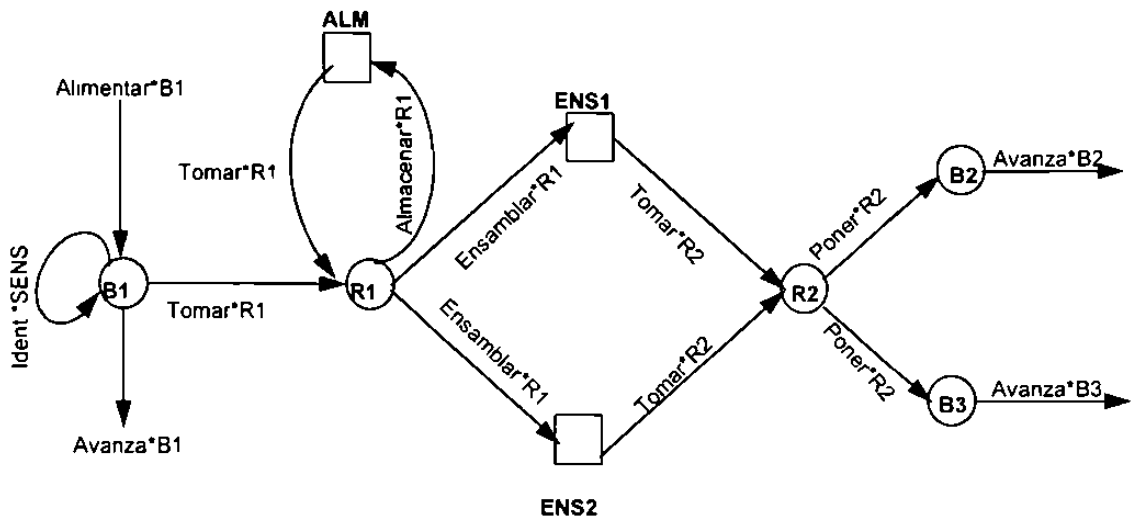
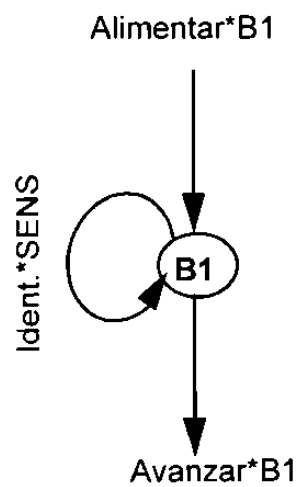
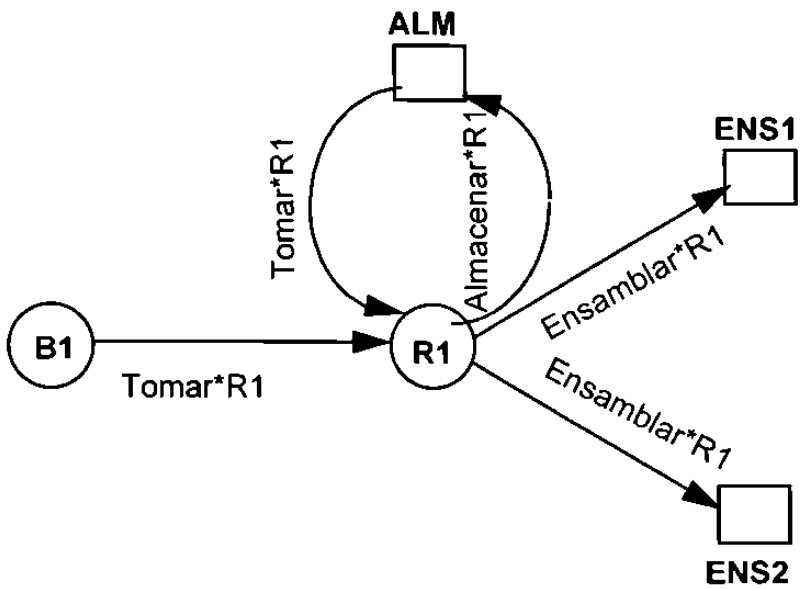


Figura 6.6 Grafo de la célula de ensamble del ejemplo 2

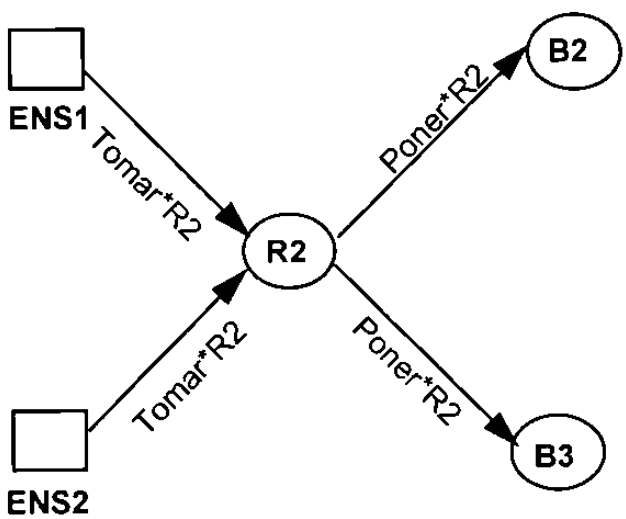
Las subtareas obtenidas correspondientes al grafo de la figura 6.6 se muestran a continuación.



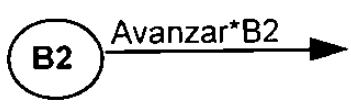
Subtarea 1



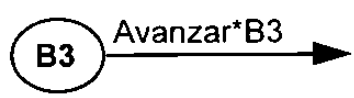
Subtarea 2



Subtarea 3



Subtarea 4



Subtarea 5

Figura 6.7 Subtareas del ejemplo 2

Las reglas simbólicas se muestran a continuación y el código del ejemplo 2 se muestra en el apéndice C.

SUBTAREA 1:

Si (En BANDA1 hay una PIEZA **) entonces**:

BANDA1 alimenta la PIEZA hacia el SENSOR

Si ((SENSOR está desocupado) y (BANDA1 tiene PIEZA) **) entonces:**

SENSOR identifica la PIEZA

Si ((PIEZA no es válida) o (PIEZA no se puede ensamblar en ENS1 o ENS2) o (PIEZA no se puede almacenar en ALM) **) entonces**:

BANDA1 avanza para sacar la PIEZA

SUBTAREA 2:

Si ((BANDA1 tiene PIEZA y es pieza valida) y (PIEZA se puede ensamblar en ENS1 o ENS2) o (PIEZA puede almacenarse en ALM) y (PINZA1 esta vacia) **) entonces**:

ROBOT1 toma la PIEZA de BANDA1

Si ((PINZA1 tiene PIEZA) y (PIEZA se puede ensamblar en ENS1) **) entonces**:

ROBOT1 ensambla en ENS1

Si ((PINZA1 tiene PIEZA) y (PIEZA se puede ensamblar en ENS2) **) entonces**:

ROBOT1 ensambla en ENS2

Si ((PIEZA no se puede ensamblar en ENS1 o ENS2) y (hay espacio en el ALM) **) entonces**:

ROBOT1 almacena la PIEZA en ALM

Si (hay PIEZA en ALM que se pueda ensamblar en ENS1 **) entonces**:

ROBOT1 toma la PIEZA del ALM y la ensambla en ENS1

Si (hay PIEZA en ALM que se pueda ensamblar en ENS2 **) entonces**:

ROBOT1 toma la PIEZA del ALM y la ensambla en ENS2

SUBTAREA 3:

Si ((hay PRODUCTO TERMINADO en ENS1) y (PINZA2 esta vacia) **) entonces**:

ROBOT2 toma el PRODUCTO TERMINADO de ENS1

Si ((hay PRODUCTO TERMINADO en ENS2) y (PINZA2 esta vacia) **) entonces**:

ROBOT2 toma el PRODUCTO TERMINADO de ENS2

Si ((PINZA2 tiene PRODUCTO TERMINADO de ENS1) y (BANDA2 está vacía) **) entonces**:

ROBOT2 pone el PRODUCTO TERMINADO en BANDA2

Si ((PINZA2 tiene PRODUCTO TERMINADO de ENS2) y (BANDA3 está vacía) **) entonces**:

ROBOT2 pone el PRODUCTO TERMINADO en BANDA3

SUBTAREA 4:

Si (BANDA2 tiene PRODUCTO TERMINADO **) entonces**:

BANDA2 avanza

SUBTAREA 5:

Si (BANDA3 tiene PRODUCTO TERMINADO **) entonces**:

BANDA3 avanza

6.4 Resumen

Se han presentado dos ejemplos ilustrativos de células de ensamble, las cuales pueden ser representativas de los Sistemas de Manufactura Flexibles.

Estos ejemplos han sido primeramente modelados utilizando la guía de desarrollo para controladores de FMS, para que posteriormente sean implementados en lenguaje Borland C++.

Algunas de las ventajas de utilizar la programación orientada a objetos para simular los Sistemas de Manufactura Flexibles, es que permite una definición rápida y natural de los componentes, acortando el ciclo de desarrollo de software de control, obteniendo programas modulares y readaptables.

CAPÍTULO 7

CONCLUSIONES Y RECOMENDACIONES

7.1 Conclusiones

En este trabajo se abordó el problema de la programación de controladores de FMS atendiendo principalmente a la secuenciación de tareas que son llevadas a cabo en situaciones de ejecución normal.

Se propuso una guía para desarrollar controladores de Sistemas de Manufactura Flexibles (FMS), la cual se enfoca básicamente en el nivel de coordinación de estos sistemas.

Con el propósito de probar que la guía propuesta es viable para controlar los Sistemas de Manufactura Flexibles, se lograron simular algunos ejemplos de coordinadores de células de ensamble, las cuales son representativas de estos sistemas.

Se comprobó que para el modelado de software de control de Sistemas de Manufactura Flexibles, es viable el enfoque basado en conocimiento, ya que se pueden obtener las reglas de producción y marcos como formas de representar los datos; dando la pauta para construir bases de conocimiento condensadas y modulares para especificar las tareas de ensamble.

Se confirmó una vez más que la Programación Orientada a Objetos (POO) permite una definición rápida y natural de objetos, clases y métodos; acortando el ciclo de desarrollo de software para el control de Sistemas de Manufactura Flexibles, obteniendo programas modulares y readaptables.

Por medio de la implementación de ejemplos, se encontró que las capacidades que ofrece el lenguaje C++ 3.1 facilitan la programación para realizar simulaciones de todo tipo de sistemas, entre ellos los Sistemas de Manufactura Flexibles.

Se mostró que para realizar simulaciones con lenguajes computacionales es necesario contar con una guía de desarrollo, que nos especifique los pasos que se tendrían que seguir para modelar Sistemas de Manufactura Flexibles.

7.2 Trabajos futuros

Debido a que la coordinación de actividades en los Sistemas de Manufactura Flexibles considerada en este trabajo fue de manera secuencial, se recomienda que también sea enfocado de una manera concurrente. Esto es, donde dos actividades o más se ejecutan al mismo tiempo.

Se aconseja que la guía de desarrollo propuesta sea probada con otros diferentes ejemplos de Sistemas de Manufactura Flexibles, ya que aquí sólo se tomaron las células de ensamble como representativas de tales sistemas.

Otra posible sugerencia es que, se exploren otras posibles alternativas para la descomposición de tareas, para así poder comparar y evaluar la guía que aquí se propone.

Se puede realizar un estudio formal, en cuanto a los beneficios que son obtenidos al utilizar tanto el enfoque basado en conocimiento como el de orientación a objetos.

También se recomienda que este trabajo sea continuado con el objetivo de elaborar una metodología integrada de computación para la simulación de Sistemas de Manufactura Flexibles.

APENDICE A

PROGRAMACION ORIENTADA A OBJETOS EN C++

Aquí presentamos como pueden ser aplicados los conceptos definidos en el capítulo cuatro utilizando en el lenguaje de programación C++ de Borland.

Clases

El C++ permite construir clases, que proporciona al programador una herramienta sencilla y poderosa para definir nuevos tipos de datos, también llamados tipos de datos abstractos (TDA). Algunos beneficios que hacen atractivo el uso de los tipos de datos propios pueden ser:

1. Los programas escritos son mas fáciles de entender y modificar.
2. Se pueden evitar muchos errores de diseño. Como los objetos se pueden manejar sólo mediante sus operaciones, los errores se localizan fácilmente.
3. Como cada tipo de datos es una entidad independiente, la tarea de diseño se puede dividir en varias subtareas que se llevan a cabo de forma aislada.

En C++, los nuevos tipos de datos ó clases se definen usando el siguiente formato:

```
class identificador-clase {
        private:
                datos y funciones privados    // No se pueden accesar directamente
        protected:
                datos y funciones protegidos // Pueden accesarse por clases derivadas
public:
                datos y funciones públicos     // Se pueden accesar directamente
} nombres-de-objetos;
```

donde *nombres-de-objetos* puede estar vacío.

Algunas características importantes de las clases en C++ son:

- Una clase puede contener partes privadas, partes protegidas y partes públicas. Por defecto, todos los elementos definidos en una clase son privados.

- En la parte privada se definen datos y/o funciones que no se pueden accesar directamente (es decir, está oculta al exterior).

- En la parte protegida los datos y/o funciones definidos están ocultos al exterior, pero pueden accesarse por clases derivadas de esta clase.

- En la parte pública pueden estar los datos y/o funciones que están accesibles para otras partes del programa, esta es una interfaz con la clase. Aunque puede tener variables public, debe tratar de limitar su uso. En vez de esto, debe hacer todos los datos privados y controlar el acceso a ellos a través de las funciones públicas.

- Los datos y las funciones que son parte de la clase se llaman miembros. Algunos autores utilizan el nombre de método como sinónimo de funciones miembro.
- Los métodos ó funciones definidas dentro de la parte pública, sí pueden accesar la sección privada de la clase.

- Dentro de la declaración de una clase se debe incluir el prototipo completo de las funciones miembro.

- La implantación de las funciones miembro pueden formar parte de la clase o definirse en forma separada. Al definir una función miembro separada de la definición de la clase, esta debe estar precedida por el nombre de la clase para indicar que pertenece a ella.

- Una vez definida la clase, su nombre denota un nuevo tipo de datos, lo que nos permite definir variables de ese tipo.

- Para llamar a una función miembro desde una parte del programa, se hace mediante el operador punto (.). Esto es, debe usar el nombre del objeto seguido por el operador punto y enseguida el nombre de la función miembro a utilizar. El formato es el siguiente:

  Nombre_Objeto . Nombre_Función(parametros) ;

- Las operaciones de una clase se aplican a sus objetos, pero nunca a la propia clase. Por lo tanto, una clase es un concepto que no tiene otra existencia concreta que la reflejada por sus objetos.

A continuación se muestran ejemplos de definición de clases, declaraciones de los objetos y la utilización de mensajes, los cuales fueron obtenidos de [19].

// Prog2.cpp.- Programa que crea y usa una clase llamada cola

```
#include<iostream.h>

class cola
{               // Parte privada· Solo se puede accesar por las funciones miembro.
  int c[100];  // Datos miembro
  int ppio, fin;
public:
  void ini(void);
  void meter(int i);
  int sacar(void);
};

void cola·:ini(void)  // Método de la clase Cola
{
      fin=ppio=0;
}                     // Termina método ini()

void cola::meter(int i)   // Método de la clase cola
{
  if(ppio= =100)
   {
     cout << " La cola esta llena ";
     return;
   }
  ppio++;
  c[ppio]=i;
}                             // Termina método meter()

int cola..sacar(void)       // Método de la clase cola
{
  if(fin==ppio)
  {
     cout << " La cola esta vacia ";
     return 0;
   }
  fin++;
  return c[fin];
}    // Termina método sacar()

main(void)
{               // Inicia programa principal
  cola a,b;    // Crea dos objetos del tipo cola
  a.ini();
  b.ini();
  a.meter(10);
  b meter(19);
  a.meter(20);
  b.meter(1);
  cout << a.sacar() << "\n ";
```

```
  cout << a.sacar() << "\n ";
  cout << b.sacar() << "\n ";
  cout << b.sacar() << "\n";
  return 0;
}
```

// Prog3.cpp Programa que crea y usa una clase llamada counter

```
#include<iostream.h>   // Para las funciones de E/S estandar
#include<conio.h>      // Para usar clrscr(), limpiar pantalla.

class counter
{
 private:         // Parte privada de clase: Solo puede accesarse por funciones miembro
   unsigned int value;   // Dato miembro
 public:
   counter() {  value=0;  }
   void increment() { if (value<65535) value++;};
   void decrement() { if (value>0) value--;};
   unsigned int access_value() { return value;}
};

main()
{
 counter c1;      // Objeto de la clase counter
 counter c2;      // Objeto de la clase counter
 clrscr();
 for( int i=1; i<=15; i++)
 {
   c1.increment();
   cout << "\n c1 = " <<  c1.access_value();
   c2.increment();
 }
 cout << "\n Valor de c2 despues del ciclo = " <<  c2.access_value();
 for(i=1; i<=5; i++)
    c2.decrement();
 cout << "\n Valor final de c2 = "  <<  c2.access_value();
 cout << "\n Valor final de c1 = " <<  c1.access_value();
 return 0;
}
```

Constructores y Destructores

En casi todos los programas es necesario la inicialización de las variables. En nuestro caso, se requiere de la inicialización de los objetos de una clase. C++ proporciona un medio para definirlos e inicializarlos al mismo tiempo.

La inicialización se da de una manera automática a través de ciertas funciones miembro especiales, llamadas constructores. Una función constructora es una función que es miembro de la clase y tiene el mismo nombre que esa clase [3,19,20,23]. Por ejemplo:

// Segmento de programa que muestra el constructor cola:

```
#include<iostream.h>
class cola {
   int c[100];
   int ppio, fin;
 public:
   cola (void);   // Constructor
   void meter(int i);
   int sacar(void);
};

cola :: cola(void)          // Esta es la función constructora:
{
   fin=ppio=0;
  cout << " Cola inicializada \n";
}
```

A continuación se muestra un segmento de programa para el constructor llamado Point:

```
class Point
{
    int xVal, yVal;
 public:
    Point (int x, int y) { xVal=x; yVal=y; }      // Constructor
    void OffsetPt (int, int);
};
```

Una clase puede tener más de un constructor, pero estos deben diferir en el número de sus argumentos para que el compilador pueda elegir el apropiado. Por ejemplo:

// Segmento de programa con diferentes constructores:

```
class Point
{
    int xVal, yVal;
 public:
    Point (int x, int y)  { xVal=x; yVal=y; }   // Argumentos enteros
    Point (float , float);                      // Argumentos flotantes
    Point () { xVal=yVal=0; }                   // Origen
    void OffsetPt (int, int);
};

Point :: Point (float len, float angle)          // Argumentos flotantes
{
   xVal = (int) (len*cos(angle));
   yVal = (int) (len*sin(angle));
}
```

El ejemplo anterior ofrece tres constructores distintos. Por lo tanto, podemos definir:

```
Point pt1(10,20);
Point pt2(60.3 , 3.14 );
Point pt3;
```

El uso de los constructores garantiza que no se pasará por alto la inicialización adecuada de los objetos.

El complemento del constructor es el destructor, el cual se utiliza para que un objeto libere memoria que previamente se le ha asignado o reservado. Esto es importante cuando un objeto está almacenado en el segmento del montón ó heap (por medio de la función new). Tal almacenamiento sólo puede ser liberado con la función delete ó mediante un tipo especial de función miembro llamado destructor.

De la misma manera en que un constructor inicializa un objeto cuando se le declara, un destructor lo suprime cuando su nombre deja de tener sentido. Un

destructor siempre tiene el mismo nombre de la clase, pero va antecedido por el símbolo ~.

Observe los programas que se presentan a continuación para comprender como funcionan los constructores y destructores.

```
// Prog8.cpp.- Programa que muestra constructores y destructores
#include<iostream.h>
#include<conio.h>

class cola
{
   int c[100];
   int ppio, fin;
 public:
   cola (void);     // Constructor
   ~cola (void);   // Destructor
   void meter(int i);
   int sacar(void);
};

cola :: cola(void)        // Esta es la función constructora
{
  fin=ppio=0,
  cout << "Cola inicializada \n ";
}

cola :: ~cola(void)    // Esta es la función destructora
{
   cout << " Cola destruida \n ";
}

void cola :: meter(int i)
{
   if(ppio= =100) {
     cout << " La cola esta llena ";
     return;
   }
   ppio++;   c[ppio]=i;
}

int cola::sacar(void)
{
   if(fin==ppio)
   {
     cout << " La cola esta vacia ";
     return 0;
```

```
  }
  fin++;
  return c[fin];
}

main()
{
  clrscr();
  cola a,b;  // Crea dos objetos tipo cola
  a.meter(10);
  b.meter(19);
  a.meter(20);
  b.meter(1);
  cout << a.sacar() << " ";
  cout << a.sacar() << " ";
  cout << b.sacar() << " ";
  cout << b.sacar() << "\n";
  getch();
  return 0;
}   // Termina Prog8.cpp
```

Es posible declarar un vector de objetos; para esto es necesario definir un constructor sin argumentos, debido a que con los argumentos por default no pueden inicializar a todos los objetos del vector [3,19,20,23].

// Prog10.cpp - Ejemplo de un vector de objetos

```
#include<iostream.h>

class punto
{
  private:
     int x1,x2;
     void init(int x, int y)
     {
           x1=x;
           x2=y;
     }
  public:
     punto(int x, int y)   // Constructor con parametros
     {
         x1=x;
         x2=y;
     }

     punto()  // Constructor sin parametros
     {
```

```
    init(0,0);
  }
  int x_cord() { return x1; }
  int y_cord() { return x2; }
};

// Bloque principal :

main()
{
  punto data(3,4);
  cout << "\n Coordenana y = " << data.y_cord();
  punto more_data[20];  // Arreglo de objetos
  cout << "\n Coordenada x del indice 18 = " << more_data[18].x_cord();
}
```

Herencia

El C++ soporta la herencia permitiendo a una clase incorporar otra clase dentro de su declaración. Obteniendo como resultado, que una clase herede los atributos otra clase [3,19,20,23].

Una clase que es heredada por otra se llama **clase base** o **clase padre**. La clase que hereda se llama **clase derivada** o **clase hijo**. La forma general para la herencia en C++ es:

```
class nombre-nueva-clase: acceso clase-heredada
{
  // cuerpo de la nueva clase
}
```

donde **acceso** es opcional y puede ser **public**, **private** o **protected**. Por omisión es **private** (privado). A continuación se muestran ejemplos donde se usa la herencia.

// Prog18.CPP - Programa que muestra la herencia , creando dos
// subclases de la clase vehiculo_ruedas llamadas camión y automóvil.

```
#include<iostream.h>
#include<conio.h>

class vehiculo_ruedas
{
   int ruedas;
   int pasajeros;
 public:
   void est_ruedas(int num);
   int obt_ruedas(void);
   void est_pasaj(int num);
   int obt_pasaj(void);
};

class camion:public vehiculo_ruedas
{
   int carga;
 public:
   void est_carga(int tamano);
   int obt_carga(void);
   void mostrar(void);
};

enum tipo {coche, camioneta, furgon};

class automovil:public vehiculo_ruedas
{
   enum tipo tipo_coche;
 public:
   void est_tipo(enum tipo t);
   enum tipo obt_tipo(void);
   void mostrar(void);
};

void vehiculo_ruedas::est_ruedas(int num)
{
  ruedas=num;
}

int vehiculo_ruedas::obt_ruedas(void)
{
   return ruedas;
}
void vehiculo_ruedas::est_pasaj(int num)
{
   pasajeros=num;
}
```

```
int vehiculo_ruedas::obt_pasaj(void)
{
   return pasajeros;
}

void camion::est_carga(int num) {
   carga=num;
}

int camion::obt_carga(void)
{
  return carga;
}

void camion::mostrar(void)
{
   cout << "Ruedas: " << obt_ruedas() << "\n";
   cout << "Pasajeros: " << obt_pasaj() << "\n";
   cout << "Capacidad de carga en pies cubicos: " << carga << "\n";
}

void automovil::est_tipo(enum tipo t)
{
  tipo_coche=t;
}
enum tipo automovil::obt_tipo(void)
{
   return tipo_coche;
}

void automovil::mostrar(void)
{
   cout << "Ruedas: " << obt_ruedas() << "\n";
   cout << "Pasajeros: " << obt_pasaj() << "\n";
   cout << "Tipo: ";
   switch(obt_tipo()) {
                        case camioneta: cout << "camioneta\n";
                                break;
                        case coche: cout << "coche\n";
                                break;
                        case furgon: cout << "furgon\n";
    } // Termina el switch
} // Termina mostrar()

main()
{
  camion c1,c2;
  automovil c;
  clrscr();
  c1.est_ruedas(18);
  c1.est_pasaj(2);
```

```
  c1.est_carga(3200);
  c2.est_ruedas(6);
  c2.est_pasaj(3);
  c2.est_carga(1200);
  c1.mostrar();
  c2.mostrar();
  c.est_ruedas(4);
  c.est_pasaj(6);
  c.est_tipo(camioneta);
  c.mostrar();
  cout << "Pulse una tecla ... ";
  getch();
  return 0;
}
```

Herencia múltiple

Una clase puede heredar atributos de dos o más clases. Para esto, en la clase derivada se listan las clases padre separadas por comas [3,19,20]. Por ejemplo, en el siguiente programa **Z** hereda de **X** y de **Y**.

```
// PROG21.CPP Muestra la herencia múltiple
#include <iostream.h>
#include <conio.h>

class X
{
  protected:
    int a;
  public:
    void hacer_a(int i);
};

class Y
{
  protected:
    int b;
  public:
    void hacer_b(int i);
};

class Z: public X,public Y  // Z hereda de X y Y
{
  public:
    int hacer_ab(void);
};
```

```
void X::hacer_a(int i)
{
  a=i;
}

void Y::hacer_b(int i)
{
  b=i;
}

void Z::hacer_ab(void)
{
  return a*b;
}

main(void)
{
  Z i;
  i.hacer_a(10);
  i.hacer_b(12),
  cout<< i.hacer_ab();
  getch();
  return 0;
}
```

En este ejemplo, **Z** tiene acceso a las porciones públicas y protegidas de **X** y de **Y**.

APÉNDICE B

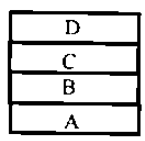
CODIFICACIÓN DEL EJEMPLO 1

```
*
 EJEMPLO1.CPP          Realizó: Ing. Carlos Alonso Camacho Ramírez

SISTEMA DE ENSAMBLE COMPUESTO POR:

        1 Banda de entrada (b1  e )
        1 Sensor de visión (c1)
        1 Robot (r1) que almacena y o ensambla
        1 Mesa de almacenamiento con 4 lugares ( alm[4] )
        2 Mesas de ensamble con 2 lugares c u  ( ens1[2], ens2[2] )
        1 Robot (r2) que toma el producto terminado de ens1 y lo pone en la banda de salida
        1 Robot (r3) que toma el producto terminado de ens2 y lo pone en la banda de salida
        1 Banda transportadora de salida (b2  s), por donde sale el producto ensamblado

LOS ENSAMBLES SON DE LA SIGUIENTE MANERA: ( En ens1 y ens2)
```




```
*

#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<dos.h>
#include<iomanip.h>

const
   maxens  2,maxalm  4,minpza  'A',maxpza  'D',vacio  ' ', neg  maxpza+1;

enum
        bool{no,si};

bool pieza  valida(char pza);      Funcion que checa si la pieza es válida

void esp(void);                    Función que espera 2 segundos

class celula                       Celula de ensamble
{
 protected:
   char contenido;                 Partes a ensamblar
   char producto;                  Productos ensamblados
};
```

```
class sitio:private celula          Lugar donde las partes quedan estables
{
 protected:
    char nombre[12];
 public.

   sitio()                  Constructor sin parametros
    {
          contenido  vacio;
    }

    sitio(char n[12])          Constructor con 1 parametro
    {
          contenido  vacio;
          strcpy(nombre,n);
    }

    sitio(char n[12],char pieza)     Constructor con 2 parametros
    {
          contenido  pieza;
          strcpy(nombre,n);
    }

     sitio(void){ };              Destructor

    char conte() {   return contenido;   }

    char prod() {   return producto;   }

    char* nom() {   return nombre;   }

    bool esta  vacio(sitio sit);          Retorna SI. si el sitio está vacio

    int hay  espacio(sitio ar[]);         Si hay espacio en ar[] retorna la posición, si no retorna neg
    int puedo  ens(sitio de[], sitio ar[], bool band);   Si se puede ensamblar en ar[],
                                                         retorna la posición, si no retorna neg

    int hay_pdcto(sitio ar[]);          Checa si existe pdcto.terminado en mesa ensamble

    inter(sitio &a, sitio &b);          Pone el contenido de a en b, y en a pone vacio

    bool hay_pieza();

    char ident_pieza();

    void establece(char valor) { contenido  valor; }

    void sitio::muestra(sitio ban1,sitio pinz1,sitio ens1[],sitio ens2[],sitio alm[],sitio pinz2,sitio pinz3,
                        sitio ban2);
};
```

```
bool sitio::esta  vacio(sitio sit)
{
  if(sit.contenido    vacio)  return si;
  else return no;
}

int sitio::hay  espacio(sitio ar[] )   Retorna la posición i del arreglo si
                                       hay espacio en él , si no retorna neg
{
 for(int i  0; i<maxalm; i++)
 {
    if(esta  vacio(ar[i]))
          return(i),
 }
 return(neg);
}

sitio::puedo  ens(sitio de[], sitio ar[], bool band  0)
{

 if( band    1)  Si band  1, checa si alguna pieza del almacen puede ensamblarse
 {
  for(int j  0; j<maxalm; j++)
   for(int i=0; i<maxens; i++)
   {
    if( (de[j].conte()-ar[i].conte())    1 )
          return j;
    if( (esta  vacio(ar[i])) && (de[j].conte()    minpza) )
          return j,
   }
 }
 else   Si band  0, checa si la pieza de.conte() puede ensamblarse
 {
   for(int i=0; i<maxens; i++)
   {
    if( (de[0].conte()-ar[i].conte())    1 )
          return i;
    if( (esta  vacio(ar[i])) && (de[0].conte()  ¬minpza) )
          return i;
   }
 }
 return (neg);
}

int sitio::hay_pdcto(sitio ar[])
{
   for(int i  0; i<maxens; i++)
    if( ar[i].conte()    maxpza )
          return i;
   return neg,
}
```

```
sitio::inter(sitio &a, sitio &b)
{
   b contenido  a.contenido;
   a.contenido  vacio;
}


class efector:private celula
{
  char tipo[10];
  char funcion[15];
  int localizacion,
};

class robot:private efector, private sitio
{
 public:
   robot(char n[10]) : sitio(n) { }   Usa el constructor de la clase padre sitio
    robot(void){ };
   bool tomar(sitio &de, sitio &conque,bool band);
   bool poner(sitio &de, sitio &donde, bool band);
   bool ensamblar(sitio &de, sitio arr[]);
   bool almacenar(sitio &de, sitio arr[]);
},

bool robot::tomar(sitio &de, sitio &conque,bool band  0)
{
    if (esta  vacio(de))
    {
          cout << "El Robot no puede tomar, ya que ";
          cout << de.nom() << " tiene ";
          cout << de.conte();
          return(no);
    }
    if (!esta  vacio(conque))
    {
          cout << conque.nom() << " esta ocupado, tiene: ";
          cout << conque.conte();
          return(no);
    }
    if (band!  0)
           cout << "\nSe está tomando el producto terminado";
    else
           cout << "\nSe está tomando la pieza " << de.conte();
    cout << " de " << de.nom();
    inter(de,conque);
    esp();
    return(si);
 }
```

```
bool robot::poner(sitio &de, sitio &donde, bool band  0)
{
   if(esta  vacio(de))
         return no;

   if (!esta  vacio(donde))
   {
         cout << "El Robot no puede poner, porque ahí esta: ";
         cout << donde.conte();
         return(no);
   }

   if (band!  0)
          cout << "\nSe está poniendo el producto terminado";
   else
          cout << " n nSe está poniendo la pieza " << de.conte();

   cout << " en " << donde.nom();
   inter(de,donde);
   esp();
   return(si);

}      Termina poner()

bool robot  ensamblar(sitio &de, sitio arre[])
{

  bool exito  no;
  sitio dee[1];
  dee[0]  de;

  int i  puedo  ens(dee,arre);
  if( i!  neg )
  {
          cout << "\nSe está ensamblando " << de.conte() << " de " << de.nom();
          cout << " en "   < arre[i].nom();
          inter(de, arre[i]);
          esp();
          exito  si;
  }

  return(exito),

}      Termina ensamblar


bool robot::almacenar(sitio &de, sitio arr[] )
{
  if(de.conte()    vacio)
    return no;
  int j;
  bool exito  no;
  j  hay  espacio(arr);
```

```
  if(j != neg)
  {
    cout << "\nSe está almacenando la pieza ";
    cout << de.conte();
    cout << " en la posicion " << j+1 << " de la mesa";
    inter(de,arr[j]);
    exito = si;
    esp();
  }
  else    No se pudo almacenar
    cout << "\nEstá ocupada la mesa de " << arr[0].nom();
  return(exito);
}     Termina almacenar()

class banda:private sitio, private efector
{
 public:
   avanzar(sitio &sit);    Avanza la banda transportadora
   alimentar();
};

bool sitio::hay_pieza()
{
  char llega;
  cout << "\n\nLLEGA PIEZA (S/N)?: ";
  cin >> llega;
  if(llega == 'S')
    return si;
  else
    return no;
}

banda :avanzar(sitio &sit)
{
  sitio basu;
  cout << "\nLa " << sit.nom() << " está avanzando\n";
  inter(sit,basu);
  esp();
}

banda::alimentar()
{
  cout << "\nLa pieza esta entrando por Banda1";
  esp();
}

char sitio ::ident_pieza()
{
  char pieza,
  cout << "\nIdentifica la pieza(S = Salir): ";
  cin >> pieza;
  return pieza;
}
```

```
class sensor:private celula
{
 public:
   char ident_pieza();
};

 / Muestra el contenido de un sitio:
void sitio::muestra(sitio ban1,sitio pinz1,sitio ens1[],sitio ens2[],
                    sitio alm[],sitio pinz2,sitio pinz3, sitio ban2)
{
  cout << "\nESTADO DE LA CELULA: ";
  cout << ban1.nom() << ":" << ban1.conte() << flush;
  cout << ", " << pinz1.nom() << ":" << pinz1.conte() << flush;

  cout << ", Ensamble1: ";
  for(int i  0; i<maxens; i++)
        cout << ens1[i].conte() << ", " << flush;

  cout << "Ensamble2: ";
  for(i  0, i<maxens; i++)
        cout << ens2[i].conte() << ", " << flush;

  cout << "\n Almacen: ";
  for(i  0; i<maxalm; i++)
        cout << alm[i].conte() << ", " << flush;

  cout << pinz2.nom() << ":" << pinz2.conte() << flush;
  cout << ", " << pinz3.nom() << ":" << pinz3.conte() << flush;
  cout << ", " << ban2.nom() << ":" << ban2.conte() << "\n" << flush;
}

bool pieza  valida(char pza)       Checa si la pieza es valida
{
 if ( (pza>  minpza) && (pza<  maxpza) )
    return(si);
 else
    return(no);
}

void esp(void)    Función que espera 2 segundos
{
 delay(1000);
}

main()
{
  clrscr();
    Inicia definicion e inicialización:
  char pieza;  banda b1,b2;
  sensor sens1;
  sitio b1  e("Banda1"), b2  s("Banda2"), bas;
  sitio ens1[2]  {"Ensamble1,1","Ensamble1,2"}, ens2[2]-{"Ensamble2,1","Ensamble2,2"};
  sitio alm[4]  {"Almacen.1","Almacen.2","Almacen.3","Almacen.4"};
  sitio pinza1("Pinza1"), pinza2("Pinza2"), pinza3("Pinza3");
```

```
robot r1("Robot1"), r2("Robot2"), r3("Robot3");
bas.muestra(b1_e,pinza1,ens1,ens2,alm,pinza2,pinza3,b2_s);  // Muestra el estado de la celula:
do
{
   pieza vacio;                    // Inicialmente pieza esta vacio
   sitio sens1("Sensor1",vacio);   Inicializa el Sensor1 con vacio
   int PosAlm_neg;                  Posicion en el ALMACEN de la pieza
   int PosTerm_neg;                 Posicion en mesa de ensamble del producto terminado

 *                                                    SUBTAREA1:
 ---------------------------------------------------------------------------------------------------------------------- */

  Si hay pieza, se alimenta por la Banda1 y se inicializa con '?'
 if(b1_e.hay_pieza()   si)
 {
       b1.alimentar();
       b1_e.establece('?');   Pone en Banda1 el contenido '?'
 }

  Si el Sensor1 esta vacio y Banda1 tiene pieza, la identifica:
 if(sens1.conte()   vacio && b1_e.conte()   '?')
 {
  pieza b1_e.ident_pieza();    Captura la pieza para identificarla
  b1_e.establece(pieza);
 }

  Checa si la pieza es o no procesable:  Si no es pieza valida o no se puede ensamblar en Ensamble1 o
  Ensamble2 y no hay espacio en almacen, entonces la pieza no es procesable.

 if( (!pieza_valida(b1_e conte()))
       ( (bas.puedo_ens(&b1_e,ens1)   neg) &&
         (bas.puedo_ens(&b1_e,ens2)   neg) &&
         (bas.hay_espacio(alm)   neg) )
  )
 {
       cout << " nPieza de " << b1_e.nom() << " no procesable";
       b1.avanzar(b1_e),
       bas.muestra(b1_e,pinza1,ens1,ens2,alm,pinza2,pinza3,b2_s);
 }

 *                                                    SUBTAREA2:
 ----------------------------------------------------------------------------------------------------------------------*

Si Banda1 tiene pieza valida y Pinza1 está vacia, entonces la toma:

 if( (b1_e.conte()!_vacio)
       && (pieza_valida(b1_e.conte()))
       && ( (bas.puedo_ens(&b1_e,ens1)!_neg)  (bas.puedo_ens(&b1_e,ens2)!_neg)
          (bas.hay_espacio(alm)!_neg) )
       && (pinza1.conte()   vacio)
  )
  {
        r1.tomar(b1_e,pinza1);  // Toma la pieza de banda1 con pinza1
        bas.muestra(b1_e,pinza1,ens1,ens2,alm,pinza2,pinza3,b2_s);
  }
```

```
Si Pinza1 tiene pieza y se puede ensamblar en ensamble1 entonces la ensambla:
if( (pinza1.conte()!  vacio) && (bas.puedo  ens(&pinza1,ens1)!  neg) )
 {
      r1.ensamblar(pinza1,ens1);
      bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,pinza3,b2  s);
 }

/ Si Pinza1 tiene pieza y se puede ensamblar en ensamble2, entonces la ensambla:
 if( (bas.puedo  ens(&pinza1,ens1)   neg) && (pinza1.conte()!  vacio)
      && (bas puedo  ens(&pinza1,ens2)!  neg) )
 {
      r1.ensamblar(pinza1,ens2);
      bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,pinza3,b2  s);
 }

Si no se puede ensamblar en ensamble1 o en ensamble2, hay espacio
en almacen y pinza1 tiene pieza,la almacena:
 if( ((bas.puedo  ens(&pinza1,ens1)   neg)  (bas.puedo  ens(&pinza1,ens2)   neg))
      && (bas.hay  espacio(alm)!  neg)
      && (pinza1.conte()!  vacio) )
      {
       r1.almacenar(pinza1,alm);   Almacena lo que tiene pinza1 en alm
       bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,pinza3,b2  s);
      }

 Si en el ALMACEN existe una pieza que se puede ensamblar,
 la toma y la ensambla·
 Si PosAlm     neg no se toma nada del ALMACEN
 Si PosAlm !   neg si se toma pieza del ALMACEN

 if( (PosAlm  bas.puedo  ens(alm,ens1,1)) !   neg )
  {
       r1.tomar(alm[PosAlm], pinza1);
       r1.ensamblar(pinza1, ens1);
       bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,pinza3,b2  s);
  }

  Si en el ALMACEN existe una pieza que se pueda ensamblar en ENS2, la
  toma y la ensambla:

 if( (PosAlm  bas.puedo  ens(alm,ens2,1)) !   neg)
  {
       r1.tomar(alm[PosAlm], pinza1);
       r1.ensamblar(pinza1, ens2);
       bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,pinza3,b2  s);
  }

  *                                                SUBTAREA3:
 ----------------------------------------------------------------------------------------------------------------------*/


  Si existe un producto terminado en la mesa de Ensamble1
  y la Pinza2 está vacia, entonces el Robot2 toma la pieza:

 if( ((PosTerm  bas hay_pdcto(ens1)) !   neg) && (pinza2.conte()-=vacio) )
```

```
 {
      r2.tomar(ens1[PosTerm],pinza2,1); // Toma el producto terminado
      bas.muestra(b1_e,pinza1,ens1,ens2,alm,pinza2,pinza3,b2_s);
 }

  Si Pinza2 tiene pieza y Banda2 está vacia, entonces la pone en Banda2:
 if( (pinza2.conte()!_vacio) && (b2_s.conte()_vacio) )
 {
  r2.poner(pinza2,b2_s,1),
  bas.muestra(b1_e,pinza1,ens1,ens2,alm,pinza2,pinza3,b2_s);
 }

 *                                          SUBTAREA4:
 ----------------------------------------------------------------------------------------------------*/

  Si existe un producto terminado en la mesa ENS2 y si la Pinza3
  esta vacia, entonces el Robot2 toma la pieza:

 if( ((PosTerm_bas.hay_pdcto(ens2))!_neg) && (pinza3.conte()_vacio) )
 {
      r3.tomar(ens2[PosTerm],pinza3,1);    Toma el producto terminado
      bas.muestra(b1_e,pinza1,ens1,ens2,alm,pinza2,pinza3,b2_s);
 }


  Si Pinza3 tiene pieza y Banda2 está vacia, entonces pone la pieza en Banda2:
 if( (pinza3.conte()!_vacio) && (b2_s.conte()_vacio) )
 {
  r3.poner(pinza3,b2_s,1);
  bas.muestra(b1_e,pinza1,ens1,ens2,alm,pinza2,pinza3,b2_s);
 }

 *                                          SUBTAREA5:
 ----------------------------------------------------------------------------------------------------*

 Si Banda2 tiene una pieza entonces avanza:
 if( b2_s.conte()!_vacio)
 {
  b2.avanzar(b2_s);
  bas.muestra(b1_e,pinza1,ens1,ens2,alm,pinza2,pinza3,b2_s);
 }

}
while(pieza!_'S');
getch();
return(0);
}
```

APÉNDICE C

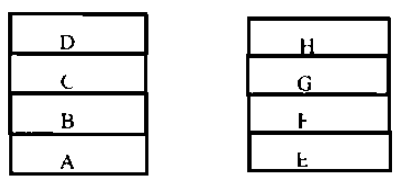
CODIFICACIÓN DEL EJEMPLO 2

```
/*
 EJEMPLO2.CPP          Realizó: Ing. Carlos Alonso Camacho Ramírez

SISTEMA DE ENSAMBLE COMPUESTO POR:

        1 Banda de entrada (b1  e )
        1 Sensor de visión (c1)
        1 Robot (r1) que almacena y/o ensambla
        1 Mesa de almacenamiento con 4 lugares ( alm[4] )
        2 Mesas de ensamble con 2 lugares c u  ( ens1[2], ens2[2] )
        1 Robot (r2) que toma el P.T. de las mesas de ensamble y lo pone en la bandas de salida
        1 Banda transportadora de salida (b2  s) por donde sale el producto ensamblado en ens1.
        1 Banda transportadora de salida (b3  s) por donde sale el producto ensamblado en ens2.

LOS ENSAMBLES SON DE LA SIGUIENTE MANERA:
```




```
*

#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<dos.h>
#include<iomanip.h>

const
   maxens  2,maxalm  4, minpza  'A', maxpza='H', vacio='_', neg=maxpza+1,
   maxpza1  'D', minpza2  'E';

enum
        bool{no,si};

bool pieza  valida(char pza);      Función que checa si la pieza es válida

int pieza1  2(char pza);    Checa si la pieza corresponde a Ensamble 1 o 2

void esp(void);             // Función que espera 2 segundos

class celula                Celula de ensamble
{
 protected:
   char contenido;          / Partes a ensamblar
   char producto;           / Productos ensamblados
};
```

```
class sitio:private celula      / Lugar donde las partes quedan estables
{
 protected:
    char nombre[12];
 public:

   sitio()                  Constructor sin parametros
    {
          contenido  vacio;
    }

    sitio(char n[12])          Constructor con 1 parametro
    {
          contenido  vacio;
          strcpy(nombre,n);
    }

    sitio(char n[12],char pieza)     Constructor con 2 parametros
    {
          contenido  pieza;
          strcpy(nombre,n);
    }

     sitio(void){ };              Destructor

    char conte() {  return contenido;  }

    char prod()  {  return producto;   }

    char* nom() {  return nombre;   }

    bool esta  vacio(sitio sit);     Retorna SI, si el sitio está vacio

    int hay  espacio(sitio ar[]);  Si hay espacio en ar[] retorna
                                    la posición, si no retorna neg

    int puedo  ens(sitio de[], sitio ar[],bool band);       Si se puede ensamblar en ar[],retorna
                                                          / la posición, si no retorna neg

    int hay_pdcto(sitio ar[]);    Checa si existe pdcto.terminado en mesa ensamble

    inter(sitio &a, sitio &b);   / Pone el contenido de a en b, y en a pone vacio

    bool hay  pieza();           / Pregunta si hay o no pieza en la banda de entrada

    char ident_pieza();           Identifica la pieza que llega (Pregunta que pieza es)

    void establece(char valor) { contenido  valor; }

   void sitio::muestra(sitio ban1,sitio pinz1,sitio ens1[],sitio ens2[],sitio alm[],sitio pinz2,sitio ban2,
                       sitio band3);
};  / Termina clase sitio
```

```
bool sitio::esta  vacio(sitio sit)
{
  if(sit.contenido    vacio)  return si;
  else return no;
}

int sitio::hay  espacio(sitio ar[] )    Retorna la posición i del arreglo si hay espacio en él , si no retorna neg
{
 for(int i  0; i<maxalm; i++)
 {
    if(esta  vacio(ar[i]))  return(i);
 }
 return(neg);
}

sitio::puedo  ens(sitio de[], sitio ar[], bool band  0)
{
  * Checa si la pieza a ensamblar es la que sigue, o bien, si la pieza es la menor y existe una posición vacia
en
     ar[i]. Si no se cumple ninguna de las 2 condiciones anteriores, entonces no se puede ensamblar:   */

 if( band    1)     Si band  1, checa si una pieza del almacen puede ensamblarse
 {
  char menor=vacio;
  for(int j  0; j<maxalm; j++)
  {
   if( pieza1  2(de[j].conte())    1)  menor=minpza;
   else if (pieza1  2(de[j].conte())    2)
          menor=minpza2;
   for(int i  0; i<maxens; i++)
   {
         if( (de[j] conte()-ar[i] conte())    1 )
            return j;
         if( (de[j].conte()    menor) &&  (esta  vacio(ar[i])) )
            return j;
   }
  }
 }
 else    Si band  0, checa si la pieza de.conte() puede ensamblarse
 {
     Si se quiere ensamblar en ENS1 una pieza que es de ENS2, entonces no se puede:

  if( ((!strcmp( ar[0].nom() ,"Ensamble1,1")) && (pieza1  2(de[0].conte())    2)) )
   return(neg);

     Si se quiere ensamblar en ENS2 una pieza que es de ENS1, entonces no se puede:

  if( ((!strcmp( ar[0].nom() ,"Ensamble2,1")) && (pieza1  2(de[0].conte())    1)) )
   return(neg),

   char menor=vacio;
   if( pieza1  2(de[0].conte())    1)  menor=minpza;
   else if (pieza1  2(de[0].conte())    2)  menor=minpza2;
```

```
   for(int i  0; i<maxens; i++)
   {
    if( (de[0].conte()-ar[i].conte())    1 )
          return i;
    if( (esta  vacio(ar[i])) && (de[0].conte()    menor) )
          return i;
   }
 }
 return (neg);
}

int sitio::hay_pdcto(sitio ar[])  Checa si existe pdcto.terminado en ar[]
{
  for(int i  0; i<maxens; i++)
  {
   if( pieza1  2(ar[i].conte())    1 && (ar[i].conte()  -maxpza1) )
          return i;
   else if( pieza1  2(ar[i].conte())    2 && (ar[i].conte()  -maxpza) )
          return i;
  }
  return neg;
}

sitio::inter(sitio &a, sitio &b)
{
   b.contenido  a.contenido;
   a.contenido  vacio;
}

class efector:private celula
{
  char tipo[10];
  char funcion[15];
  int localizacion;
};

class robot:private efector, private sitio
{
 public:
   robot(char n[10]) : sitio(n) { }   Usa el constructor de la clase padre sitio
    robot(void){ };
   bool tomar(sitio &de, sitio &conque, bool band);
   bool poner(sitio &de, sitio &donde, bool band);
   bool ensamblar(sitio &de, sitio arr[]);
   bool almacenar(sitio &de, sitio arr[]);
};

bool robot::tomar(sitio &de, sitio &conque, bool band  0)
{
   if (esta  vacio(de)) {
         cout << "El Robot no puede tomar, ya que ";
         cout << de.nom() << " tiene ";
         cout << de.conte();
         return(no);
   }
```

```
    if (!esta_vacio(conque))
    {
          cout << conque.nom() << " esta ocupado, tiene: ";
          cout << conque.conte();
          return(no);
    }

    if (band! 0)
          cout << "\nSe está tomando el producto terminado";
    else
          cout <<. "\nSe está tomando la pieza " << de.conte();

    cout << " de " << de.nom();
    inter(de,conque);
    esp();
    return(si);
}

bool robot::poner(sitio &de, sitio &donde, bool band 0)
{
    if(esta_vacio(de))
          return no;
    if (!esta_vacio(donde))
    {
          cout << "El Robot no puede poner, porque ahí esta: ";
          cout << donde.conte(),
          return(no);
    }
    if (band! 0)
          cout << "\nSe está poniendo el producto terminado";
    else
          cout << "\nSe está poniendo la pieza " << de.conte();
    cout << " en " << donde.nom();
    inter(de,donde);
    esp();
    return(si);
}

bool robot::ensamblar(sitio &de, sitio arre[])
{
  bool exito no;
  sitio dee[1];
  dee[0] de;
  int i-puedo_ens(dee,arre);
  if( i!-neg )
  {
          cout << "\nSe está ensamblando " << de.conte() << " de " << de.nom();
          cout << " en " << arre[i].nom();
          inter(de, arre[i]);
          esp();
          exito si;
  }
  return(exito);
}
```

```
bool robot::almacenar(sitio &de, sitio arr[] )
{
  if(de.conte()    vacio)   return no;
  int j;
  bool exito  no;
  j  hay  espacio(arr);
  if(j !  neg) {
     cout << "\nSe está almacenando la pieza ";
     cout << de.conte();
     cout << " en la posicion " << j+1 << " de la mesa";
     inter(de,arr[j]);
     exito  si;
     esp();
   }
   else  cout << "\nEstá ocupada la mesa de " << arr[0].nom(),    No se pudo almacenar
   return(exito);
}

class banda:private sitio,private efector
{
 public:
    avanzar(sitio &sit);   / Avanza la banda transportadora
    alimentar();
};

bool sitio::hay_pieza()
{
   char llega;
   cout << "\nLLEGA PIEZA (S/N)?: ";
   cin >> llega;
   if(llega    'S')   return si;
   else  return no;
}

banda::avanzar(sitio &sit)
{
   sitio basu;
   cout << "\nLa " << sit.nom() << " está avanzando";
   inter(sit,basu);
   esp();
}

banda::alimentar()
{
   cout << "La pieza esta entrando por Banda1";
   esp();
}

char sitio::ident_pieza()
{
   char pieza;
   cout << "\nQue pieza llega(S para Salir)? ";
   cin >> pieza;
   return pieza;
}
```

```
class sensor:private celula, private sitio
{
 public:
   char ident_pieza();
};

  Muestra el contenido de los sitios:
void sitio::muestra(sitio ban1,sitio pinz1,sitio ens1[],sitio ens2[],
                    sitio alm[],sitio pinz2,sitio ban2, sitio ban3)
{
  cout << "\nESTADO DE LA CELULA: ";
  cout << ban1.nom() << ":" << ban1.conte() << flush;
  cout << ", " << pinz1.nom() << ":" << pinz1.conte() << flush;
  cout << ", Ensamble1: ";
  for(int i  0; i<maxens; i++)
        cout << ens1[i].conte() << ", " << flush;
  cout << "Ensamble2: ";
  for(i  0; i<maxens; i++)
        cout << ens2[i].conte() << ", " << flush;
  cout << "\n Almacen: ";
  for(i  0; i<maxalm; i++)
        cout << alm[i].conte() << ", " << flush;
  cout << pinz2 nom() << ":" << pinz2.conte() << flush,
  cout << ", " << ban2.nom() << ":" << ban2.conte() << flush;
  cout << ", " << ban3.nom() << ":" << ban3.conte() << "\n" << flush;
}

bool pieza  valida(char pza)       Checa si la pieza es valida
{
 if ( (pza>  minpza) && (pza<  maxpza) )  return(si);
 else  return(no);
}

int pieza1  2(char pza)      Retorna 1 si la pieza es del ensamble 1 y 2 si es del ensamble 2
{
  if ( (pza>  minpza) && (pza<  maxpza1) )    Si es una pieza del Ensamble1
        return(1);
 else   Si es una pieza del Ensamble2
        return(2);
}

void esp(void)    Función que espera 2 segundos
{
 delay(2000);
}

main()
{
  clrscr();

    Inicia definición e inicializacion:
  char pieza;  banda b1,b2,b3;
  sensor sens1;
  sitio b1  e("Banda1"), b2  s("Banda2"), b3  s("Banda3"), bas;
  sitio ens1[2]  {"Ensamble1,1","Ensamble1,2"}, ens2[2]-{"Ensamble2,1","Ensamble2,2"};
```

```
sitio alm[4] {"Almacen.1","Almacen.2","Almacen.3","Almacen.4"};
sitio pinza1("Pinza1"), pinza2("Pinza2");
robot r1("Robot1"), r2("Robot2");

/ Muestra el estado de la celula:
bas.muestra(b1_e,pinza1,ens1,ens2,alm,pinza2,b2_s,b3_s);

do
{

 pieza vacio; / Inicialmente pieza esta vacio
 sitio sens1("Sensor1",vacio); / Inicializa el Sensor1 con vacio
 int PosAlm_neg;     Posicion en el ALMACEN de la pieza
 int PosTerm_neg; // Posicion en mesa de ensamble del producto terminado


 *                                        SUBTAREA1:
 ----------------------------------------------------------------------------------------*/

  Si existe pieza, es alimentada por la Banda1 e inicializada con '?'

 if(b1_e.hay_pieza()== si)
 {
       b1.alimentar();
       b1_e.establece('?');  Pone en Banda1 la pieza desconocida '?'
 }

  Si el Sensor1 esta vacio y Banda1 tiene pieza, la identifica:

 if(sens1.conte()==vacio && b1_e.conte()=='?')
 {
  pieza b1_e.ident_pieza();  Captura la pieza para identificarla
  b1_e.establece(pieza);
 }

 * Si no es pieza valida o no se puede ensamblar ni en ENS1 ni en ENS2
  y no hay espacio en ALMACEN, entonces la pieza no es procesable  */

 if( (!pieza_valida(b1_e.conte())) |
       ( (bas.puedo_ens(&b1_e,ens1)   neg) &&
        (bas.puedo_ens(&b1_e,ens2)==neg) &&
        (bas.hay_espacio(alm)   neg) )
  )
 {
       cout << "\nPieza de " << b1_e.nom() << " no procesable";
       b1.avanzar(b1_e);
       bas.muestra(b1_e,pinza1,ens1,ens2,alm,pinza2,b2_s,b3_s);
 }
```

```
*                                    SUBTAREA2:
-------------------------------------------------------------------------------------------
Si Banda1 tiene pieza, esta pieza es valida, la puedo ensamblar en  ENS1 o en ENS2, o bien hay espacio
para almacenarla, y Pinza1 está vacia, entonces la toma:          */

if( (b1  e.conte()!  vacio)
      && (pieza  valida(b1  e.conte()))
      && ( (bas.puedo  ens(&b1  e,ens1)!  neg) || (bas.puedo  ens(&b1  e,ens2)!  neg)
        (bas.hay  espacio(alm)!  neg) )
      && (pinza1.conte()    vacio)
 )
 {
       r1.tomar(b1  e,pinza1);  /  Toma la pieza de banda1 con pinza1
       bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,b2  s,b3  s);
 }

 Si Pinza1 tiene pieza, es de ENS1 y se puede ensamblar, la ensambla:
if ( (pinza1.conte()!  vacio) && (pieza1  2(pinza1.conte())==1) &&
      (bas.puedo  ens(&pinza1,ens1)!  neg)
 )
 {
      r1.ensamblar(pinza1,ens1);
      bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,b2  s,b3  s);
 }

  Si Pinza1 tiene pieza, es de ENS2 y se puede ensamblar, la ensambla:
if( (pinza1 conte()!  vacio) && (pieza1  2(pinza1.conte())    2) &&
      (bas.puedo  ens(&pinza1,ens2)!  neg)
 )
 {
      r1.ensamblar(pinza1,ens2);
      bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,b2  s,b3  s);
 }

  Si PINZA1 tiene pieza, no se puede ensamblar en ENS1 o en ENS2, y hay espacio en ALMACEN,
  entonces la almacena:

if( ((bas.puedo  ens(&pinza1,ens1)=  neg)   (bas.puedo  ens(&pinza1,ens2)=  neg))
      && (bas.hay  espacio(alm)!  neg)
      && (pinza1.conte()!  vacio)
 )
      {
        r1.almacenar(pinza1,alm);    Almacena lo que tiene pinza1 en alm
        bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,b2_s,b3  s);
      }

  Si en el ALMACEN existe una pieza que se pueda ensamblar en ENS1, la toma y la ensambla:
 if( ((PosAlm  bas.puedo  ens(alm,ens1,1)) !=neg) && (pieza1  2(alm[PosAlm].conte())  =1) )
 {
       r1.tomar(alm[PosAlm], pinza1); /  Toma la pza del ALMACEN con PINZA1
       r1.ensamblar(pinza1, ens1);
       bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,b2  s,b3  s);
 }
```

```
// Si en el ALMACEN existe una pieza que se pueda ensamblar en ENS2, la toma y la ensambla:

if( ((PosAlm  bas.puedo  ens(alm,ens2,1))!  neg) && (pieza1  2(alm[PosAlm].conte())  -2) )
{
      r1.tomar(alm[PosAlm], pinza1); // Toma la pza del ALMACEN con PINZA1
      r1.ensamblar(pinza1, ens2);
      bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,b2  s,b3  s);
}

/*                                                    SUBTAREA3:
-------------------------------------------------------------------------------------------------------------*/

/ Si existe un P.T. en la mesa ENS1 y si la Pinza2 está vacia, entonces el Robot2  toma la pieza:

if( ((PosTerm  bas.hay_pdcto(ens1)) !  neg) && (pinza2.conte()   vacio) )
{
      r2.tomar(ens1[PosTerm],pinza2,1);  // Toma el producto terminado
      bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,b2  s,b3  s);
}

Si existe un P.T. en la mesa ENS2 y si la Pinza2 está vacia, entonces el ROBOT2 toma la pieza:

if( ((PosTerm  bas.hay_pdcto(ens2)) !  neg) && (pinza2.conte()==vacio) )
{
      r2.tomar(ens2[PosTerm],pinza2,1);  // Toma el producto terminado
      bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,b2_s,b3  s);
}

  Si PINZA2 tiene pieza y BANDA2 está vacia, entonces la pone en BANDA2

if( (pinza2.conte()!  vacio) && (b2  s.conte()==vacio) && (pieza1_2(pinza2.conte())==1) )
{
 r2.poner(pinza2,b2  s,1);
 bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,b2_s,b3  s);
}

  Si PINZA2 tiene pieza y BANDA3 está vacia, entonces la pone en BANDA:
if( (pinza2.conte()!  vacio) && (b3  s.conte()=  vacio) && (pieza1_2(pinza2.conte())==2)  )
{
 r2.poner(pinza2,b3  s,1);
 bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,b2_s,b3  s);
}

/*                                                    SUBTAREA4:
-------------------------------------------------------------------------------------------------------------*/

/ Si BANDA2 tienen una producto terminado entonces avanza:

if( (b2  s.conte()!  vacio) )
{
  b2.avanzar(b2_s);
  bas.muestra(b1  e,pinza1,ens1,ens2,alm,pinza2,b2_s,b3  s);
}
```

```
 /*                                        SUBTAREA5:
 ----------------------------------------------------------------------------------------------------------------*/

 / Si BANDA3 tienen una producto terminado entonces avanza:

 if( (b3_s.conte()!_vacio) )
 {
   b3.avanzar(b3_s);
   bas.muestra(b1_e,pinza1,ens1,ens2,alm,pinza2,b2_s,b3_s);
 }

}
while(pieza!_'S');
getch();
return(0);
}
```

REFERENCIAS

[1] Dennis de Champeaux , Douglas Lea, and Penelope Faure. " Object-Oriented System Development", 1993, Addison Wesley.

[2] James Martin y James J. Odell. "Métodos orientados a objetos, conceptos fundamentales", 1997, Prentice Hall.

[3] Hebert Schildt. "C++, Manual de referencia" , 1995 , McGraw Hill.

[4] Mikell P. Groover, Mitchell Weiss, roger N. Nagel y Nicholas G. Odrey. "ROBOTICA INDUSTRIAL, Tecnología, Programación y Aplicaciones", 1990, McGraw Hill.

[5] J.P. Sáanchez y Beltrán. "SISTEMAS EXPERTOS, una metodología de programación", 1990, Macrobit.

[6] J. Ma. Angulo y A. del Moral. "Guía fácil: INTELIGENCIA ARTIFICIAL", 1994, PARANINFO.

[7] K. y S. Brain. "INTELIGENCIA ARTIFICIAL EN EL DRAGON", 1985, Editorial Gustavo Gili.

[8] Arjona-Suárez, E. and López-Mellado, E.. "A computer Language for the Modelling of Flexible Manufacturing Systems", in Proceedings of the *13th. IASTED International Symposium on Robotics and Manufacturing*. Santa Barbara, CA, Nov. 1990: 183-187.

[9] Bakalem, M., G. Habchi and A. Courtois. "PPS: An Integrated Object Oriented Approach for Modelling and Simulation of Manufacturing Systems", *IEEE International Conference on Systems, Man and Cybernetics*. San Antonio, Texas, USA October 1994.

[10]Ramzi G., Bakalem M. and G. Habchi. "An Object Model for Simulation of Manufacturing Systems", IEEE International Conference on Systems, Man and Cybernetics. Vancouver, Canada. October 1995.

[11]López-Mellado, E.. "Design of Knowledge-Based Real-Time Controllers for Assembly Systems", *IEEE International Conference on Systems, Man and Cybernetics*. Vancouver, Canada. October 1995.

[12] Chochon, H. Alami, R.. "A Knowledge-based system for programming and execution control of multi-robot assembly cells", *International Conference on Advanced Robotics*.Versailles (France), October 1987.

[13] López-Mellado, E.. "Object-based design of FMS controllers", in Proceedings of the Fifth *IASTED International Conference Robotics and Manufacturing.* May 29-31, 1997 - Cancún, México.

[14] Jackson J. R.. "Jobshop-Like Queue Systems", Management Science, 1963.

[15] Jafari M.. "Performance Modelling of a Flexible Manufacturing Cell with Two Workstations and a Single Material Handling Device", Proceedings of the IEEE Internacional Conference on Robotics and Automation. Raleigh, North Carolina, 1987.

[16] Yao D. D. , Buzacott J. A.. "Queueing Models for a Flexible Machining Station", International Journal of Prod. Res.". 1985.

[17] Chu Zhou M., Di Cesare F.. " A Petri Net Design Method for Automated Manufacturing Systems with Shared Resources", Proceedings of the IEEE International Conference on Robotics and Automation. 1990.

[18] Narahari Y., Viswandham N.. " Analysis and Synthesis of Flexible Manufacturing Systems". Ann Arbor, Michigan, 1984.

[19] Greg Perry, "Aprendiendo Programación Orientada a Objetos con TURBO C++ en 21 días", Prentice Hall, 1995.

[20] Sharam Hekmatpour, "C++ GUIA PARA PROGRAMADORES EN C", Prentice Hall, 1992.

[21] H. Paul Haiduk, "TURBO PASCAL ORIENTADO A OBJETOS", McGraw Hill, 1994.

[22] Luis Joyanes Aguilar, "TURBO C++ INICIACION Y REFERENCIA", McGraw Hill, 1996.

[23] Richard S. Wiener y Lewis J. Pinson, "An introduction to Object-Oriented Programming and C++", Addison -Wesley, 1988.

[24] Elaine Rich, Kevin Knight, "INTELIGENCIA ARTIFICIAL", 2a. Edición, Mc Graw Hill, 1994.

[25]Patrick Henry Winston, " ARTIFICIAL INTELLIGENCE", Third edition, ADDISON WESLEY, 1992.

[26]David W. Rolston, " Principios de Inteligencia Artificial y Sistemas Expertos ", Mc Graw Hill, 1992.

[27]David D. Bedworth, Mark R. Henderson, Philip M. Wolfe, " COMPUTER - INTEGRATED DESIGN AND MANUFACTURING ", McGRAW-HILL INTERNATIONAL EDITIONS, 1991.

RESUMEN AUTOBIOGRÁFICO

Carlos Alonso Camacho Ramírez

Candidato para el Grado de

Maestro en Ciencias de la Administración con Especialidad en Sistemas

Tesis: COORDINACIÓN DE ACTIVIDADES EN SISTEMAS DE MANUFACTURA FLEXIBLES.

Campo de Estudio: Sistemas

Biografía:

Nacido en Colonia Anahuac, Chihuahua, el 14 de Diciembre de 1968, hijo de Hilaria Ramírez Romero y Adolfo Camacho Camacho.

Educación:

Egresado del Instituto Tecnológico de Chihuahua II; obteniendo el titulo de Ingeniero en Sistemas Computacionales en 1991, por la opción VIII (promedio).

Experiencia Profesional:

Docente de las materias para las carreras de Licenciatura en Informática e Ingeniería en Sistemas Computacionales del Instituto Tecnológico de Ciudad Cuauhtémoc, Chihuahua.