UNIVERSIDAD DE BELGRANO

Las tesinas de Belgrano

Facultad de Tecnología Informática
Ingeniería Informática

Construcción de aplicaciones distribuidas aplicando el enfoque MIC

Nº 260 **Rodrigo Catalano Tchouhadjian**

Tutor: Gustavo Dejean

Departamento de Investigaciones
Junio 2009

Agradecimientos

A Myriam, Micaela y Juan Pablo, mi esposa e hijos que me han acompañado y ofrecido su amor a lo largo de este camino que emprendí y me dieron de su tiempo para poder recorrerlo.
A mi tutor Gustavo Dejean por su tiempo, críticas, consejos y compromiso brindado para la terminación de esta tesina.

Índice

II. Definiciones, Acrónimos y Abreviaciones

.NET: plataforma creada por Microsoft para el desarrollo rápido de aplicaciones empresariales distribuidas orientas a componentes y servicios.

API: acrónimo en inglés de Application Programming Interface, Interfaz de Programación de Aplicaciones.

BGML: acrónimo en inglés de Benchmark Generation Modeling Language, Lenguaje de Modelado de Generación de Evaluación de Prestaciones.

CCM: acrónimo en inglés de CORBA Component Model, Modelo de Componentes CORBA.

CCM Perf: herramienta CCM para describir el despliegue de los componentes de una aplicación CCM.

CIDL: acrónimo en inglés de Component Implementation Definition Language, Lenguaje de Definición de Implementación de Componentes.

CIF: acrónimo en inglés de Component Interface Framework, Marco de Trabajo de Implementación de Componentes.

Components Middleware: del inglés Middleware orientado a Componentes, ver capítulo 2.1.6.

CORBA: acrónimo en inglés de Common Object Request Broker Architecture, Arquitectura Común de Agentes de Peticiones de Objeto. Es un estándar que establece una plataforma de desarrollo de sistemas distribuidos facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos. Es independiente del lenguaje de implementación ya que define un lenguaje propio para la definición de las interfaces de los objetos.

CoSMIC: acrónimo en inglés de Component Synthesis using Model Integrated Computing, síntesis de componentes utilizando la computación integrada por modelos.

DAnCE: acrónimo en inglés de Deployment and Configuration Engine, Motor de Despliegue y Configuración.

DOC: acrónimo en inglés de Distributed Object Computing, Computación de Objetos Distribuida.

DSML: acrónimo en inglés de Domain Specific Modeling Language, Lenguaje de Modelado para Dominios Específicos.

FOC: acrónimo en inglés de First Class Object: Primer Clase de Objeto, tipo de elemento del meta-metalenguaje del GME..

GME: acrónimo en inglés de Generic Modeling Environment, Entorno de Modelado Genérico.

Grid Computing: del inglés Computación en Grilla, ver capítulo 2.1.6.

IDL: acrónimo en inglés de Interface Definition Language, Lenguaje de Definición de Interfaces, es un lenguaje que permite la definición de interfaces de componentes CORBA.

IDML: acrónimo en inglés de Interface Definition Modeling Language, lenguaje de modelado para la definición de interfaces.
Ingeniería de Software: es la aplicación de un enfoque sistémico, disciplinado y cuantificable para el desarrollo, operación y mantenimiento de software; esto es la aplicación de la ingeniería de software. (Definición según la IEEE)
J2EE: acrónimo en inglés de Java 2 Enterprise Edition, es la edición empresarial de las tecnologías, interfaces y herramientas de desarrollo de Java. Comprenden un conjunto de especificaciones y funcionalidades orientadas al desarrollo de aplicaciones empresariales.
MIC: acrónimo en inglés de Model Integrated Computing, Computación Integrada por Modelos.
MIDCESS: acrónimo en inglés de Model Integrated Deployment and Configuration Environment for Composable Software Systems, entorno integrado para el modelado del despliegue y la configuración para sistemas de software compuestos.
MODEM: es un acrónimo del término MOdulador-DEModulador; es decir, que es un dispositivo que transforma las señales digitales de la computadora en señal telefónica analógica y viceversa, con lo que permite a la computadora transmitir y recibir información por la línea telefónica.
OCL: acrónimo en inglés de Object Constrain Language, Lenguaje de Restricciones de Objetos. Es un lenguaje que permite especificar restricciones sobre los elementos de los diagramas UML.
OCML: acrónimo en inglés de Options Configuration Modeling Language, Lenguaje de Modelado de Configuración de Opciones.
OMG: acrónimo en inglés de Object Management Group, Grupo de Administración de Objetos. Es un consorcio dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como UML, XMI, CORBA.
On-Line: modo de operación de una aplicación en donde el dato introducido directamente en el punto de entrada es procesado en el punto en donde es usado. (IEEE 620.12-1990).
O.O.: acrónimo de Orientado a Objetos.
ORB: acrónimo en inglés de Object Request Broker, Agente de Petición de Objetos.
QoS: acrónimo en inglés de Quality Of Service, Calidad del Servicio.
PICML: acrónimo en inglés de Platform Independent Component Modeling Language, lenguaje de modelado de componentes independientes de la plataforma.
POS: acrónimo en inglés de Point Of Sale, Punto de Venta.
RPC: acrónimo en inglés de Remote Procedure Call, llamada a un procedimiento remoto. Es una infraestructura cliente / servidor que incrementa la interoperatividad, portabilidad y flexibilidad de una aplicación permitiendo que ésta pueda ser distribuida a lo largo de múltiples plataformas heterogéneas[1].
SBC: Sistemas Basados en Computadoras.
SOAP: acrónimo en inglés de Simple Object Access Protocol, protocolo estándar que define el intercambio de objetos a través de mensajes XML.
UML: acrónimo en inglés de Unified Modeling Language, Lenguaje de Modelado Unificado, es un lenguaje unificado para el modelado de aplicaciones.

1. Definición obtenida desde el Instituto de Ingeniería de Software, Carnegie Mellon, EEUU. http://www.sei.cmu.edu/str/descriptions/rpc.html

1. Introducción

1.1. Resumen

Uno de los principales temas de investigación dentro de la disciplina de la Ingeniería de Software es el estudio de: "la generación automática de código a través de la definición de modelos de alto nivel". Dentro de esta problemática aparece otra que se desprende de esta ultima: "la construcción automática de software a partir de las definiciones de un modelo específico del dominio del problema". Uno de los enfoques que soporta esta característica, es el MIC, el cual incluye modelos de análisis de dominios específicos y herramientas de síntesis de modelos. La idea que encierra el MIC es la siguiente: en vez de que un programador este constantemente transformando el conocimiento del modelo de un dominio específico en modelos de bajo nivel (diagramas de clases[2] y código ejecutable[3] por ejemplo), primero se construye un "lenguaje" que permite modelar los conceptos del dominio específico y luego se desarrolla un traductor que transforma el modelo creado a partir del lenguaje a código ejecutable para una plataforma[4] de ejecución determinada. Este enfoque cede la responsabilidad de la creación del modelo del software a un especialista en el dominio específico del problema, dejando al desarrollador la responsabilidad de la construcción del traductor[5]. Para comprobar el desarrollo de aplicaciones distribuidas utilizando el enfoque MIC se eligió como caso de estudio el modelado de una aplicación del mundo real, la cual está actualmente implementada a través del proceso de desarrollo de software orientado a objetos tradicional[6].

1.2. Objetivo

El presente trabajo final de carrera tiene como objetivo presentar la Computación Integrada por Modelos, elegir una herramienta MIC (el GME) y un lenguaje del dominio específico de las aplicaciones distribuidas, utilizar el lenguaje para crear un modelo, realizar la síntesis del modelo utilizando los traductores proporcionados por CoSMIC. Y por último analizar las diferencias entre el desarrollo orientado a objetos tradicional frente al enfoque del MIC.

El modelo a presentar en el caso de estudio es una solución para la automatización del proceso operativo de una planta de elevación de granos[7]. Dicho modelo será especialmente creado para la comprobación del desarrollo de aplicaciones distribuidas utilizando el enfoque MIC para el presente trabajo.

1.2.1. Objetivo Secundario

Como objetivo secundario se establece la utilización solo de herramientas de código abierto[8].

1.3. Alcance

El trabajo comprende:

- Una introducción a los sistemas distribuidos y a un Middleware orientado a componentes.
- Una descripción del MIC y de herramientas que lo soportan.
- La construcción de un modelo de una aplicación distribuida.
- El análisis de las diferencias entre el desarrollo utilizando el enfoque MIC con respecto al desarrollo orientado a objetos tradicional.

1.4. Límites

- No se abordará el estudio de los sistemas distribuidos ni del CCM, solo se hará una breve descripción.
- Solo se describirán los principales elementos de modelado del GME, dejándose de lado los aspectos del manual de usuario de la aplicación.
- Para la demostración del desarrollo de aplicaciones distribuidas utilizando el enfoque MIC se construyó el modelo de una aplicación distribuida y se ejecutaron los intérpretes de CoSMIC. No se implementó una aplicación en forma completa.

---

2. Diagramas de clase: Uno de los diagramas definidos por el lenguaje de modelado UML.
3. Código ejecutable es aquel código que puede ser interpretado por un compilador o alguna otra herramienta y convertido en un artefacto que puede ser ejecutado en alguna plataforma de ejecución de SW.
4. Plataforma: termino de carácter genérico que designa una arquitectura de software o de hardware o ambas.
5. [KAGLEDE2003], pág. 2.
6. Desarrollo de software orientado a objetos: aplicación del análisis y diseño orientado a objetos en la construcción de aplicaciones de software.
7. Planta de elevación de granos: planta en donde se descarga el cereal proveniente de los camiones y vagones y se carga en los buques para su exportación.
8. Para obtener una definición completa del código abierto (open source) ver la siguiente página: http://www.opensource.org/docs/definition.php

1.5. Organización del Trabajo

El trabajo esta organizado de la siguiente manera:
Capítulo 1: presenta el resumen, objetivo, alcance, límites y organización del presente trabajo.
Capítulo 2: consta del marco teórico que incluye en el capítulo 2.1 una introducción al concepto de los sistemas distribuidos y en el capítulo 2.2 se realiza una descripción detallada del enfoque MIC. Se incluye la definición de los DSMLs.
Capítulo 3: se realiza una descripción de la herramienta MIC denominada GME, la cual provee un entorno para el desarrollo de DSMLs.
Capítulo 4: consta de una introducción del CCM, específicamente de una implementación llamada CIAO.
Capítulo 5: presenta el conjunto de herramientas y DSMLs denominado CoSMIC.
Capítulo 6: creación del modelo de una aplicación distribuida para resolver un problema del mundo real. Finalmente se realizará un análisis de las diferencias entre el desarrollo utilizando en el presente trabajo con respecto al desarrollo orientado a objetos tradicional.
Capítulo 7: se presenta la conclusión y se analizan las posibles futuras líneas de investigación de este trabajo.
Capítulo 8: enumera la bibliografía utilizada en la tesina.
Capítulo 9: especifica el software utilizado en la tesina.
Capítulo 10: describe el contenido del CD que acompaña al trabajo.

2. Marco Teórico

2.1. Sistemas Distribuidos

Con la llegada de las redes de computadoras y la INTERNET muchos de los recursos utilizados por las aplicaciones de software no residen en una única computadora[9]. Por ejemplo en una aplicación de pago electrónico con tarjeta de crédito, la computadora que recibe el pago, se conecta con otra que realiza las primeras validaciones y determina a que Centro Autorizador debe enviar la información, el cual a su vez tiene otra/s computadoras que autorizan la transacción y devuelven la respuesta, que finalmente llega a la computadora que capturo el pago. Esta disposición es lo que denominamos un sistema distribuido. Originalmente los procesos distribuidos se realizaban en forma diferida (batch), con el paso del tiempo y el abaratamiento de las redes de telecomunicaciones los procesos se empezaron a realizar en tiempo real. Esta necesidad, llevo a crear una serie de servicios de red heterogéneos para manejar la comunicación entre procesos y aplicaciones distribuidas. Uno de los primeros servicios fue el RPC, el cual solo soportaba la interacción a través llamadas a procedimientos remotos, con el advenimiento del paradigma de objetos, se creo CORBA, hoy estamos en pleno desarrollo del paradigma orientado a componentes y servicios, en donde las tecnologías más significativas son: SOAP, CCM, .NET y J2EE entre otros.

2.1.1. Conceptos

En la literatura podemos encontrar varias definiciones de ¿qué es? un sistema distribuido, es este trabajo utilizaremos la siguiente definición:

> "Un sistema distribuido es una colección de computadoras independientes que se comportan para el usuario del sistema como una única computadora".[10]

Esta definición presenta dos afirmaciones. La primera hace referencia al hardware: las computadoras son autónomas. La segunda hace referencia al software: el usuario piensa al sistema como una única computadora.

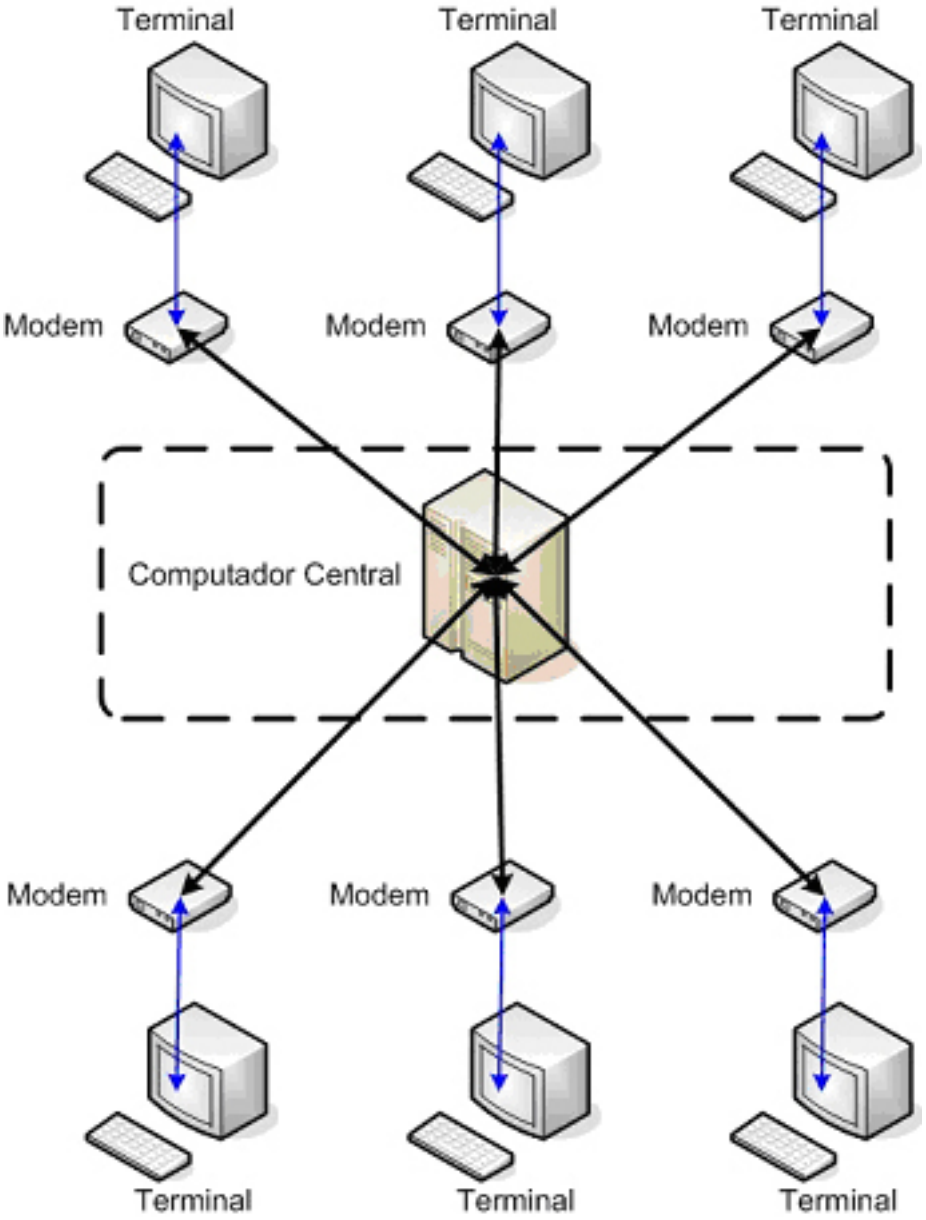
Un sistema distribuido es una oposición a los sistemas centralizados, los cuales consisten en una única computadora con uno o más unidades de proceso central que procesa todas las peticiones de los clientes. En síntesis, el software da la sensación de transparencia a la distribución del sistema.[11]

2.1.2. Arquitecturas

La definición e implementación de los sistemas distribuidos evolucionó desde los sistemas en donde los terminales remotos se comunicaban en forma independiente y diferida (modo batch) periódicamente

9. [KANN2004], capitulo 13, parte 3.
10. [TANEN2002], página 2.
11. TABU2001], página 3.

con los computadores centrales[12]. Es decir en esta época -entre 1970 y comienzos de 1980- hablar de sistemas distribuidos era hablar de procesamiento distribuido.



En la figura 1 se puede ver la arquitectura de los sistemas distribuidos de los mediados de 1970, en donde cada Terminal se comunicaba a través de un MODEM al computador central, debido a que el vínculo entre los equipos no era permanente el proceso se realizaba en forma diferida.

Al final de los años 80 los sistemas distribuidos experimentaron una evolución. Como se muestra en la figura 2, inicialmente los sistemas distribuidos contenían en cada sitio uno o múltiples componentes de software que proveían a los usuario el acceso a los recursos distribuidos. Luego la granularidad de la distribución del control se volvió mas fina, permitiendo que las funciones de un único componente de software se distribuyan a través de las redes. Hoy en día un sistema distribuido consiste de un conjunto de computadoras autónomas que están interconectadas a través de una o mas redes, en donde el software utilizado por estos sistemas se encuentra divido en múltiples componentes o servicios cada uno residiendo en diferentes sitios.[14]

12. Computador central: computadoras que procesan una gran cantidad de datos y soporta el acceso concurrente de un gran número de usuarios y sistemas.
13. [TABU2001], página 4.
14. [TABU2001], página 6.

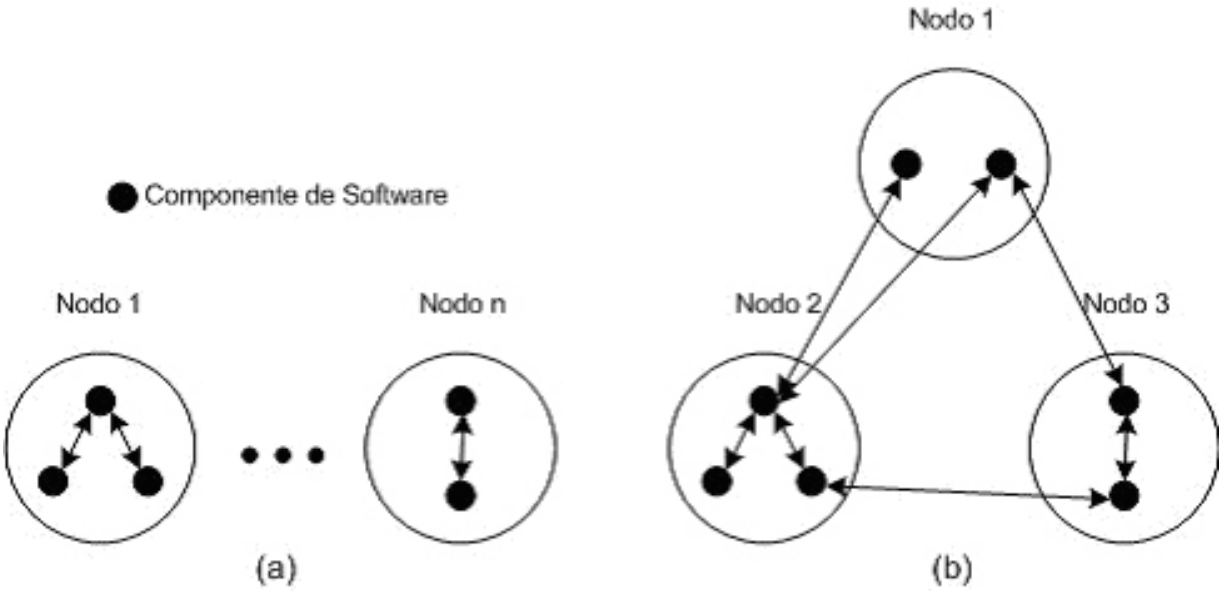


Fig. 2. (a) Sistema distribuido de principios de los 90's, (b) Sistema distribuido actual.[15]

2.1.3. Características Especiales de los Sistemas Distribuidos

Las características distintivas de los sistemas distribuidos se pueden resumir en las siguientes:[16]

- Concurrencia: los componentes de un sistema distribuido pueden ejecutarse al mismo tiempo.
- Modo de fallas independientes: los componentes de un sistema distribuido y la interconectividad entre las redes que lo componen pueden fallar en forma independiente.
- No existe un reloj único: se asume que cada componente del sistema tiene su propio reloj pero que los relojes no tienen la misma frecuencia, el sistema no garantiza que estén sincronizados. Esta característica es llamada *clock drift*[17].
- Demora en las comunicaciones: esto hace referencia al tiempo que tarda un evento desde la generación del mismo en el propio computador hasta la propagación del evento a todo el sistema.
- Estados inconsistentes: la concurrencia, las fallas independientes y la demora en las comunicaciones hacen que sea difícil mantener la consistencia del estado sobre los componentes del sistema distribuido.

2.1.4. El Middleware

Las mejoras dadas en el hardware y en las tecnologías referidas a las de redes de comunicaciones en la pasada década han producido una serie de avances importantes en las capacidades de las computadoras y las redes de comunicaciones. A pesar de estas mejoras el esfuerzo y costo requerido para desarrollar, validar, probar y desplegar las aplicaciones distribuidas no disminuyó. Como solución al problema antes descrito surgió lo que se dio en llamar *Middleware: "un software de infraestructura que reside entre las aplicaciones y los sistemas operativos subyacentes, las redes y el hardware"*[18], creado con el objetivo de proveer una plataforma mas apropiada para la construcción y operación de los sistemas distribuido.

Las principales funciones del Middleware son:

1. Actuar como un puente entre los programas de aplicación, el hardware de bajo nivel y el sistema operativo con el objetivo de coordinar las relaciones entre los diferentes componentes de la aplicación.
2. Proveer un conjunto de servicios reusables que permitan construir, configurar y desplegar los sistemas distribuidos en forma rápida y robusta a través de la integración de componentes que pueden ser desarrollados utilizando múltiples tecnologías.

El Middleware representa la convergencia de dos áreas principales de la Tecnología de la Información: los sistemas distribuidos y los avances de la ingeniería del software. En la figura 3 se puede observar como un cliente accede a los servicios de un componente que reside en otro computador sobre una plataforma distinta. El Middleware permite la construcción de sistemas distribuidos en forma eficaz sobre una multitud de topologías, dispositivos de computadoras y redes de comunicaciones. Tiene como meta brindar a los desarrolladores de aplicaciones distribuidas las plataformas y herramientas necesarias para 1)

15. [TABU2001], página 7.
16. [BAHA2003], capitulo 7, parte 5.
17. Clock drift: del inglés, significa desajuste del reloj.
18. [SCHSCH2002], página 1.

formalizar y coordinar como los componentes de la aplicación son ensamblados y como ellos interoperan y 2) monitorear, facilitar y validar la (re)configuración de recursos para asegurar la calidad de servicio de la aplicación entre extremo a extremo.[19]

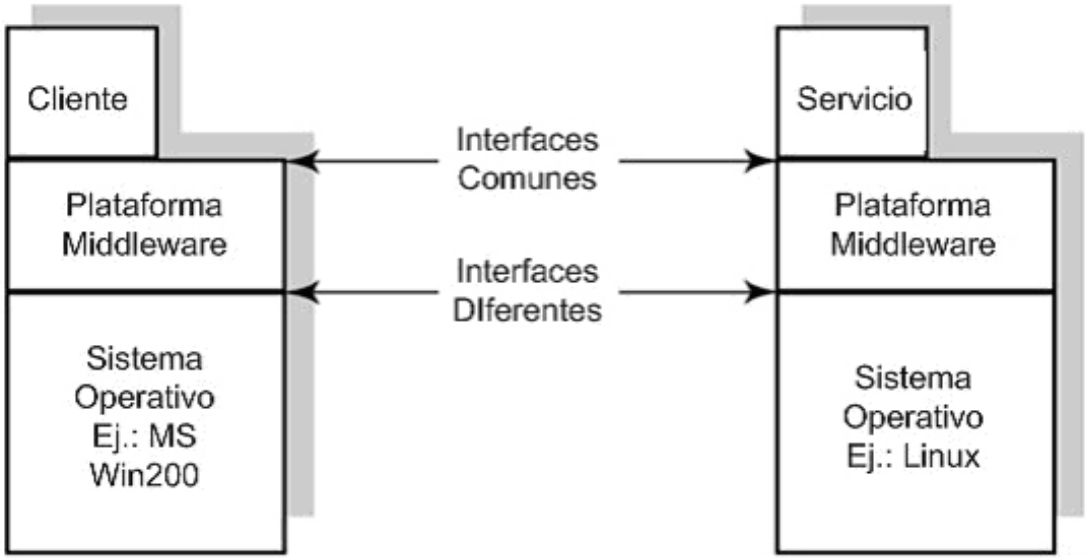


Fig. 3. El Middleware.[20]

Los desarrolladores que usan el Middleware pueden desarrollar sus aplicaciones distribuidas como si fueran aplicaciones que residen en un único computador, en vez de tener que utilizar las funciones de bajo nivel del sistema operativo: demúltiplexación de eventos, gestión de mensajes y la administración de las conexiones entre los componentes.

2.1.5. Beneficios del Middleware

La utilización del Middleware en el desarrollo de las aplicaciones distribuidas origina las siguientes ventajas:[21]

- Abstrae a los programadores de los detalles y errores propios de la plataforma de bajo nivel, como por ejemplo la programación de los sockets[22] de red.
- Amortiza el costo del ciclo de vida del desarrollo de software debido a la reutilización de la experiencia de implementaciones anteriores, capturando patrones y marcos de trabajo, en vez de tener que reconstruirlos manualmente cada vez.
- Provee un conjunto consistente de abstracciones de alto nivel orientadas a la intercomunicación de componentes que están más cerca de los requerimientos funcionales de las aplicaciones, simplificando el desarrollo de los sistemas distribuidos.
- Proveen una gran colección de servicios orientados al programador, como ser: seguridad, manejo de transacciones, etc., que resultan necesarios para operar de forma adecuada en ambientes distribuidos.

2.1.6. Principales Tecnologías del Middleware[23]

- Computación de Objetos Distribuida (Distributed Object Computing): provee una base para el soporte de objetos que pueden ser distribuidos a todo lo largo de una red, en donde los clientes invocan las operaciones en los objetos remotos. El código que provee los servicios remotos es especificado a través de interfaces que definen las operaciones y atributos de los objetos distribuidos. Ejemplos de este enfoque son: CORBA, RMI, SOAP.
- Middleware Orientado a Componentes (Components Middleware): es el sucesor del DOC, se basa en la creación de componentes relativamente autónomos, y en herramientas para dar soporte al ciclo de vida de la aplicación, como ser el armado, el despliegue, y la configuración de las aplicaciones distribuidas. Ejemplos de este enfoque son: J2EE, CCM, y .NET.

19. [SCHSCH2002], página 1.
20. [BAHA2003], capitulo 2.
21. [SCHSCH2002], página 2.
22. Socket: permite a dos nodos intercambiar un flujo de datos a través de una red, esta definido por un protocolo, un puerto y una dirección IP.
23. [SCHSCH2002], página 1.

- Middleware Orientado a la World Wide Web: que posibilita de una forma simple la conexión entre los exploradores WEB y los servidores de los sistemas de Información.
- Computación en Grilla (Grid Computing): posibilita a las grandes computadoras científicas y de alto poder de procesamiento colaborar en la solución de grandes y complejos problemas a través de una amplia variedad de redes interconectadas, como por ejemplos son los modelos de cambios climáticos.

2.2. Computación Integrada por Modelos

2.2.1. Dominios

Las computadoras evolucionaron significativamente desde la aparición de la ENIAC en 1940. Como así también las tecnologías, procesos y técnicas de programación de aplicaciones para computadoras. Los primeros lenguajes de alto nivel creados fueron FORTRAN y COBOL, los cuales pertenecían a dominios específicos en particular, el FORTRAN a la ingeniería y el COBOL a los cálculos de negocio. Sin embargo -a pesar de ser lenguajes de alto nivel-, eran dependientes del hardware de la computadora para la cual se desarrollaba. Por dominio entendemos al conjunto común de características, funciones y requerimientos propios de un contexto en particular en donde se plantea el problema a resolver. Por ejemplo el domino de las aplicaciones de tiempo real.

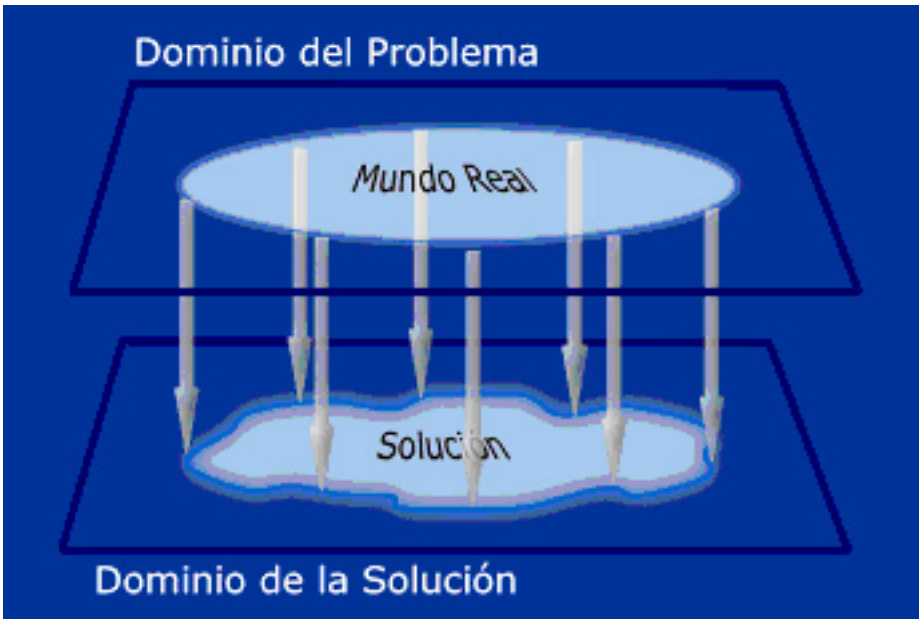


Fig.4. Representación de dominios.

Uno de los objetivos de la ingeniería de software es crear un modelo a partir de la representación de un problema del mundo real, dicho modelo es la solución al problema presentado. El dominio de la solución representa conceptualmente lo mismo que el modelo del problema pero el contexto en donde se implementa es el de la solución. Por ejemplo la representación del dominio de una solución de un problema en particular puede ser el análisis y diseño orientado a objetos. Cabe aclarar que el modelo de la solución nunca es igual al real, precisamente uno de los objetivos que busca el MIC es achicar la brecha existente entre el dominio del problema y el domino de la solución, y de esta manera hacer que el modelo creado sea lo mas cercano posible al problema planteado. En la figura 4 se grafica el concepto de dominio del problema y domino de la solución.

Antes de la aparición de los lenguajes de alto nivel el proceso de desarrollo de software requería de un diseñador que definía cual debería ser el resultado y un programador que escribía el código (generalmente eran el mismo). Una vez escrito el programa era muy difícil modificarlo ante un cambio en los requerimientos y generalmente se re-hacia completamente de nuevo. El FORTRAN y el COBOL permitieron una mayor separación de responsabilidades entre el diseñador y el programador.

Con la evolución de los lenguajes de computación aparecieron lenguajes como el C++ y el JAVA, los cuales sin ser lenguajes específicos de un dominio en particular –como lo son el FORTRAN y el COBOL-, permiten generar una solución del dominio del problema a través de una conceptualización de clases e interrelaciones de objetos, es decir los programadores a partir de una abstracción del problema específico

del dominio tratan de crear una solución a través de conceptos independientes del dominio del problema y propios de los lenguajes[24].

2.2.2. Diseño dirigido por modelos

Cada programa de software tiene como fin satisfacer los requerimientos de uno o más usuarios. El área en donde el usuario aplica el programa es el dominio de la solución. El dominio no tiene porque estar relacionado con la ingeniería del software, puede ser real o intangible.

La creación de software con valor implica tener en cuenta todas las cuestiones relacionadas con el dominio del usuario. Es una tarea que suele requerir un amplio conocimiento del dominio del problema, en donde el volumen y la complejidad de la información pueden llegar a ser inmanejables. Para contrarrestar estar complejidad se usan los modelos.

Para la ingeniería clásica: "*un modelo es una abstracción matemática que explica y/o predice el comportamiento de un determinado artefacto físico*".[25] Aquí el modelo hace referencia a cualquier construcción del pensamiento que puede ser formalizada a través de elaboraciones matemáticas.

Una definición mas genérica de modelo expresa: "*un modelo es una selección simplificada y concientemente estructurada de una forma de conocimiento*".[26] Un modelo apropiado da sentido a los datos del mismo y pone foco en el problema.

Un modelo de un dominio no es un diagrama en particular, sino que es la idea que el diagrama intenta transmitir. No es solo el conocimiento que puede tener el usuario, es una abstracción rigurosa y selectivamente organizada de ese conocimiento.

En el diseño dirigido por modelos hay tres cuestiones básicas que determinan el uso de un modelo[27]:

1. El modelo y el diseño se desprenden uno del otro. Esta relación asegura que el producto final satisfaga los requerimientos del dominio.
2. El modelo es la columna vertebral del lenguaje usado por todos los miembros del equipo. A través del lenguaje los desarrolladores pueden hablar del modelo con los usuarios expertos del dominio en forma directa.
3. El modelo representa el conocimiento desde diferentes puntos de vista, los cuales en su conjunto explican el comportamiento del dominio específico.

El diseño dirigido por modelos deja de lado la dicotomía entre el modelo de análisis y el modelo de diseño para buscar un único modelo que sirva para ambos propósitos. Esto incrementa la complejidad en el proceso de construcción del modelo, pues ahora se deben satisfacer dos objetivos muy diferentes.

2.2.3. Introducción al MIC

Podemos empezar diciendo que el MIC es una aproximación relativamente nueva al proceso de desarrollo de software, la misma actualmente es aplicable básicamente al desarrollo de los sistemas distribuidos embebidos y de tiempo real.

Es una técnica que usa modelos para describir los sistemas basados en computadoras (SBC). Fue desarrollado en el transcurso de 10 años por la Universidad de Vanderbilt, Nashville, EEUU. El MIC esta sustentado en el uso de modelos, y fue pensando para simplificar el desarrollo de grandes y complejos sistemas de computación. El MIC utiliza un metamodelo para definir un lenguaje específico del dominio y restricciones de integridad, y utiliza estos metamodelos para construir automaticamente un ambiente de diseño específico del dominio. Los diseñadores del sistema usan el ambiente resultante para crear modelos de dominio que luego analizan y automáticamente traducen el modelo a la plataforma de destino del SBC.[28]

El resumen podemos decir que el MIC es un paradigma de desarrollo que aplica en forma sistemática lenguajes de modelado específicos del dominio a la ingeniería del software para la construcción de sistemas que abarcan desde las soluciones de pequeña escala embebidas hasta las soluciones de gran escala empresariales. En el paradigma del MIC los desarrolladores de la aplicación trabajan con un modelo que reúne todas las vistas posibles del dominio del problema. En vez de enfocarse es una única aplicación, el modelo del MIC captura la esencia de todas las clases de aplicaciones. Cuando el MIC es utilizado en forma apropiada ayuda a:[29]

24. [SPRINKLE2002], página 1.
25. [SPRINKLE2002], página 1.
26. [EVANS2003], parte 1.
27. [EVANS2003], parte 1.
28. [MIC2001] , página 44.
29. [MIC2003] , página 5.

- Liberar a los desarrolladores de la dependencia en una determinada API, lo cual asegura que el modelo sirva por mucho tiempo, a pesar de que las APIs se vuelvan obsoletas.
- Proveer de un marco de pruebas de los algoritmos utilizados, a través del análisis automático del modelo.
- Confiabilidad en la síntesis del código, pues las herramientas utilizadas serán probablemente las correctas.
- Permitir la construcción fácil y rápida de prototipos a través de la introducción de nuevos conceptos al modelo usando el metamodelo.
- Ahorrar una significativa cantidad de esfuerzo y tiempo en los proyectos, y reducir el time-to-market[30] de la aplicación.

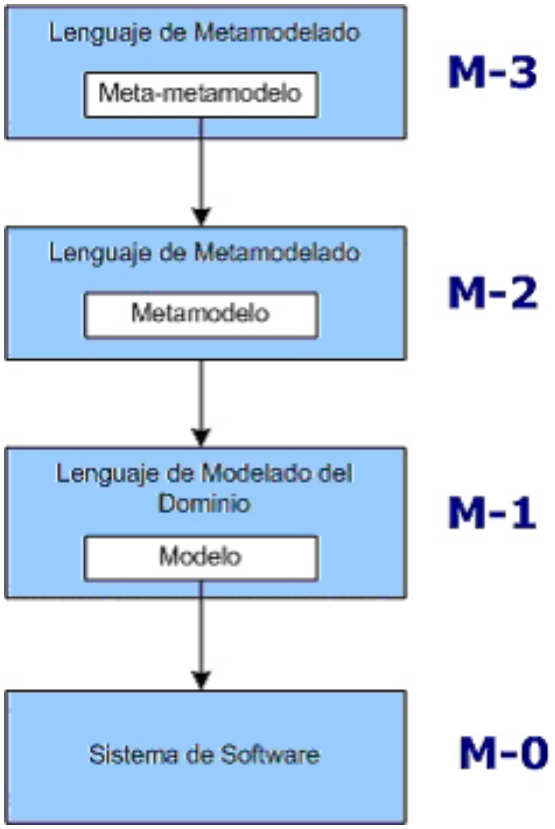
El MIC usa un conjunto de herramientas para[31]:

1. Analizar las características interdependientes del sistema capturado en el modelo.
2. Determinar la factibilidad del soporte de los requerimientos no funcionales del sistema como ser la Calidad de Servicio (QoS).

Una de las justificaciones más fuertes que alienta el enfoque del desarrollo de sistemas basados en modelos, es que el modelo ofrece un mejor camino para manejar complejidad, que los actuales lenguajes utilizados orientados a objetos y procedurales.

2.2.4. Metamodelos

La creación de ambientes de diseño específicos de dominios requiere del soporte de modelos conceptuales de abstracción genéricos que sean aplicables a un amplio rango de dominios diferentes. El MIC utiliza una arquitectura multi-nivel, como se aprecia en la figura 5, la cual consta de cuatro capas como la aplicada en el UML. El MIC define para el nivel 3 un único lenguaje, el cual permite la descripción de lenguajes de modelado para una gran variedad de dominios específicos. Este lenguaje es llamado meta-metamodelo, pues se define así mismo. El meta-metamodelo especifica un metamodelo, un lenguaje de modelización del dominio, que a su vez, especifica modelos de dominios específicos de un SBC. Lo más particular de esta arquitectura de cuatro niveles es que siempre un nivel es descrito en términos del próximo nivel superior en la jerarquía[32].



30. Time to market: del inglés, hace referencia al tiempo de salida de un producto al mercado.
31. [MIC2003] , página 5.
32. [MIC2001], página 44.
33. [MIC2001], página 45.

2.2.5. La Semántica Estática

El metamodelo especifica el lenguaje de modelización del dominio, esto es la sintaxis del lenguaje, el MIC utiliza para este propósito el lenguaje unificado de modelado. El metamodelo es definido a través de los diagramas de clases UML. Con respecto a la semántica estática del lenguaje, el metamodelo no cubre todos los aspectos, es decir no provee un conjunto básico de reglas para la correcta formación de un modelo de dominio –como por ejemplo la reglas de multiplicidad de la asociación-. El MIC utiliza para la especificación de reglas semánticas complejas el OCL un lenguaje lógico de predicados. El metamodelo consiste de diagramas de clases UML y de reglas con restricciones OCL.[34]

2.2.6. La Semántica Dinámica

Una de los mayores beneficios que produce la utilización del MIC en el proceso de desarrollo de software radica en el hecho, de que por un lado permite la configuración automática de las herramientas de análisis a partir de la información capturada en los modelos de dominio, por ejemplo la verificación del despliegue correcto de los componentes de una aplicación distribuida. Y por otro lado el MIC traduce automáticamente los modelos de dominio en la actual plataforma de implementación y/o simulación del SBC. Cada dominio tiene por lo menos una plataforma de ejecución, la cual tiene su propio lenguaje de modelización de ejecución que define las semánticas de la ejecución del modelo de dominio.

Primero un traductor, traduce los modelos de dominio en modelos ejecutables –por ejemplo en la forma de interfaces IDL de CORBA-. Este proceso de mapeo asigna la semántica dinámica a los modelos. Luego la plataforma de ejecución ejecuta estos modelos[35].

2.2.7. Los Traductores

Los traductores son una de las principales claves del MIC. Habitualmente los traductores son implementados manualmente a través de lenguajes como Java o C++. Siguiendo la idea de la transformación del lenguaje del modelo de dominio al lenguaje de ejecución de la plataforma, se puede pensar que a partir de la especificación de un modelo formal, se puede crear un meta-traductor que genere el código para el traductor. Uno de los mayores inconvenientes que presenta la utilización del enfoque del MIC es la dificultad a la hora de crear los meta-traductores, ya que generalmente esta actividad involucra una ingeniería inversa de la plataforma de ejecución del modelo de domino[36].

2.2.8. El Proceso del MIC

El proceso de desarrollo de software con el MIC difiere del proceso tradicional de desarrollo de software. Por un lado participan distintos roles y se aplican diferentes actividades y se producen diferentes artefactos. En la figura 6 se puede ver que el proceso comienza con los metamodeladores que definen el lenguaje de modelado del dominio, usando el ambiente de metamodelado para especificar formalmente el modelo y generar automáticamente el ambiente; este proceso es altamente iterativo. Con el metamodelo del dominio creado, se configura la herramienta de modelado específico del dominio y los modeladores del dominio lo utilizan para construir los modelos del SBC. Ellos después aplican los traductores para analizar el diseño y generar el código de la aplicación a construir[37].

34. [MIC2001] , página 45.
35. [MIC2001] , página 45.
36. [MIC2001] , página 46.
37. [MIC2001] , página 46.

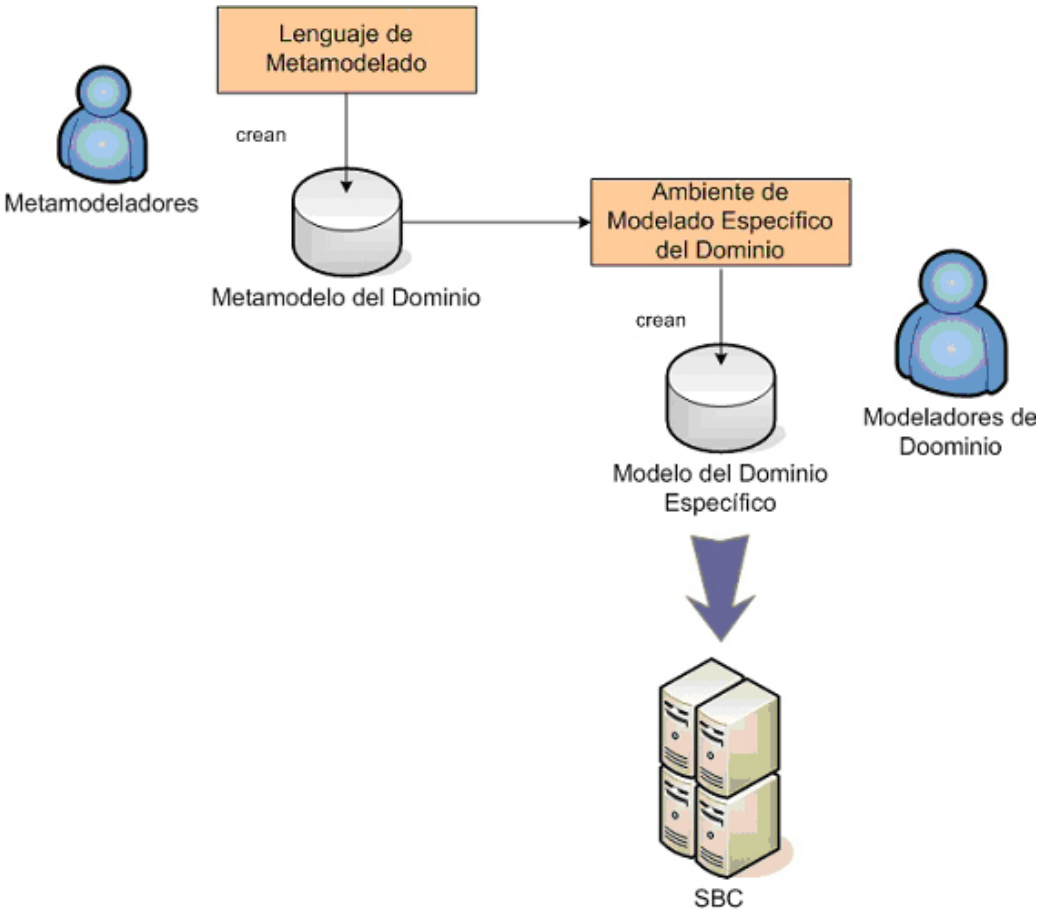


Fig.6. El enfoque del MIC del proceso de desarrollo[38].

2.2.9. Evolución con el MIC

La evolución con el MIC tiene dos componentes: por un lado la evolución de la aplicación y por el otro la evolución del metamodelo mismo.

El proceso de evolución de la aplicación en el MIC se ve facilitado por la aplicación de los modelos en la construcción del SBC, pues una modificación en la aplicación solo requiere una modificación en el modelo desde la cual se traduce, muy por el contrario en el proceso de desarrollo convencional la evolución de una aplicación requiere de un programador experto, que analice el impacto del cambio y sus consecuencias en el resto de las funcionalidades de la aplicación.

Con respecto a la evolución del metamodelo, esta se traduce en la evolución del ambiente específico de dominio creado originalmente. Por lo que implica un cambio estructural para la aplicación, sin embargo el MIC lo soluciona simplemente modificando el metamodelo, y luego el modelo que finalmente generara el nuevo artefacto a través del traductor.[39]

2.2.10. Una Definición

Ahora presentaremos la definición formal de MIC encontrada en [KAGLEDE2003]:

> *"El MIC es un enfoque, dirigido por modelos específicos del dominio para el desarrollo del software, que usa modelos y transformaciones entre modelos como una primera clase de artefactos, y donde los modelos son sentencias de lenguajes de modelado de dominio específicos (domain-specific modeling-language según sus siglas en inglés DSML). MIC captura las invariantes del dominio en las construcciones fijas de los DSML (o sea la "gramática") y la variabilidad del dominio en los modelos (o sea las "frases o sentencias")".*[40]

MIC orienta el desarrollo de software a un marco de trabajo lingüístico: en donde el desarrollador debe definir un DSML (incluyendo el motor de transformación que interpreta las sentencias), y entonces usa éste para construir el producto final. De lo anterior se desprenden dos áreas de estudio en las cuales el MIC pone el foco: (1) como definir nuevos lenguajes, y (2) como definir herramientas para la transformación de esos lenguajes.

38. [SPRINKLE2002], página 3.
39. [SPRINKLE2002], página 2.
40. [KAGLEDE2003], página 2.

2.2.11. Lenguajes de Modelado Específicos del Dominio

Los lenguajes de modelado específicos del dominio son declarativos, utilizan símbolos específicos del dominio, y tienen restricciones precisas en la semántica. Para definir un DSML, primero se debe definir la sintaxis concreta (C), la sintaxis abstracta (A), y el dominio semántico (S), luego el mapeo sintáctico y semántico (M_S y M_C). La sintaxis concreta define la notación específica (textual o gráfica) usada para expresar los modelos. La sintaxis abstracta define los conceptos, relaciones y restricciones de integridad disponibles en el lenguaje. La sintaxis abstracta determina todas las sentencias correctas (sintácticamente) que pueden ser construidas (en nuestro caso: modelos). La sintaxis abstracta incluye elementos semánticos, estos definen la formación correcta de reglas para los modelos, estas construcciones son llamadas generalmente "semánticas estáticas" del modelo. El dominio semántico S es definido generalmente por medio de algún formalismo matemático en términos del significado del modelo que esta siendo explicado. El mapeo $\boldsymbol{M_C: A \rightarrow C}$ asigna las construcciones sintácticas (gráficas, textuales o ambas) a los elementos de la sintaxis abstracta. El mapeo semántico $\boldsymbol{M_S: A \rightarrow S}$ relaciona los conceptos sintácticos con el dominio semántico.

Formalmente un lenguaje de modelización esta definido por una tupla de cinco elementos: una sintaxis concreta (C), una sintaxis abstracta (A), un dominio semántico (S), un mapeo sintáctico (M_C) y un mapeo semántico (M_S):

$$\boldsymbol{L = <A, C, S, M_S, M_C>}$$

Como se menciono anteriormente una de los factores determinantes del éxito del MIC, es la definición de los DSML a través de métodos y herramientas adecuados que automaticen parte del proceso de definición de cada uno de los elemento de la tupla, ya que sino el alto costo producido en el esfuerzo de la definición de los lenguajes no permitiría la aplicación del MIC en el proceso de desarrollo de software. Como estos lenguajes son usados para definir lenguajes de modelado, se los denominan *meta-lenguajes*, y las especificaciones concretas del DSML *metamodelos*.[41]

2.2.11.1. Modelado de la Sintaxis Abstracta

La especificación de la sintaxis abstracta de los DSMLs requiere un meta-lenguaje con la capacidad para expresar conceptos, relaciones y restricciones de integridad. Como ya se menciono anteriormente el MIC adopto los diagramas de clase UML y el OCL como meta-lenguajes. Esta elección esta fundamentada por las siguientes razones: a) el UML y el OCL son dos estándares de la industria soportados por la OMG, (b) la gran variedad de herramientas actuales que los soportan y (c) la ventaja del uso extendido de los diagramas de clase UML minimiza la dificultad encontrada en la definición conceptual de la arquitectura a través de los lenguajes de los metamodelos[42].

2.2.11.2. Modelado de la Sintaxis Concreta y el Mapeo Sintáctico

La sintaxis concreta puede ser considerada como un mapeo de la sintaxis abstracta a un modelo específico de interpretación. Mientras que el objetivo de la sintaxis abstracta es definir las estructuras de datos que deberán definir a nuestros modelos, la sintaxis concreta captura como los modelos deben ser interpretados.

Esta distinción entre la sintaxis concreta y abstracta tiene un profundo impacto en la tecnología utilizada para la construcción de modelos: operación y edición. El modelador usa la sintaxis concreta para crear y modificar estructuras utilizando los conceptos "primitivos" de la sintaxis concreta. Sin embargo estas estructuras son simplemente una interpretación de los objetos subyacentes de la sintaxis abstracta[43].

2.2.11.3. Modelado del Dominio Semántico y Mapeo Semántico

El dominio semántico y el mapeo semántico definen la semántica de un DSML. El rol de la semántica es describir las propiedades (el significado) del modelo que queremos crear a partir del lenguaje de modelado. Lo interesante es que un DSML puede tener más de una semántica, en función de cada una de las propiedades que se desea describir de él. Por ejemplo generalmente un modelo tendrá una semántica para las propiedades *estructurales* y otra para las propiedades del *comportamiento*. La semántica estructural de un lenguaje de modelado describe el significado de los modelos en términos de su composición: la configuración posible de sus componentes y las relaciones entre ellos. La semántica estructural puede ser definida formalmente por intermedio de un conjunto de relaciones matemáticas. La semántica del comportamiento describe la evolución del estado de los elementos modelados a lo largo de algún modelo de tiempo. Por otro lado la semántica del comportamiento es modelada formalmente a

41. [MIC-2-2003], página 6.
42. [MIC-2-2003], página 6.
43. [MIC-2-2003], página 7.

través de estructuras matemáticas que representan alguna forma de dinamismo, como por ejemplo las "maquinas de estados finitos".
Existen dos enfoques usados para la especificación de semánticas: uno es a partir del metamodelado y otro es a partir de la traducción.

- En el enfoque del metamodelado, la semántica es definida por un meta-lenguaje que ya cuenta con una semántica bien definida. Por ejemplo el UML/OCL que utiliza el MIC para definir la sintaxis abstracta de un DSML tiene un significado estructural: éste describe los posibles componentes y la estructura válida de modelos de dominio sintácticamente correctos.
- El enfoque de traducción especifica las semánticas vía la definición de un mapeo entre el DSML y otro lenguaje de modelado que posea una semántica bien definida.

La formulación de reglas bien formadas en la sintaxis abstracta requiere una compresión cabal del dominio semántico S a fin de asegurar que el mapeo semántico de cada modelo correcto conduce a un modelo semántico consistente[44].

2.2.12. Composición de Metamodelos y Modelos

La composición en el diseño basado en modelos se presenta en dos niveles: (1) composición de DSMLs por medio de la composición del metamodelo y (2) composición de modelos en el contexto específico de los DSMLs.

2.2.12.1. Composición del Metamodelo

La construcción por composición de DSMLs requiere la $L_N = L_1 \,||\, L_2 \;\ldots\; ||\, L_K$ composición de metamodelos a partir de los DSMLs componentes. Mientras que la composición de lenguajes ortogonales (independientes) es una tarea simple, la construcción de DSMLs desde lenguajes no ortogonales es un problema complejo. La no ortogonalidad quiere decir que los componentes DSMLs comparten conceptos y reglas bien formadas que se extienden a través de los aspectos individuales del modelado.
Como el MIC utiliza el UML/OCL como lenguaje de metamodelado no soporta la composición modular de metamodelos, esta restricción provoco el agregado de tres nuevos operadores que posibilitan la composición de los metamodelos (ver figura 7). Dos de estos operadores son una especialización del operador de relación de herencia del UML. El objetivo es que el diseñador del lenguaje especifique metamodelos, y luego los compone y los extiende para crear metamodelos nuevos utilizando estos operadores.
El primer operador" "Equivalencia" afirma que dos clases (en diferentes diagramas de clase) deben ser consideradas idénticas.
El segundo operador: "Herencia de Implementación" afirma que las clases derivadas heredaran los atributos de la clase base y todas aquellas asociaciones en donde la clase base juega el rol de un contenedor.
El último operador "Herencia de Interfaz" declara que las clases derivadas heredarán solo aquellas asociaciones donde la clase padre no sea un contenedor.

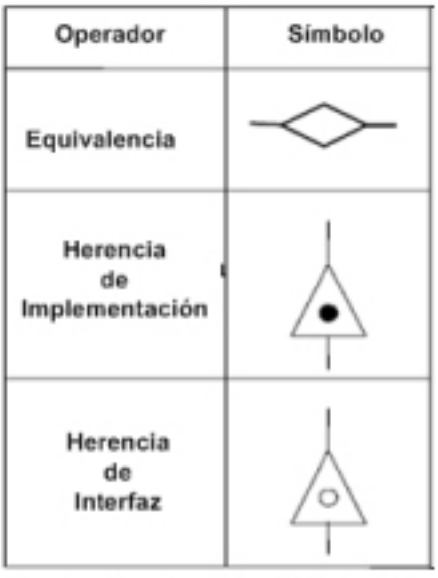


Fig.7. Operadores de Composición del Metamodelo.[45]

44 [MIC-2-2003], página 8, párrafo 2
45 [MIC-2-2003], página 9.

La inclusión de estos operadores en el lenguaje da como resultado un nuevo metamodelo, el cual conforma la semántica subyacente del UML[46].

2.2.12.2. Composición del Modelo

La composición de modelos en un DSML es una tarea esencial, que requiere de la existencia de una serie de herramientas que deben estar disponibles para el modelador. La sintaxis abstracta de un DSML define las técnicas de composición disponibles en el lenguaje. Sin embargo estas técnicas son siempre específicas a un lenguaje, la dificultad radica en poder definir técnicas generales de composición (validas para una gran variedad de dominios)[47].

Las técnicas generalmente más usadas para la creación de modelos son las siguientes:

1. *Abstracción*: es la técnica largamente mas usada en el modelado. Se puede definir en nuestro caso como la capacidad para la representación de sistemas con diferentes niveles de detalle en forma simultanea.
2. *Modularización*: es una técnica de implementación que ayuda al soporte de la abstracción en la práctica. La forma de modelar un sistema complejo es descomponerlo en entidades intrínsicamente coherentes: los modelos en un DSML deberían ser módulos y soportar la composición. Soportar la composición significa que la composición de dos o mas módulos resultan en otro modulo también. Por otro lado la modularización es un vehiculo para implementar la abstracción, si tomamos vemos al modulo como una caja negra que oculta todos los detalles internos de implementación del modulo.
3. *Componentes e Interfaces*: permite definir un método común de interconexión entre módulos.
4. *Múltiples Aspectos*: permiten controlar la complejidad restringiendo la información presentada al modelador como así también ayuda al modelado orientado a aspectos.
5. *Referencias*: reducen la complejidad permitiendo la vinculación de componentes a diferentes niveles de jerarquías dentro del modelo. Dan al modelador la herramienta necesaria para hacer uso de entidades en otras jerarquías o modelos que de otro forma requeriría hacer una copia de la entidad referenciada. En la figura 8 se puede apreciar el concepto de referencia explicado.

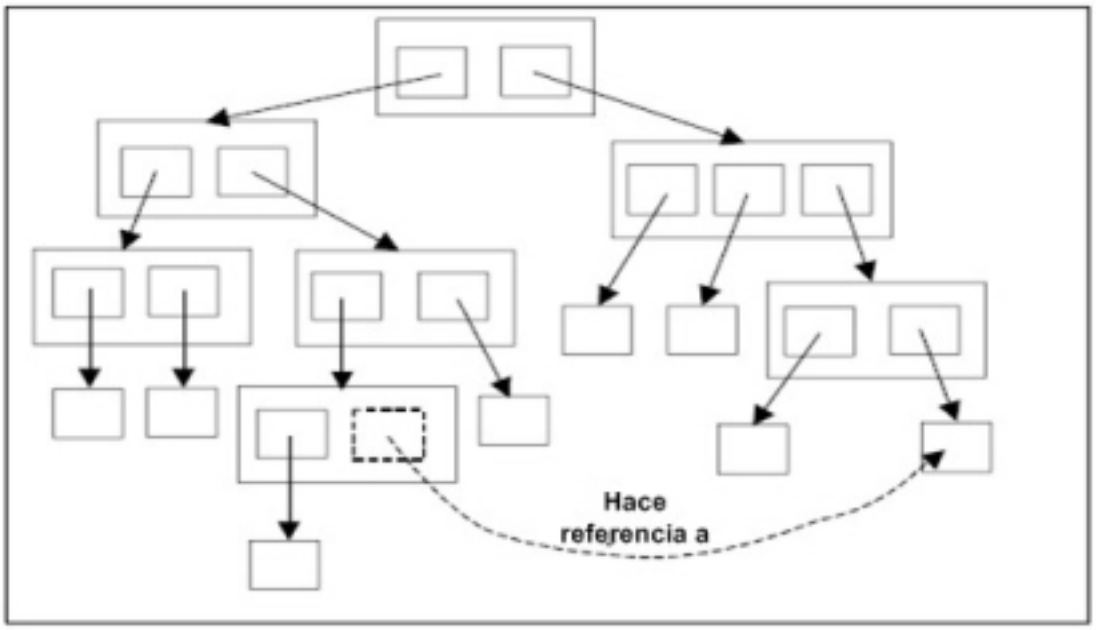


Fig.8. Concepto de Referencia.[48]

Estas técnicas de modelado permiten modelar los sistemas orientados a intereses, esto se logra a través del modelado por múltiples aspectos, en donde cada aspecto describe el sistema desde un punto de vista en particular, y la clave esta en como cada uno de estos aspectos interactúan entre ellos.

2.2.13. El Modelado por Composición

La construcción de modelos complejos a través de la composición de componentes: $\mathbf{M_N = M_1 \,||\, M_2 \ldots ||\, M_K}$ es una técnica muy eficiente de modelado. La condición de composicionalidad en el método bottom-

46. [MIC-2-2003], página 9.
47. [MIC-2-2003], página 12.
48. [MIC-2-2003], página 13.

up[49] es que si el componente M_K tiene la propiedad P_K, entonces dicha propiedad deberá ser preservada luego de la integración del componente M_K con otros componentes[50].

2.2.14. Síntesis del Modelo

La síntesis del modelo puede ser formulado como un problema de búsqueda: partiendo de un conjunto de componentes del modelo $\{M_1, M_2, \ldots M_k\}$ (los cuales pueden representar diferentes vistas del sistema) y un conjunto de operadores de composición. El problema que se presenta aquí es como satisfacer el conjunto de propiedades en el modelo resultante de la integración de uno o más modelos[51].

2.2.15. Generadores Basados en Modelos

La principal característica del enfoque de desarrollo basado en modelos es que los modelos son usados como la entrada de los generadores, los cuales traducen el modelo en otro modelo (o artefacto) usado para en el análisis y en la ejecución de la aplicación.
La integración entre los elementos participantes en el proceso de desarrollo basado en modelos radica en el uso de *generadores* (interpretes de modelo o traductores) que traducen el modelo a otra forma (otro modelo). Nosotros usamos la traducción de modelos para: (1) la traducción de modelos al lenguaje de entrada de herramientas de análisis y viceversa y, (2) para la traducción de modelos a código ejecutable, archivos de configuración, etc. Y cualquier otro artefacto que sea necesario para ejecutar la aplicación en una plataforma determinada.
Un generador implementa un mapeo entre la semántica del modelo de dominio y otro modelo.
Las tres técnicas más utilizadas para la implementación de generadores son: la implementación directa, el uso de patrones y los meta-generadores[52].

2.2.15.1. Implementación Directa

Esta técnica es similar a la usada por los compiladores, la diferencia radica en que el generador mapea la sintaxis abstracta del lenguaje de entrada a la sintaxis abstracta del lenguaje de salida, donde el lenguaje de destino tiene semánticas de ejecución bien definidas. Volviendo al caso de los compiladores, la tarea del generador se reduce a la creación de un árbol de salida a partir de un árbol de entrada[53].
Un generador realiza las siguientes operaciones:

1. Construye el árbol de entrada, el árbol de entrada es el modelo mismo.
2. Recorre en forma transversal el árbol de entrada en múltiples pasadas para construir el árbol de salida. En este paso el generador detecta todas las entidades, atributos, relaciones y propiedades necesarias para armar el árbol de salida.
3. Crea el producto final en la forma requerida: un archivo de texto, código de análisis, etc.

2.2.15.2. Uso de Patrones

El uso de patrones mejora la técnica descripta anteriormente incorporando el uso de patrones en el diseño de los generadores. El uso del patrón de diseño “Visitador” permite recorrer en forma transversal el árbol. En función de las entidades visitadas el patrón implementa ciertas acciones en cada uno de los nodos del árbol de entrada, ejecutando las operaciones específicas del nodo. La construcción de generadores utilizando el patrón visitador se puede ver facilitada por la utilización de técnicas que automaticen la construcción de generadores de código[54].

2.2.15.3. Meta Generadores

Como se comentó anteriormente la construcción de generadores de código se puede ver facilitada si utilizamos modelos que nos permitan automatizar su creación, dicha técnica es llamada meta-generadores.
La idea descripta aquí hace referencia a la creación de un modelo de generadores, que permita la especificación declarativa de los generadores en vez de la imperativa (la utilizada por los lenguajes de programación). Una aproximación del uso de modelos de generadores es la utilización de grafos y gramáticas las cuales son expresadas en forma matemática, entonces el árbol de entrada y el árbol de salida son representados como grafos los cuales soportan un gramática en particular y luego por medio

49. Bottom-up: del inglés, estrategia para la descomposición de sistemas de información, las partes individuales se diseñan con detalle y luego se enlazan para formar componentes más grandes, que a su vez se enlazan hasta que se forma el sistema completo
50. [MIC-2-2003], página 18.
51. [MIC-2-2003], página 19.
52. [MIC-2-2003], página 20.
53. [MIC-2-2003], página 21.
54. [MIC-2-2003], página 22.

de una transformación generan el código en un determinado lenguaje de programación que por último implemente la transformación del modelo[55].

3. GME

3.1. Introducción

La principal pieza de un proceso de desarrollo basado en modelos es el DSML[56]. Sin embargo la construcción de nuevos DSMLs es muy compleja e insume mucho tiempo y esfuerzo. El costo es una suma de varios factores: el costo del desarrollo de un nuevo lenguaje, el costo del entrenamiento del modelador en el nuevo lenguaje y el costo de las herramientas que deberán soportar el lenguaje. Estos costos pueden ser minimizados a través de diferentes métodos pero lo que resulta fundamental es contar con herramientas que permitan una rápida y sencilla definición de nuevos lenguajes de modelado. Esto último se logra utilizando la técnica llamada metamodelado y meta-programación.
En síntesis podemos decir que resulta imprescindible para una aplicación real del MIC la existencia de herramientas que produzcan nuevos modelos específicos de dominio para los modeladores del DSML. Esta herramienta debe ser una herramienta de modelado meta-programable, programada vía la representación explicita de metamodelos. La figura 9 es una ampliación de la figura 6 que describe este proceso. El diseñador (metamodelador) del DSML crea los metamodelos: la definición de la sintaxis y de la gramáticas del DSML utilizando una herramienta gráfica para generar metamodelos, y estos metamodelos son entonces utilizados en el proceso de configuración del entorno de modelado específico del dominio y por último se crean los modelos específicos del dominio utilizando una herramienta grafica para modelar dominios específicos. Lo que resulta aquí es que la misma herramienta es usada para la creación del modelo y para la creación del metamodelo, pues el lenguaje de un metamodelo es otro DSML

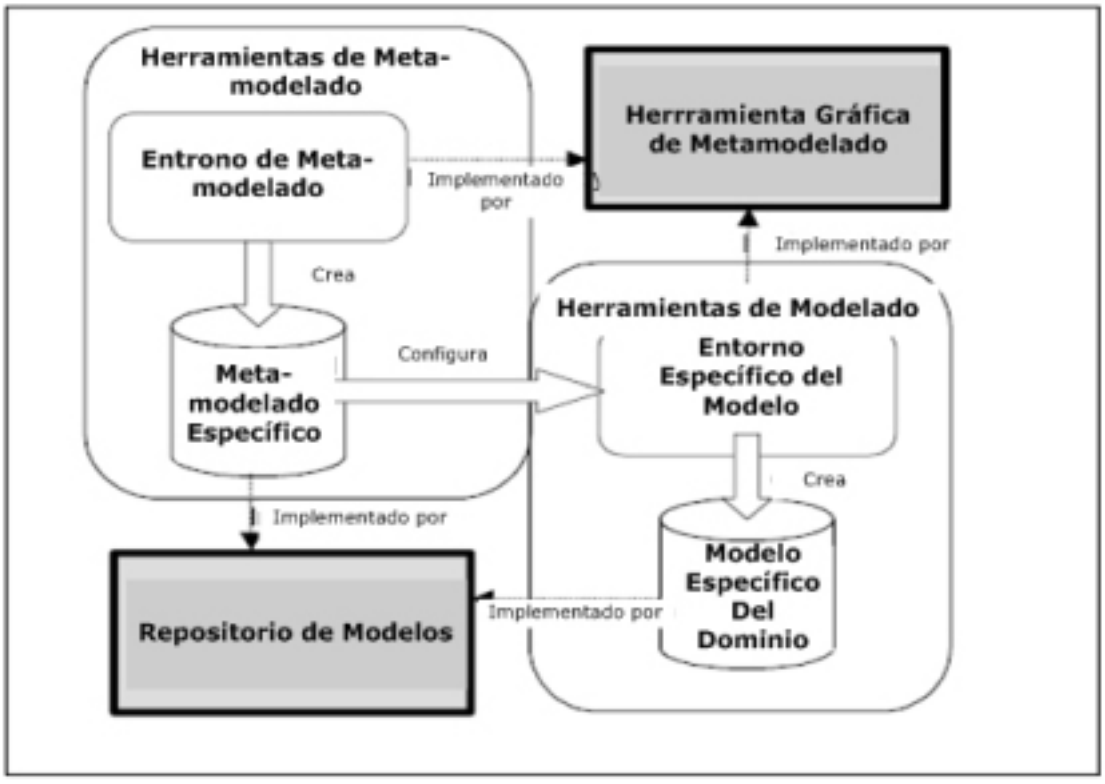


Fig.9. Las herramientas en el MIC.[57]

Una herramienta que soporta las características descriptas más arriba es el GME *(ver capítulo 9)*, una aplicación de código abierto desarrollada por la Universidad de Vanderbilt, Nashville, EEUU. El GME es un entorno de modelado para un dominio específico, que puede ser configurado y adaptado a través de

55. [MIC-2-2003], página 22.
56. [MIC-2-2003], página 15
57. [MIC-2-2003], página 16.

metamodelos que especifican DSMLs. Las tres principales características de GME son:[58]

1. El GME provee primitivas de modelado genérico que asisten al diseñador en la especificación de nuevos entornos gráficos de modelado.
2. Estas primitivas genéricas permiten crear los conceptos específicos del modelo a través del metamodelado. El metamodelo soporta la composición, la cual posibilita la creación de lenguajes de modelado compuestos para múltiples dominios.
3. Permite la clonación de los modelos gráficos para la creación de otros modelos a partir de los primeros.

El éxito del uso de este tipo de herramientas radica en la necesidad de crear un ambiente de colaboración entre profesionales de distintos perfiles y profesiones. El uso del MIC plantea la necesidad de nuevos roles en el proceso de desarrollo de software:

Diseñadores del Entorno: construyen los metamodelos específicos del domino. Requieren un conocimiento profundo de GME y del dominio del problema, pues el metamodelo debe soportar todos los conceptos que los expertos del dominio necesitarán para crear los modelos de la aplicación. Estos también deben especificar como se realiza la transformación del modelo en los artefactos para la plataforma elegida.

Expertos del Dominio: construyen los modelos específicos del dominio. No requieren un conocimiento profundo de GME pero si del dominio específico para el cual se desea crear el modelo.

Desarrolladores GME: construyen el traductor que realiza la síntesis del modelo. Requieren un conocimiento profundo de las técnicas genéricas de modelado de GME.

3.2. Conceptos Generales de Modelado en GME

El modelado de cualquier sistema grande y complejo requiere que el modelador pueda describir las entidades, atributos y relaciones en una forma clara y concisa. La herramienta de modelado debe restringir al modelador solo a la creación de modelos validos sintáctica y semánticamente, a su vez debe dar al modelador la flexibilidad necesaria para el soporte de modelos que soporten una gran variedad de dominios. Las cuestiones de ¿qué modelar?, ¿cómo modelarlo?, y ¿qué tipos de análisis? serán realizados para la construcción del modelo deberán ser especificados antes de comenzar a crea el metamodelo. Estas cuestiones son descriptas por el *paradigma del modelo*. Por consiguiente la primera tarea y la más importante a realizar en la creación de un DSML: es la definición del paradigma del modelo[59].

Un paradigma de modelo define la clase de modelos que podrán ser construidos, como ellos están organizados, que información contendrán, etc. Cuando el GME es utilizado, éste está configurado para un dominio de aplicación en particular. Un paradigma de un modelo puede ser uno de los siguientes:

- Paradigmas para el modelado de sistemas de tiempo real embebido.
- Paradigmas para el modelado de procesos químicos.
- Paradigmas para aplicaciones empresariales financieras.
- Paradigmas que describen otros paradigmas.

El GME tiene dos tipos de modelos jerárquicos distintos: el metamodelo (creado por el diseñador del entrono) y el modelo de la aplicación (creado por el experto del dominio). Ambos contienen jerarquías formadas por un árbol. Cada elemento del modelo de la aplicación tiene un elemento que le corresponde en el metamodelo, la representación del elemento es llamado *tipo*. La primera función que tiene el metamodelo es describir los diferentes tipos de elementos de modelado que el experto del dominio puede usar

3.3. Modelo

Por modelo nosotros entendemos cualquier entidad abstracta que represente algo en el mundo[60]. Lo que representa un modelo depende del dominio en particular sobre el cual estemos trabajando. Por ejemplo:

- Un Validador de Transacciones es un modelo que representa un autorizador de tarjetas de crédito en el dominio de aplicaciones de retail.
- Un Proceso es un modelo que representa un proceso químico para el dominio de las plantas de procesos químicos.
- Un Dashboard[61] es un modelo que representa un panel de control para el dominio de las aplicaciones de control de infraestructura de HW.

58. [GME2004], página 263.
59. [GMEMAN2004], Página 12.
60. [GMEMAN2004], página 12.
61. Dashboard: panel de control.

Fig.10. Icono por defecto que representa un modelo en GME.

En la figura 10 se ilustra el icono de un modelo en GME. Un modelo, en términos computacionales es un objeto que puede ser manipulado[62]. Un objeto tiene un estado, una identidad y un comportamiento. El propósito del GME es crear y manipular estos modelos, otros componentes del GME interpretan estos modelos y generan código para ser ejecutado en varios contextos (plataforma de ejecución, herramientas de análisis, etc).
Los paradigmas pueden estar creados por varias clases de modelos. Un modelo generalmente contiene *parts*[63], otros objetos contenidos dentro del modelo. Una part puede ser alguno de los siguientes ítems:

- *atoms*[64] (parts indivisibles),
- otros modelos,
- *references*[65] (apuntan a otro objeto del modelo),
- *sets*[66] (pueden contener otras parts) y
- *connections*[67].

Si un modelo contiene parts, nosotros decimos que el modelo es el padre de esas parts. Las parts pueden tener varios atributos. Un atributo especial asociado con un atom permite a la part ser usada como una *link part*[68]. Las link parts actúan como puntos de conexión entre modelos. Los modelos que contienen otros modelos como parts son llamados *modelos contenedores*. Y los modelos que no pueden contener a otros modelos son llamados *modelos primitivos*. Si un modelo contenedor puede contener a otros modelos se lo denomina *modelo jerárquico*.
En GME cada part es representado por un icono:

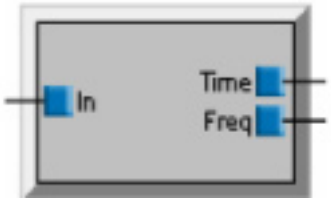


Fig.11. Modelo con atoms como links ports.

3.4. Atoms

Los atoms (o atomic parts) son objetos de modelado simples que no tienen una estructura interna (no contienen a otros objetos) pero pueden tener atributos. Son usados para representar entidades indivisibles, y existen en el contexto de su modelo padre.

Fig.12. Icono por defecto que representa un atom en GME.

En la figura 13 se aprecia un modelo primitivo en donde se definieron 4 atoms.

62. [GMEMAN2004], página 13.
63. Parts: del inglés, significa parte.
64. Atoms: del inglés, significa átomo.
65. References: del inglés, significa referencias.
66. Sets: del inglés, significa conjunto de elementos.
67. Connections: del inglés, significa conexiones.
68. Link part: del inglés, significa enlace entre partes.

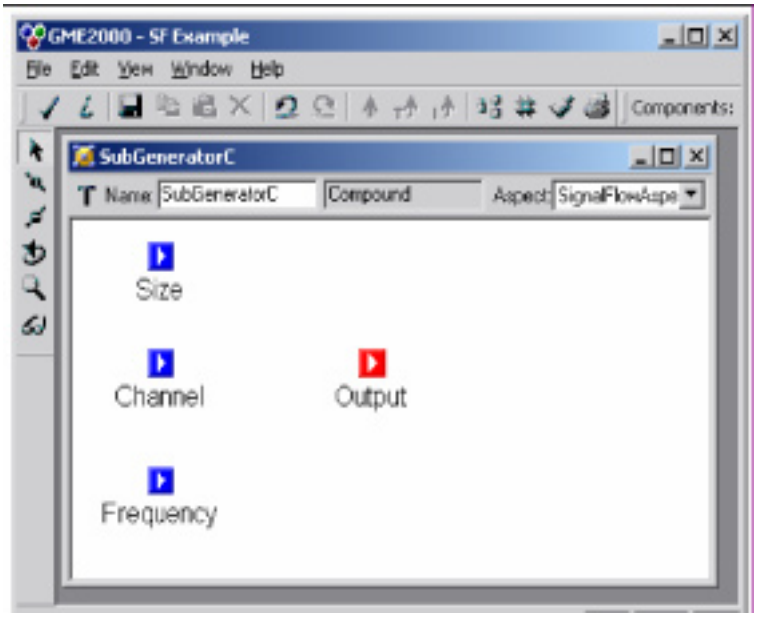


Fig.13. Un modelo primitivo conformado por 4 atoms.

Los siguientes son ejemplos de atoms:

- Un POS en el modelo de un Validador de Tarjetas de Crédito para el dominio de aplicaciones de retail.
- Un Variable en el modelo de un Proceso para el dominio de las plantas de procesos químicos.
- Un Widget en el modelo de un Dashboard para el dominio de las aplicaciones de control de infraestructura de HW.

3.5. Modelo Jerárquico

Los modelos representan el mundo en diferentes niveles de abstracción. Entonces un modelo que contiene a otros modelos representa algo con un mayor nivel de abstracción que un modelo que no contenga a otros modelos (menor nivel de abstracción). Esta organización jerárquica ayuda al modelador a manejar la complejidad en la construcción de modelos que representen grandes sistemas, usando un alto nivel de abstracción (menos detallado). En los modelos con menor nivel de abstracción se describen una mayor cantidad de detalles pero la vista proporcionada respecto a todo el sistema es menor. En los casos en donde un modelo contiene a otros modelos como parts, hablamos de un *modelo jerárquico*. En la figura 14 se muestra un modelo jerárquico de un generador de flujo de datos en donde hay atoms de entrada de la señal, componentes en el medio y un atom de salida con de la señal ya transformada.

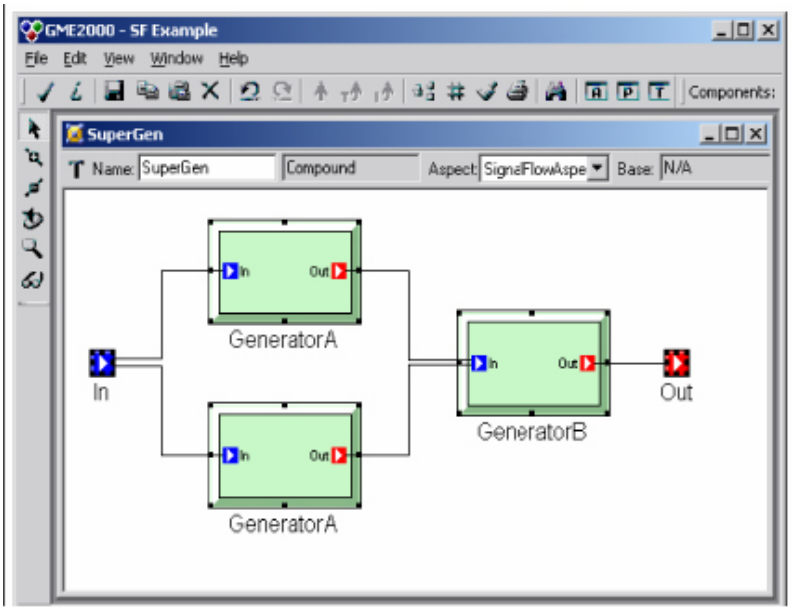


Fig.14. Un modelo jerárquico de un generador de flujo de datos.

3.6. References

Las *references* son parts que poseen un concepto similar a los punteros en los lenguajes de programación. En la creación de modelos complejos, algunas veces es necesario acceder en forma directa desde un modelo a una part contenida en otro modelo. Para eso en GME se introdujo el concepto de references parts: las references parts son objetos que hacen referencia a otros (apuntan a) objetos del modelo. Una referente part puede apuntar a un modelo, a un atomic part de un modelo, a un modelo embebido en otro modelo o a otra referente part o set. Una reference puede ser creada sólo si el objeto al que hace referencia fue creado, sin embargo se pueden crear reference a que hagan referencia a nulo.

Fig.15. Icono por defecto que representa una reference nula en GME.

El icono para las references es definido por el usuario pues es el mismo que representa al objeto al cual referencia.

3.7. Connections y Links[69]

Para representar las relaciones del modelo el GME utiliza diferentes métodos, las *connections* es el método mas simple. Una connection es una línea que une dos parts de un modelo. Las connections tienen al menos dos atributos: la apariencia (ayudan al modelador a distinguir las connections) y la direccionalidad (representa la dirección de la relación).
La semántica de la connection es determinada por el modelo de paradigma, es definida en el metamodelo, en el metamodelo se especifican todas las connections validas del modelo. Luego GME verifica la validez de las connection del modelo. La validez radica en la verificación de si los dos tipos de objetos pueden conectarse y en la direccionalidad de la conexión.
Para definir una conexión el modelador debe seleccionar en GME el modo “Add Connections”. Una connection siempre conecta dos parts, si una de las parts es un modelo entonces la connection tiene un connection point[70] o *link*. Un link es un port a través del cual un modelo puede conectarse a otro part dentro del modelo padre. Las connectios pueden conectar atomic parts y modelos.

3.8. Sets

Las parts y las connections representan los componentes estáticos del modelo. GME introduce el concepto de *sets* para representar los componentes dinámicos del modelo. Desde el punto de vista visual significa que, dependiendo del estado del sistema, se deberán ver ciertas parts y otras no.

Fig.16. Icono por defecto que representa una set en GME.

Estos estados son definidos por el modelador, dependiendo del estado actual del sistema se deberán ver ciertas parts, mientras que otras se ocultarán. Cuando un set es activado solo los objetos pertenecientes a ese set son visibles. Las parts pueden pertenecer a una sola set, a más de un set o a ningún set. Para añadir o remover parts de un set, se debe seleccionar el modo “Set“ en el GME.

3.9. Aspects

Como se mencionó anteriormente para manejar la complejidad del modelado de grandes sistemas se usan las jerarquías. Sin embargo puede ocurrir que un modelo contenga una gran cantidad de parts, para estas situaciones GME introduce el concepto de *aspects*[71], Un aspect es definido por las clases de parts que son visibles en una vista, los aspects relacionan grupos de parts. La visibilidad o existencia de un part dentro de un particular aspects son definidas en el paradigma del modelo. Para cada clase de part hay dos aspect: el primario y el secundario. Las parts solo pueden se añadidas o removidas desde el aspect primario. Los aspects secundarios solo pueden heredar las parts de los aspects primarios. Asimismo diferentes connections se pueden definir para las mismas parts en diferentes aspects.

69. Links: del inglés, significa enlaces.
70. Connection point: del inglés, significa punto de conexión.
71. Aspects: del inglés, significa significa aspectos.

3.10. Atributos

Los modelos, atoms, references, sets y connections pueden tener atributos. Un atributo es una propiedad de un objeto que es expresada textualmente y que representa un tipo simple (en el sentido que son expresadas en forma de texto). El modelo del paradigma define que atributos son presentes para los objetos, sus tipos de datos, son rangos, sus valores. Son usados para interpretar el las restricciones creadas con el OCL.

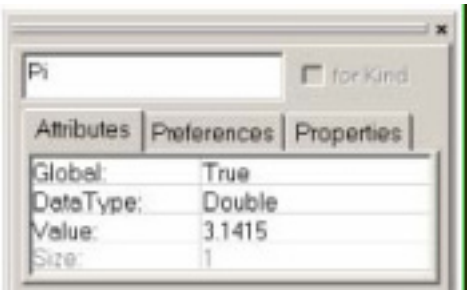


Fig.17. Ejemplo de la definición de atributos en GME.

3.11. Projects

El diagrama UML de la figura 18 describe los conceptos enumerados anteriormente del GME y las relaciones entre ellos. Estas estructuras son las primitivas de modelado genéricas que ayudan a la creación de los metamodelos.

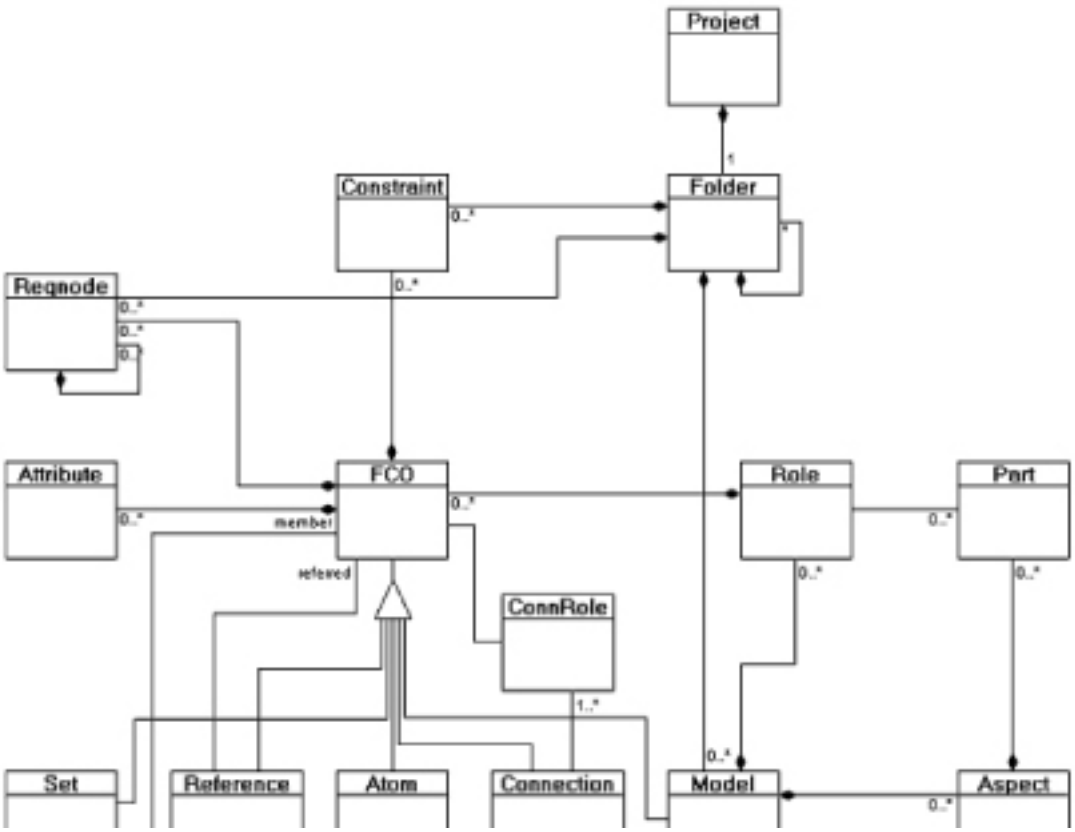


Fig.18 – Metamodelo de GME[72].

La clase raíz contenedor es llamada *Project*, y un Project contiene un único *Folder*[73], Los Folders son contenedores que ayudan a organizar los modelos. Folders contienen Modelos. Los Modelos contienen FCOs y los FCOs son cualquiera de las parts mencionadas anteriormente.

72. [GME2004], página 266.
73. Folder: del inglés, significa carpeta.

4. CCM

4.1. Introducción

Como vimos en el capitulo 2.1.4, el Middleware facilitó la construcción de sistemas distribuidos. Más precisamente el Middleware Orientado a Componentes es la nueva tecnología de Middleware que soluciona la mayoría de los problemas encontrados en la construcción de sistemas distribuidos. Y el CCM es la respuesta de OMG a este paradigma. El modelo del CCM esta basado en componentes que implementan una interfaz que exporta un conjunto de métodos y componentes estándar a los clientes.

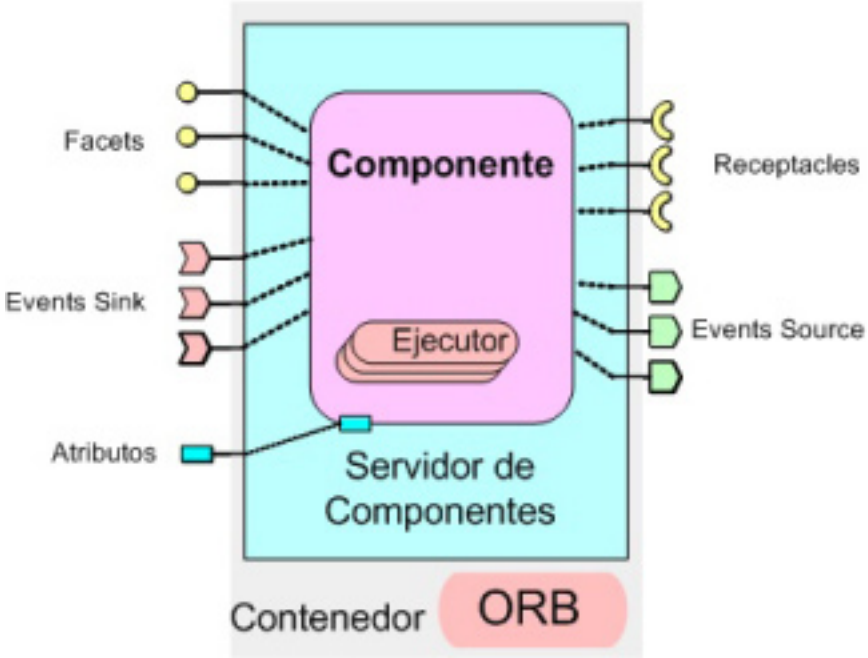


Fig.19 – Los componentes del CCM.

Los componentes también pueden expresar la intención de colaborar con otros componentes definiendo interfaces llamadas *ports*[74]. Existen 3 tipos de ports en CCM[75]:

- **Facets,** definen una interfaz que acepta invocaciones de métodos sincrónicos desde otros componentes.
- **Receptacles,** indica una dependencia a una interfaz de un método sincrónico proveído por otro componente.
- **Event sources, sinks,** indica que esta interesado en intercambiar mensajes asincrónicos con otros componentes. Los componentes que declaran los event source generan eventos y los componentes que declaran event sink consumen eventos.

Un *contenedor* provee el entorno de ejecución para un componente. El contenedor contiene diversos servicios predefinidos como ser, notificación de eventos, estrategias, persistencia, transacciones y seguridad para el componente que administra. Cada container administra un tipo de componente y es responsable de su inicialización y de la interconexión con otros componentes de los servicios del ORB. Los desarrolladores se valen de meta-datos para definir el mecanismo de cómo desplegar estos contenedores.

La idea del CCM es brindar el soporte para la construcción de aplicaciones distribuidas ensamblando bloques de software. Para eso el CCM estandarizó la implementación, el ensamblado, el empaquetamiento y el despliegue de componentes. La figura 19 describe un componente CCM, sus ports y el contenedor.

74. Ports: del inglés, significa puerto. Hace referencia a un punto de desde el cual el componente puede interactuar con el mundo exterior a él.
75. [MIC2003], página 10.

4.2. Ciclo de Desarrollo del CCM

El paradigma de programación del CCM define y describe el proceso para cada una de las diferentes etapas del ciclo de vida del desarrollo de la aplicación. En la figura 20 se muestra el ciclo de vida de desarrollo que describe el CCM.

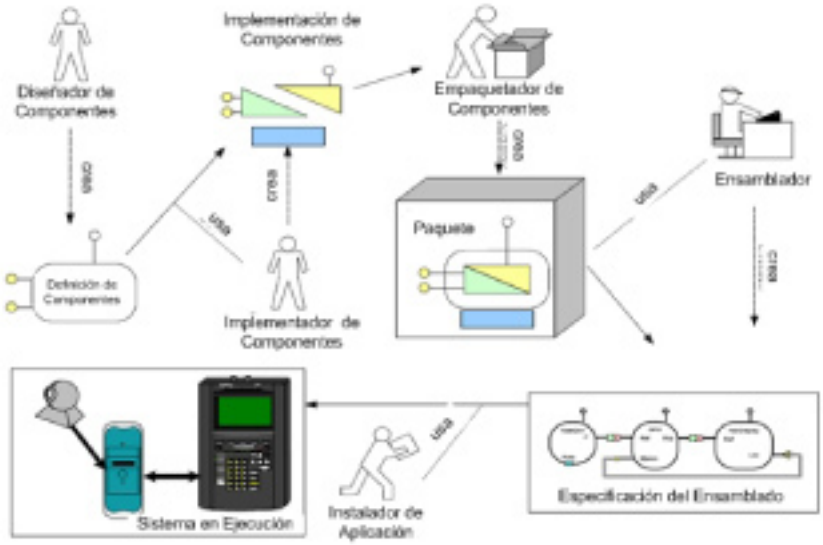


Fig.20 – Ciclo de vida del desarrollo con CCM[76].

Gracias a esta clara separación cada una de las etapas puede ser manejada en forma separada por diferentes roles[77]:

- Diseñador de Componentes: definen las características de cada componente y como colaboran entre ellos. Definen las interfaces que exportan los componentes para la comunicación con sus clientes. No especifican como debe implementarse las interfaces.
- Implementador de Componentes: realizan la implementación de los componentes definidos por los diseñadores. Existen también herramientas que ayudan al implementador a generar meta-datos llamados *descriptores de componentes*, que describen cuales son los requerimientos de ejecución de cada componente.
- Empaquetador de Componentes: crea paquetes de componentes, utilizando meta-datos para especificar la agrupación de los componentes en *paquetes de componentes*.
- Ensamblador: configura las aplicaciones seleccionando el tipo de implementación de los componentes, específica las restricciones de la instanciación de los componentes, y las instancias de conexión entre ellos a través de un meta-dato llamado *descriptor de ensamblado*.
- Instalador de la Aplicación: analiza los requerimientos necesarios para la ejecución de los componentes especificados en el descriptor de ensamblado y prepara y despliega los recursos necesarios para que la aplicación pueda ser puesta en marcha.

4.3. Implementación de Componentes

Con el objetivo de simplificar la implementación de componentes el CCM especifica un marco denominado CIF para la implementación de componentes que estandariza y automatiza la generación de la mayoría de las implementaciones del servidor. El CIF define un lenguaje llamado CIDL para definir como un componente debe ser usado.

El compilador CIDL genera automáticamente la implementación de la interfaz del componente del servidor pero deja al desarrollador de componentes la responsabilidad de generar la implementación específica de la aplicación a entidades llamadas *ejecutores*. En resumen el código generado automáticamente por el CIDL ayuda con las siguientes tareas a la implementación de componentes:

- Administra el ciclo de vida de la instancia y de los ejecutores del componente.
- Acepta la invocación de los clientes sobre las interfaces y consumidores de eventos y redirige estas invocaciones a los ejecutores correspondientes.
- Administra la conexión de la interfaz y la publicación de eventos que los consumidores pueden usar para invocar las operaciones.

76. [CCM2004], página 22.
77. [CCM2004], página 23.

El CIDL genera archivos descriptores de componentes que son documentos XML que contienen meta-datos que describen las características de los componentes, por ejemplo: que ports y atributos están disponibles, los requerimientos de ejecución, etc.

4.4. Composición y ensamblado de Componentes

La especificación del CCM estandariza la sintaxis y estructura de un meta-dato basado en XML llamado descriptor de ensamblado para la especificación del ensamblado de aplicaciones basadas en componentes. Un archivo descriptor de ensamblado esta compuesto por 3 elementos XML:

1. Implementaciones del componente: especifica las implementaciones del componente requeridas para el ensamblado de la aplicación. Un ensamblado de una aplicación puede usar distintas implementaciones del mismo componente para diferentes instancias del mismo.
2. Localizaciones del componente: especifica las restricciones de localización en las instalaciones de las instancias *home*[78] del componente.
3. Conexiones del componente: describe como el componente y las instancias home deben ser conectadas. Las conexiones son creadas por el paso de una referencia a un objeto.

Los descriptores de ensamblado del CCM proveen información clave acerca de cómo los componentes home y los componentes de usuario necesitan ser instanciados y como interoperan entre ellos y con otro software para formar una aplicación. Pero no contiene información acerca de la plataforma hardware, de la localización en la red, sistema operativo o lenguaje necesario para la instalación.

4.5. Empaquetado y Despliegue

La estandarización del empaquetado y el despliegue en CCM separa las cuestiones referidas a la distribución, instalación e ejecución de la aplicación de software de las cuestiones de diseño e implementación. Un *paquete de software* de CCM es un archivo comprimido que contiene una colección de implementaciones de software, se definen 2 tipos de paquetes:

- Paquete de componentes: contiene la información necesaria para realizar el despliegue de una implementación específica de un componente.
- Paquete de ensamblado de la aplicación: contiene la información necesaria para realizar el despliegue de una aplicación formada por un conjunto interconectado de componentes.

Los mecanismo de despliegue definidos por CCM especifican una serie de interfaces que posibilitan la ejecución de los componentes en distintas plataformas, que están físicamente dispuestos en los paquetes nombrados anteriormente.

4.6. CIAO

CIAO *(ver capítulo 9)* es una implementación de CCM liviano acorde a las especificaciones de la OMG de CORBA 3.0, con extensiones adicionales para el soporte de la Calidad de Servicio desarrollado bajo la modalidad de código abierto por la Universidad de Washington, Washington, EEUU. CIAO esta construido por los siguientes 4 bloques fundamentales[79]:

Fig.21 – Componentes de CIAO[80].

78. Home: del inglés, significa hogar, hace referencia al objeto que maneja el ciclo de vida del componente definido por el usuario.
79. [CCM2004], página 62.
80. [CCM2004], página 62.

1. Bibliotecas Principales: proveen la implementación de las interfaces definidas por la especificación de CCM.
2. Herramientas de Implementación: CIAO proporciona prototipos y especificaciones para la generación de código del CIDL.
3. Entorno de Ejecución: componentes para el entorno de ejecución.
4. Herramientas para el Despliegue de Componentes: un conjunto de herramientas en CIAO proveen la funcionalidad para interpretar los archivos de despliegue y desplegar los componentes de las aplicaciones en el ambiente de ejecución.

En la figura 21 se aprecia los principales componentes de CIAO y también los componentes en los que se basa: el ACE y el TAO, ambos son un conjunto de bibliotecas de código abierto desarrollados por la universidad de Washington, Washington, EEUU.
El conjunto de estos bloques forman la infraestructura de CIAO, juntos permiten a los usuarios definir las interfaces, generar el código de implementación y, proveen un entorno de ejecución para la implementación de los componentes.
En el CCM liviano hay dos categorías de componentes: (1) *componentes monolíticos*: archivos binarios ejecutables, y (2) *componentes ensamblados*: un conjunto de componentes interconectados, los cuales pueden ser componentes monolíticos o ensamblados.
Un *contenedor* provee el entorno de ejecución para los componentes que administra. Cada contenedor es responsable por la inicialización de las instancias de los componentes y gestiona el acceso a otros componentes y a los servicios del Middleware. Un componente *NodeApplication*[81] es un fabricador de servidores, crea contenedores y provee el contexto de ejecución para los demás componentes de la aplicación[82].

5. CoSMIC

5.1. Introducción

Con el objetivo de facilitar la construcción de aplicaciones distribuidas con soporte para la calidad del servicio, la Universidad de Vanderbilt, Nashville, EEUU, desarrolló un conjunto de herramientas de código abierto que implementan un paradigma de lenguajes de modelado específicos del dominio (DSML) del CCM, llamado CoSMIC *(ver capítulo 9)*. CoSMIC es una colección integrada de lenguajes de modelado de dominios específicos, y de herramientas de transformación que soportan el desarrollo, configuración, despliegue, y validación de sistemas distribuidos basados en componentes[83]. Todas las herramientas de CoSMIC fueron desarrollados utilizando el GME.
Las herramientas de diseño dirigido por modelos de CoSMIC permiten abarcar las principales fases del desarrollo de sistemas distribuidos en tiempo real:

1. Especificación e Implementación: en esta fase se define la granularidad del sistema, los componentes y las opciones de implementación.
2. Ensamblado y Empaquetado de Componentes: tiene como objetivo el armado de un conjunto de módulos binarios de software y meta-datos que representan como interactúan los componentes de la aplicación.
3. Configuración: implica la configuración de los parámetros del Middleware con el objetivo de satisfacer los requerimientos funcionales y no funcionales de la aplicación.
4. Planificación del Despliegue: toma de decisiones referidas al despliegue, esto incluye la identificación de las entidades –como ser los nodos, etc,- de la plataforma de destino en donde los paquetes software serán desplegados.
5. Despliegue y Puesta en Marcha: hace referencia al proceso de despliegue de los artefactos sobre las entidades de la plataforma de destino. Y colocar en un estado listo para ejecutar a los componentes instalados.
6. Análisis y Benchmarking[84]: realización de pruebas empíricas y análisis del rendimiento de la aplicación para validar la configuración de despliegue de los componentes y los requerimientos de la calidad de servicio de la aplicación.
7. Aseguramiento de la Calidad y Adaptación: reconfiguración del entorno de ejecución y administración de los recursos para asegurar la calidad del servicio de la aplicación en tiempo de diseño y de ejecución.

81. NodeApplication: del inglés, significa Aplicación de Nodo.
82. [CIAO2005], página 4.
83. [COSMIC2005], página 1.
84. Benchmarking: del inglés, hace referencia a la realización de pruebas y evaluación del rendimiento.

5.2. Especificación e Implementación

CoSMIC utiliza un metamodelo de GME para describir la especificación e implementación de CIAO. Utilizando el metamodelo del dominio específico del CCM se crean los componentes y archivos de configuración del CCM. CoSMIC provee un DSML para la especificación de las interfaces de los componentes llamado IDML. CoSMIC aún no ha implementado un DSML para modelar la implementación de los componentes.

El modelador de la aplicación creará el modelo, especificando los componentes y sus propiedades: las *facets*, los *receptacles*, los *events sources* y *sinks*. En este caso se utiliza el IDML para especificar las interfaces de CORBA 3.0 implementadas por CIAO. La figura 22 muestra el modelo del DSML de CCM generado en CoSMIC.

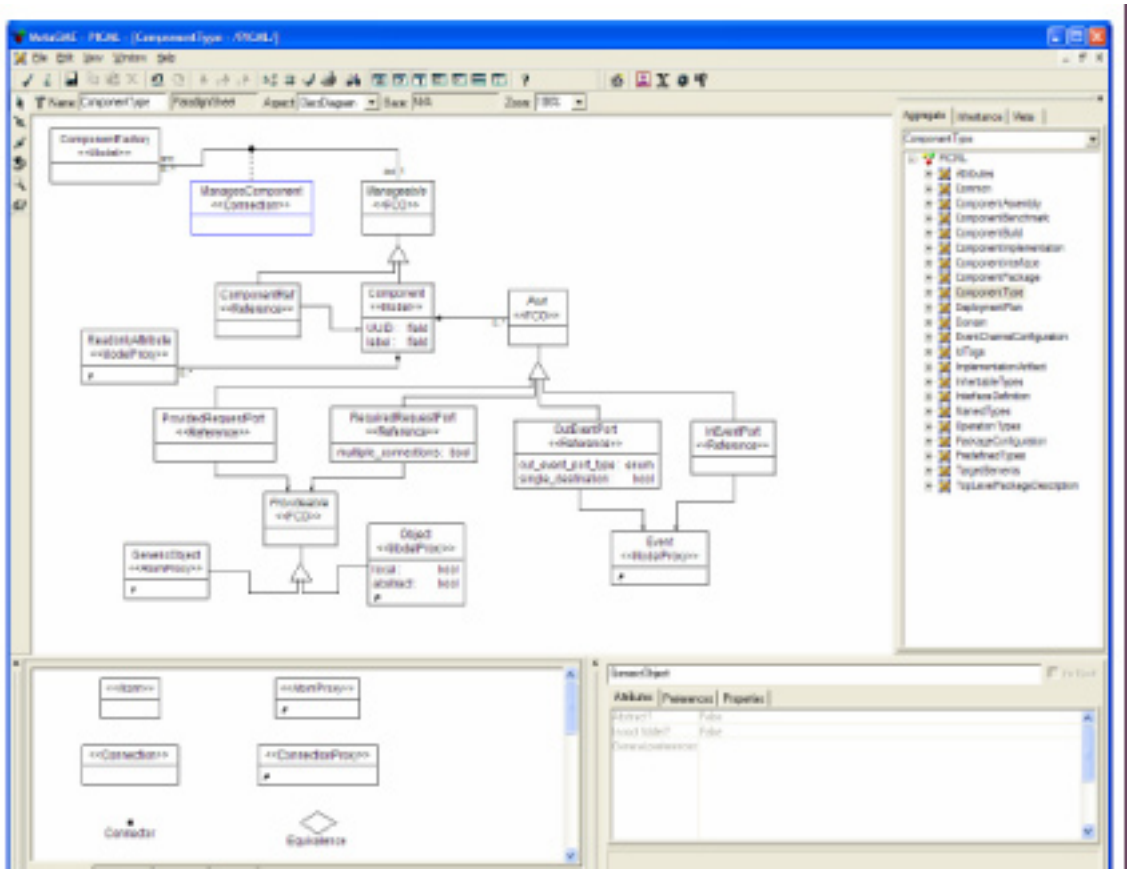


Fig. 22 – El metamodelo GME de CoSMIC[85].

5.3. Ensamblado y Empaquetamiento

En CCM los componentes de las aplicaciones y sus meta-datos asociados son ensamblados juntos dentro de un paquete. Un paquete puede contener más de un ensamblado con diferentes propiedades de calidad de servicio cada uno. Los sistemas distribuidos pueden tener una gran cantidad de ensamblados con cientos de componentes, esta característica provoca dos tipos de complejidad que deben ser manejadas[86]:

- Complejidad Inherente: implica la validación y verificación de la compatibilidad sintáctica y semántica.
- Complejidad Accidental: la cual se origina de la complejidad en manejar cientos de conexiones en los archivos XML.

El la figura 23 se puede observar el diagrama del ensamblado de la aplicación del caso de estudio.

85. [COSMIC2005-2], página 13
86. [COSMIC2005-2], página 14.

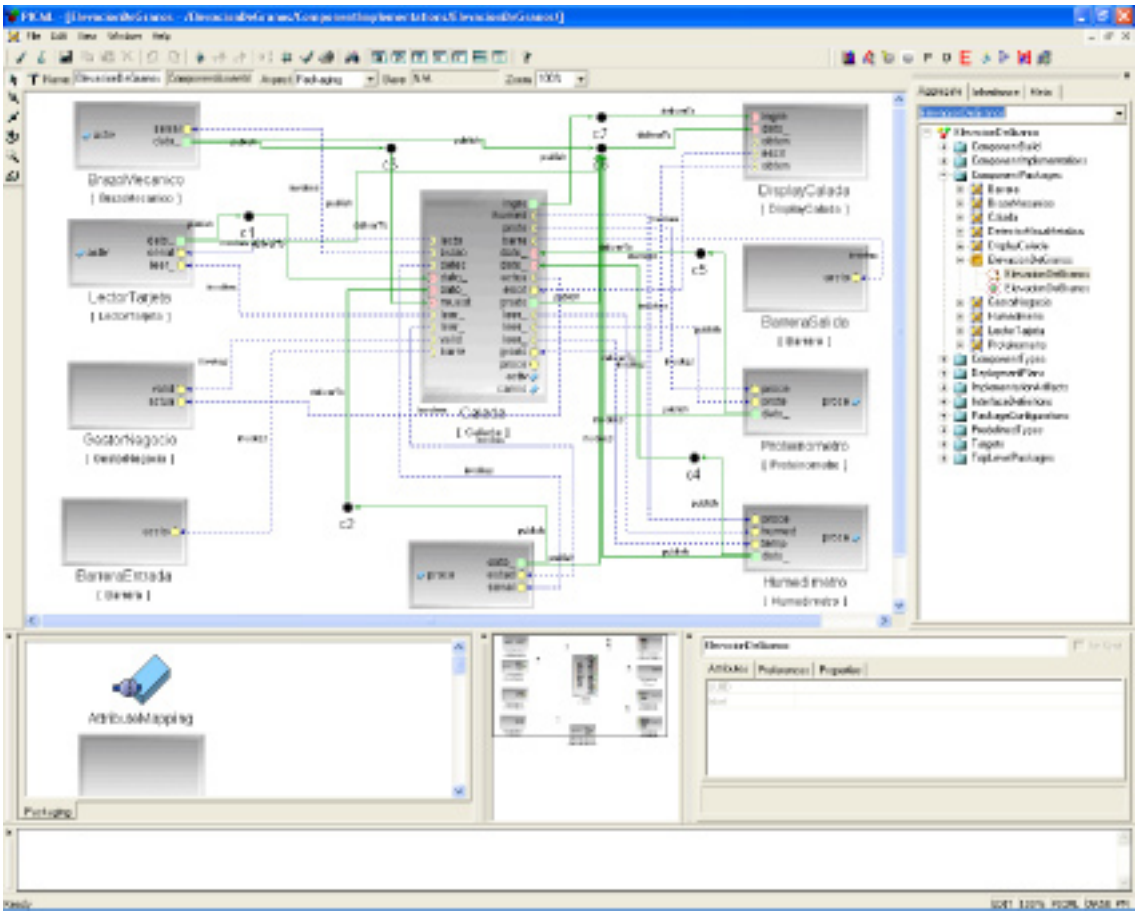


Fig. 23 – Un modelo de ensamblado de CCM.

Con el objetivo de ocuparse de estos problemas la Universidad de Washington desarrollo el PICML. El PICML es un DSML construido con el GME que facilita la realización de una gran cantidad de tareas complejas de la ingeniería de software, como son la visualización de múltiples aspectos, la manipulación de componentes y las interacciones con los subsistemas, la planificación del despliegue de los componentes, el modelado por jerarquías y la generación del ensamblado de componentes. El PICML incluye al IDML.

Las interfaces de los componentes son especificadas a través del IDL de CORBA 3.0, esta información es importada al PICML, de dos maneras distintas, o bien desde archivos IDL o sino desde la interfaz grafica del PCIML en GME directamente. Luego los modeladores pueden comenzar a interconectar los componentes en forma visual. Las reglas semánticas que determinan las conexiones válidas entre los componentes son forzadas durante el ensamblado de los componentes a través de restricciones definidas en el metamodelo del PICML.

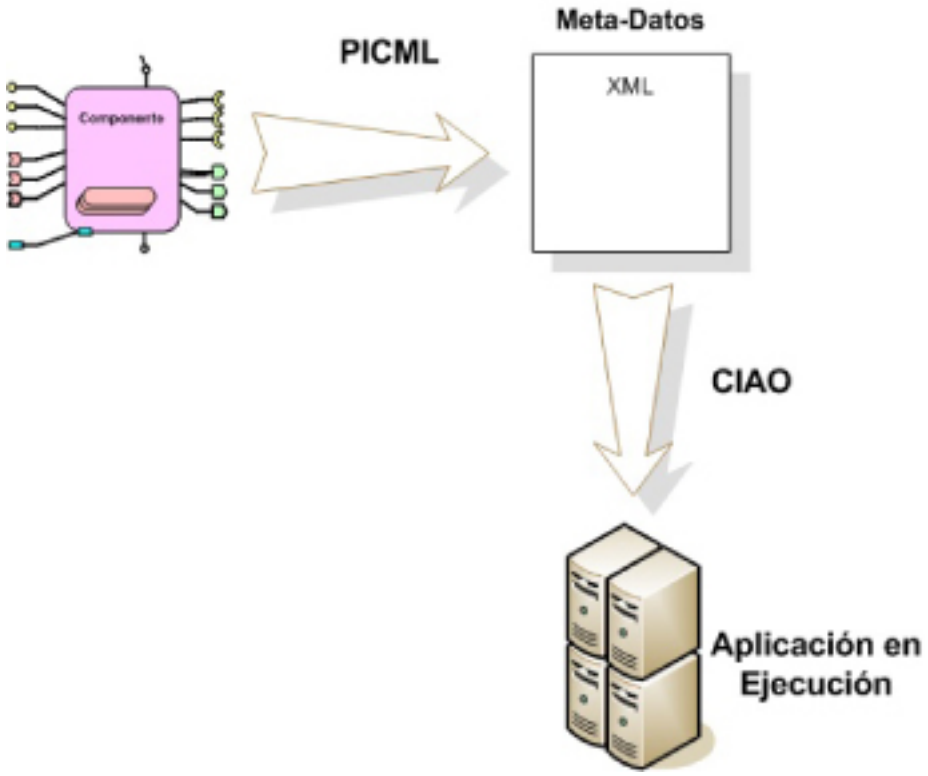


Fig. 24 – Actividades de la fase de empaquetamiento.

El PICML soporta la composición jerárquica del ensamblado de componentes dentro de grupos de mayor nivel de ensamblado y permite agrupar estos ensamblados en varios paquetes.
Una vez creados los componentes con el PICML, se ejecutan los intérpretes para generar los meta-datos necesarios para el despliegue de las aplicaciones CCM. Estos meta-datos son archivos XML, que luego son usados por el entorno de ejecución de CIAO para desplegar las aplicaciones y sus componentes. En la figura 24 se muestra el proceso descrito.

5.4. Configuración

Las aplicaciones distribuidas poseen un extenso y complejo proceso de configuración de sus componentes, interconexiones, servicios, etc. Todas estas alternativas pueden ser seleccionadas en más de un punto de la configuración.
Generalmente estos puntos de configuración ocurren durante: *el desarrollo de componentes*, donde son especificados estos valores por defecto; durante *la integración de la aplicación* en donde la configuración de los componentes puede ser sobre escrita por los valores específicos del dominio y; durante *el despliegue de la aplicación* en donde los valores específicos del dominio pueden ser sobre escritos por los valores actuales de la plataforma destino[87].
Con el objetivo de atender estas necesidades de configuración de una forma menos propensa a errores, se creo el OCML. El OCML es un DSML construido utilizando GME que simplifica la especificación y validación de la configuración de los sistemas distribuidos complejos. El PICML y el OCML permiten la especificación y configuración de aplicaciones a través de 3 capas como se aprecia en la figura 25.

87. [COSMIC2005-2], página 17.

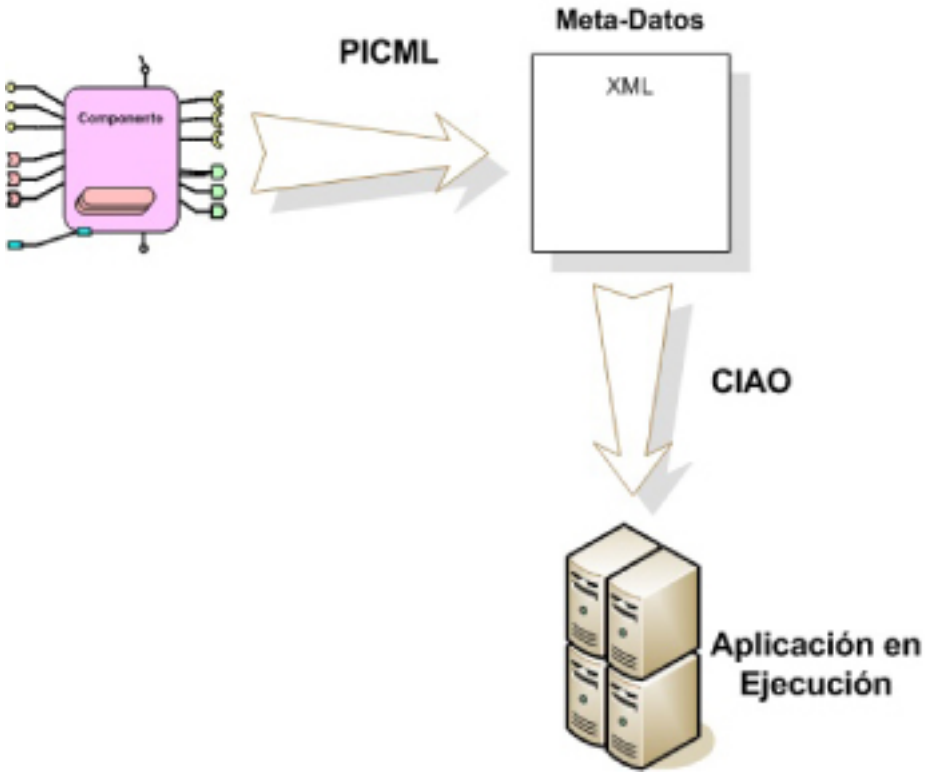


Fig.25 – El proceso de configuración con OCML[88].

La capa del metamodelo es donde se especifican las opciones de configuración del Middleware. Los desarrolladores usan el OCML para diseñar el modelo de opciones de CIAO. A su vez el OCML usa un modelo de opciones para generar los *configurator*[89] específicos de CIAO.
La capa del modelo es en donde los desarrolladores usan el configurator generado por el OCML para configurar el Middleware acorde a las necesidades específicas de las aplicaciones.
La capa de aplicación realiza la configuración en tiempo de ejecución de la aplicación. El configurator específico de CIAO genera los archivos de configuración de cada componente del CCM liviano en la forma de archivos del servicio de configuración del ACE.

5.5. Planificación del Despliegue

La fase de planificación es en donde los integradores de componentes deben tomar las decisiones de despliegue, incluyendo la identificación de nodos de la plataforma destino en donde los paquetes ensamblados serán desplegados[90]. Las actividades de esta fase incluyen: (1) el empaquetado de los componentes, (2) la especificación de las plataformas de destino y (3) la asignación de recursos en la plataforma de destino. Todas estas decisiones son especificadas en un archivo XML que contiene el mapeo de la configuración de la aplicación al dominio, incluyendo el mapeo de los componentes monolíticos a los nodos, conectores a interconexiones y requerimientos a recursos.
CoSMIC provee el MIDCESS y la herramienta CCM Perf para resolver el problema de la planificación del despliegue. MIDCESS es usado para especificar el entorno de destino elegido y realizar el despliegue de los paquetes de ensamblado. En la figura 26 se observan los elementos que constituyen el plan de despliegue de CIAO.

88. [COSMIC2005-2], página 19.
89. Configurator: archivo de configuración utilizado por el framework del ACE: *Servicio de Configuración* que permite la configuración en tiempo de ejecución de las aplicaciones.
90. [COSMIC2005-2], página 21

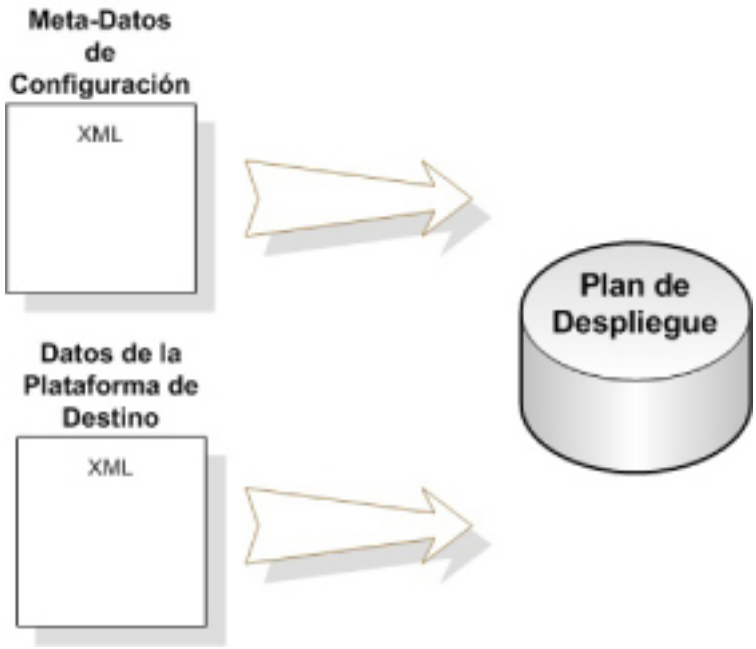


Fig.26 –Planificación del Despliegue.

5.6. Despliegue y Puesta en Marcha

El despliegue y puesta en marcha de la aplicación se realiza a través de una infraestructura de ejecución llamada DAnCE la cual implementa la especificación de Despliegue y Configuración de la OMG para el CCM, que permite modelar el plan de despliegue de los paquetes y provee un conjunto de componentes que automatizan la puesta en marcha de los componentes sobre los nodos de la aplicación distribuida.

5.7. Análisis y Benchmarking

El principal desafió de la fase de análisis y benchmarking es asegurar que la configuración elegida entregue la calidad de servicio requerida. Para satisfacer este requerimiento de calidad de servicio se desarrollo el BGML. El BGML es un DSML creado en el GME que sintetiza un conjunto de pruebas para evaluar las prestaciones de las aplicaciones con el objetivo de analizar la calidad de servicio de los sistemas distribuidos configurados con OCML.
El BGML se desarrolla a lo largo de un proceso de 4 fases.[91] Las cuales se muestran en la figura 27.

Fig.27 – Vista del proceso de BGML[92].

91. [COSMIC2005-2], página 22.
92. [COSMIC2005-2], página 22.

1. Primero un ingeniero de pruebas usa el PICML para representar el escenario de prueba de la aplicación en forma visual.
2. Luego se asocian las propiedades de la calidad de servicio con el escenario de prueba de la aplicación a evaluar.
3. En el tercer paso el BGML interpreta el modelo y genera el código de pruebas para ejecutar, y por último obtiene las métricas de calidad obtenidas en las pruebas.
4. El último paso consiste en comparar las mediciones obtenidas con los resultados deseados para verificar la calidad de servicio requerida.

5.8. Aseguramiento de la Calidad y Adaptación

Actualmente CoSMIC no implementa ningún DSML para modelar el aseguramiento de la calidad en tiempo de diseño o tiempo de ejecución de CCM.

6. Caso de estudio

Con el objetivo de evaluar el desarrollo dirigido por modelos se presenta un caso de estudio en donde se aplicará el paradigma del MIC para desarrollar un prototipo de una aplicación del mundo real en base a una existente, construida usando el paradigma orientado a objetos y lenguajes de tercera generación[93]. Se usará el conocimiento y la información obtenida del desarrollo de la aplicación actual para realizar una comparación del enfoque tradicional de desarrollo orientado a objetos contra el enfoque presentado.

6.1. Definición del problema

Se quiere desarrollar una aplicación –aplicando el MIC- que controle el puesto de la Calada *(ver capítulo 6.1.2)* de una planta de elevación de granos.
En las plantas de elevación de granos se realiza el proceso de exportación del cereal. Es el lugar en donde los camiones descargan cereal -se acopia en los silos o depósitos- y se carga en los buques por medio de balanzas de embarque, están situados en los puertos. El promedio de camiones de las plantas en época de cosecha es de 1000 camiones/día. El proceso de la planta se desarrolla a través de un circuito con una serie de puestos por los cuales debe pasar el camión para descargar el cereal. Primero el camión se debe identificar en Mesa de Entrada, en donde se le entrega una tarjeta -TAG- inteligente que le servirá para identificarse en forma automática en los demás puestos. Una vez registrado se analiza la mercadería para determinar la calidad de la misma, si la calidad es la requerida según el contrato comercial o esta dentro de los parámetros de aceptación de la planta, se obtiene el peso bruto del camión, ahí se le informa al chofer la plataforma de descarga en función de la mercadería y de la calidad determinada.
Luego se dirige a la plataforma de descarga en donde es elevado para vaciar su contenido, una vez descargado se obtiene el peso tara del camión para calcular el peso de la mercadería neta descargada, por último el camión pasa por la Mesa de Salida en donde se le entrega la documentación de la recepción de la mercadería y el chofer deposita la tarjeta inteligente en un buzón. Y el circuito finaliza.
La solución requiere interactuar con dispositivos de acceso –barreras y lectores de tarjetas inteligentes-, dispositivos de medición –determinan la calidad o grado[94] del cereal-, terminales para la captura de datos, un sistema de gestión donde reside la lógica de negocio y un repositorio único en donde se almacenan los datos para que luego los procese el sistema comercial.

6.1.1. Alcance del prototipo

Se aclara que el prototipo a desarrollar se limitará solo a la funcionalidad necesaria para llevar a cabo el proceso de la Calada. Por ende los componentes a desarrollar son los componentes que implementan la Calada, y solo algunos componentes genéricos a todo el proceso necesarios para implementar el proceso mencionado.

93. Lenguaje de tercera generación: Es un lenguaje con construcciones similares al lenguaje natural (típicamente al inglés) que permite crear programas (aplicaciones) complejas y relativamente sencillas de mantener y modificar. Un programa en un lenguaje de alto nivel no es entendible directamente por un computador.
94. Grado del Cereal: el grado del cereal es una escala numérica que determina la calidad del mismo en base a cálculos realizados sobre las características del grano, como ser la humedad, las proteínas, el porcentaje de rotura, etc. Dicha escala la genera la ONCCA que es el ente regulatorio de cereales de la R.A. y esta dada en función de cada tipo de cereal.

6.1.2. Los Requerimientos de la Calada

En el puesto de Calada se realiza el análisis de la calidad del cereal de los camiones. Es el puesto más crítico, pues es en donde se determina la calidad y por ende el precio a pagar por la empresa al proveedor en concepto de la mercadería a comprar.
El proceso se desarrolla de la siguiente manera: los camiones hacen una fila para entrar al galpón en donde se realiza la calada, cuando al camión le toca el turno, primero se identifica aproximando el TAG al lector, el sistema lo valida y levanta la barrera para que avance el camión, luego el sistema encuesta al detector de masa metálica para determinar cuando el camión se detiene para bajar la barrera, luego un operario comprueba si el cereal contiene bichos realizando un control ocular, mientras tanto el sistema mueve un brazo mecánico y toma tres muestras del cereal en distintos lugares del camión; luego el calador[95] deposita las muestras en una bandeja y realiza un visteo[96] de los granos, aquí el calador ingresa los rubros[97] del cereal (grado de rotura, grado de quiebre, etc.); y deposita una porción de la muestra en un humedímetro y en un proteinómetro los cuales miden la humedad, la temperatura y las proteínas del cereal respectivamente.
Con esta información el sistema determina el grado del cereal y en consecuencia cuanto se le pagará al vendedor por su mercadería. En el lugar se pueden calar hasta dos camiones simultáneamente, por lo que se requieren dos terminales para la captura y visualización de los datos. Con el objetivo de simplificar la construcción del prototipo solo se implementará una sola terminal.

6.1.3. La Arquitectura Actual

La aplicación actual se encuentra desarrolla en C# y tiene la siguiente arquitectura:

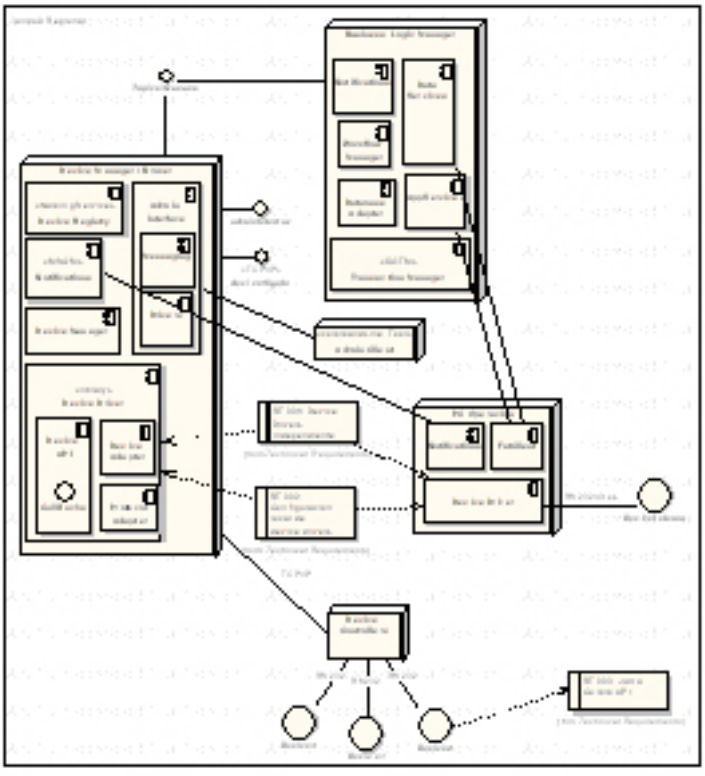


Fig.28 – Arquitectura actual.

Como se puede apreciar por la cantidad de componentes que integran la aplicación más el contexto distribuido en donde se despliegan, ésta es una aplicación con una alta complejidad. El prototipo presentado en este trabajo implementa la funcionalidad requerida por la Calada, sin embargo el desarrollo de esta funcionalidad implica la construcción de otros componentes de software que implementen los diferentes servicios de: mensajería asincrónica, invocación remota de objetos, interconexión de componentes, transacciones, etc.
La utilización de una plataforma Middleware como CCM nos facilita el acceso a una cantidad de servicios que reducen la complejidad y los tiempos de desarrollo de la aplicación.

95. Calador: es el operario encargado de realizar el visteo al grano de cereal, debe ser un perito en granos.
96. Visteo: proceso que efectúa el perito en granos con el objetivo de determinar el estado del cereal en forma ocular.
97. Rubro: atributo del grano.

6.1.3.1. Métricas

A continuación se presentan una serie de métricas de la aplicación actual y otras especificas del modulo funcional que se presenta en este trabajo. Tienen como objetivo informar acerca de las características y magnitud de la aplicación actual.

Líneas de Código	Archivos	Líneas/ Archivo	Máxima Complejidad	Complejidad Promedio	Sentencias	Clases
150356	459	326	12	0,8	105960	468

Tabla 1

Puntos de función[98] del subprograma Calada:

Funciones	Puntaje
Archivos Lógicos Internos	15
Archivos de Interfaz Externa	7
Entradas Externas	6
Salidas Externas	7
Consultas Externas	3
Grados de Influencia	
Total	48
Puntos por Función	
Sin ajustar	38
Factor de valor de ajuste	1,13
Puntos Totales	**42,94**

Tabla 2

Grados de Influencia	Puntaje
Comunicaciones	4
Procesamiento de Datos Distribuido	4
Performance	4
Configuración	4
Carga Transaccional	4
Carga de Datos On-Line	5
Facilidad de la Interfaz de Usuario	3
Actualización On-Line	3
Procesamiento Complejo	1
Reusabilidad	3
Facilidad de Instalación	2
Facilidad de Operación	3
Múltiples Sitios	3
Facilidad de Cambios	5
Total	**48**

Tabla 3

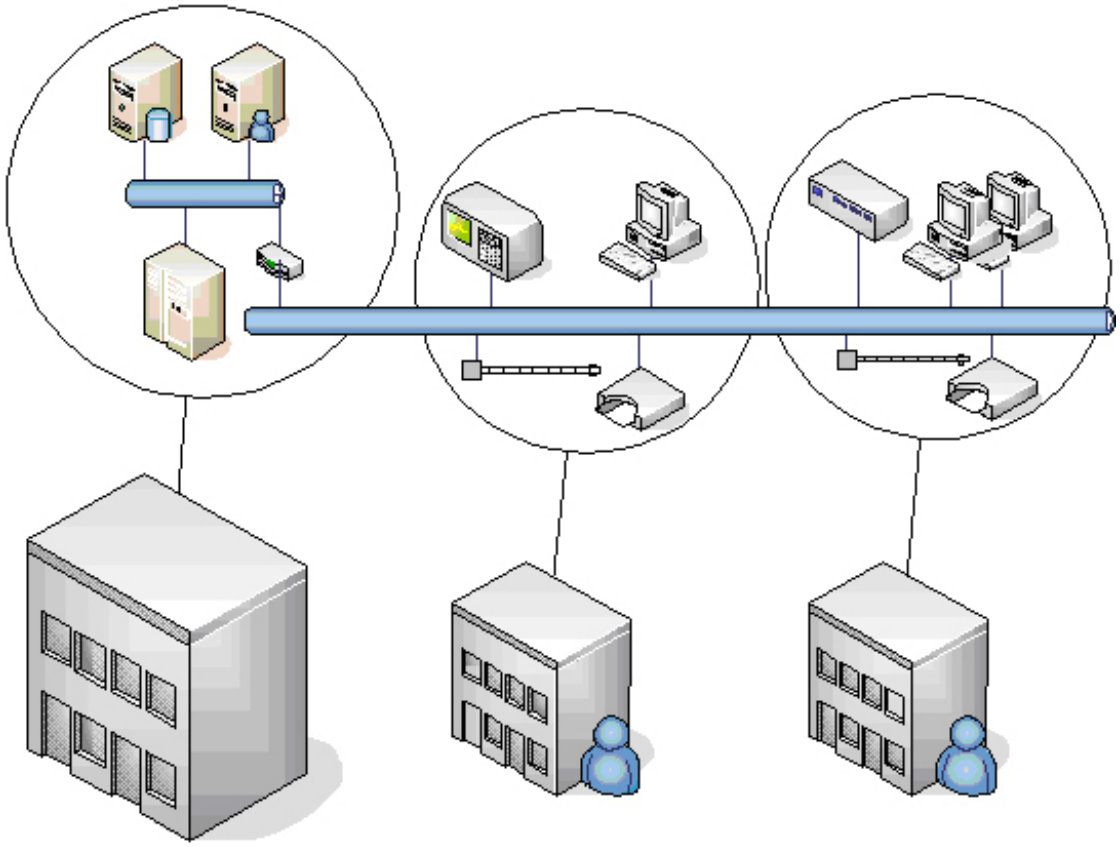
98. Se utilizó el estándar definido por el International Function Point Users Group (IFPUG) para realizar el cálculo de los puntos de función.

6.1.4. La Arquitectura del Prototipo

La arquitectura física de la aplicación esta compuesta por los siguientes nodos, un nodo como servidor de dispositivos, un nodo como servidor de aplicaciones, un nodo como repositorio único y nodos para la captura y visualización de los datos situados físicamente en cada uno de los puestos. Esta arquitectura entra dentro de la definición de un sistema distribuido de tiempo real blando.

Es un sistema distribuido por lo expuesto en 2.1.1 y de tiempo real de acuerdo a la siguiente definición encontrada en [QINGYAO2003]: “Son los sistemas en donde la exactitud de la salida del sistema depende de tanto de la exactitud de los requerimientos funcionales como de la exactitud en la ejecución a tiempo de las tareas”. Y un sistema real blando: “Es un sistema de tiempo real en donde existen restricciones de tiempo en la ejecución de la tareas pero tiene un mayor grado de flexibilidad. Una violación de tiempo no da como resultado una falla del sistema pero si una degradación del mismo”.

En la figura 29 se muestra un diagrama en donde se describe la arquitectura física actual del sistema, ahí se puede apreciar en centro de servidores en donde residen el servidor de aplicaciones, de dispositivos y el repositorio, luego en cada puesto físico de la planta se encuentran los dispositivos de acceso, -y demás tipo dependiendo el puesto- y las terminales de captura y visualización de datos.



La arquitectura lógica de la solución se compone de los siguientes subsistemas basados en componentes del CCM:

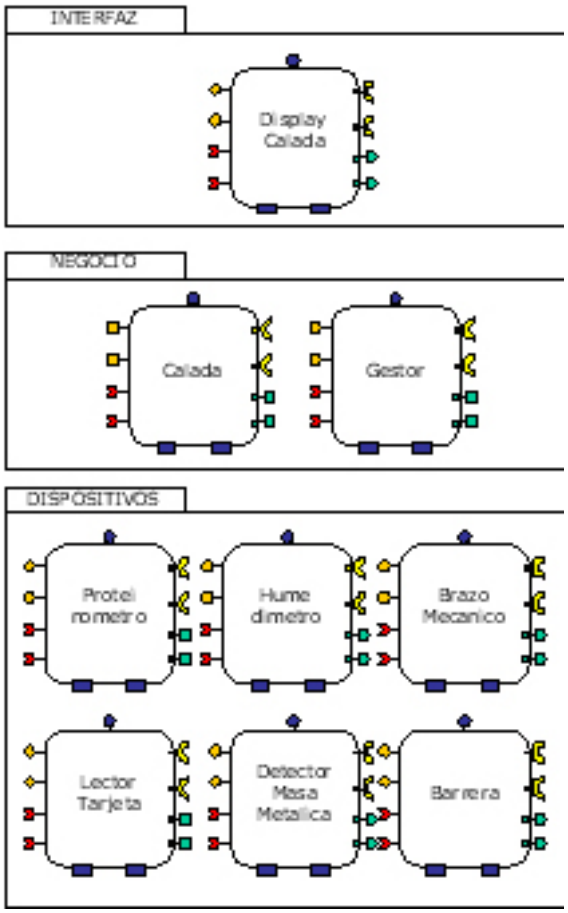


Fig.30 – Arquitectura Lógica.

En donde algunos de los componentes exponen interfaces para los actores[99] del sistema y otros componentes encapsulan la funcionalidad de los dispositivos de hardware con los que debe interactuar la solución.

6.2. Desarrollo del prototipo

Vamos a aplicar el proceso del ciclo de vida de desarrollo soportado por CoSMIC *(ver capítulo 5)* para la construcción de la solución, debido a la restricción actual de CoSMIC, la ultima fase “Aseguramiento de la Calidad y Adaptación” no va a ser implementada.
Al final de este capitulo se analizarán las diferencias entre el enfoque MIC y el desarrollo orientado a objetos tradicional.

6.3. Fase 1: Especificación e Implementación

El primer paso es definir en GME las interfaces de los componentes a construir. Para eso creamos un proyecto nuevo en GME utilizando el paradigma PICML llamado “ElevacionDeGranos”, paso siguiente en el explorador de agregaciones insertamos una nueva carpeta llamada “InterfaceDefinitions” en donde definiremos todas las interfaces de los componentes de la solución.

99. Actor: persona, sistema o dispositivo que interactúa con el sistema de alguna forma.

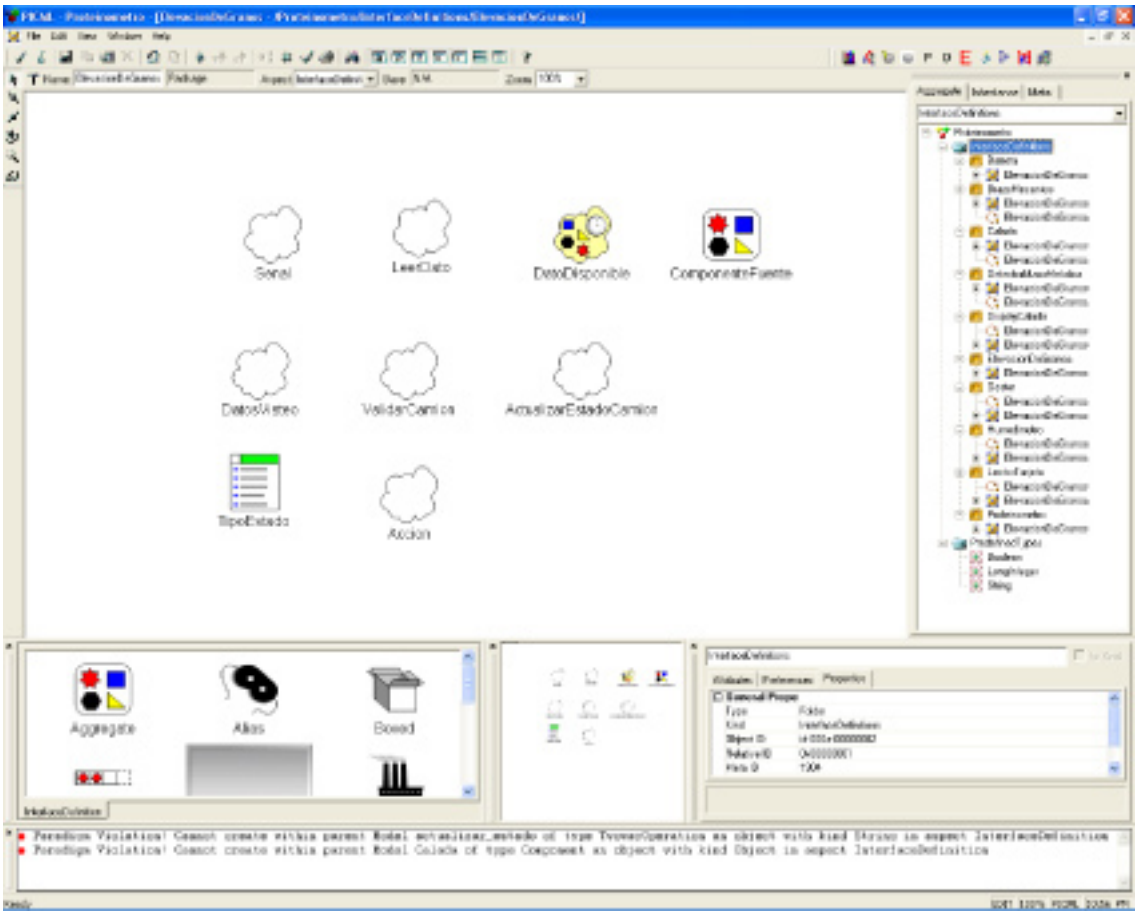
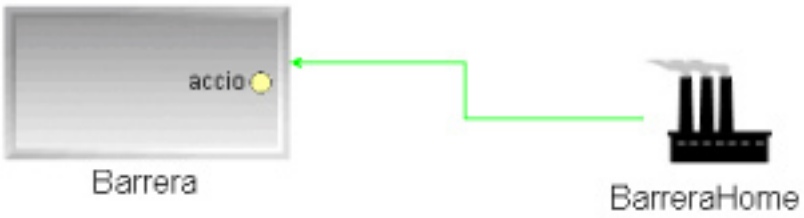


Fig.31 – Vista de las interfaces creadas en GME.

Los componentes creados son los siguientes:

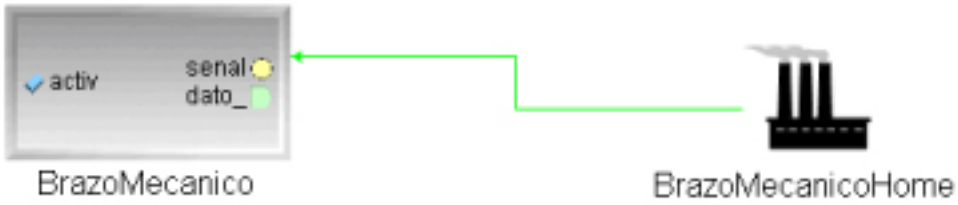
- **Barrera**: este componente modela el comportamiento de una barrera.



El archivo generado a partir del intérprete IDL es el siguiente:

```
#ifndef BARRERA_IDL
#define BARRERA_IDL

#include <Components.idl>
#include "ElevacionDeGranos.idl"

module ElevacionDeGranos
{
  component Barrera
  {
    provides ElevacionDeGranos::Accion accion;
  };

  home BarreraHome
    manages Barrera
  {
  };
};

#endif // BARRERA_IDL
```

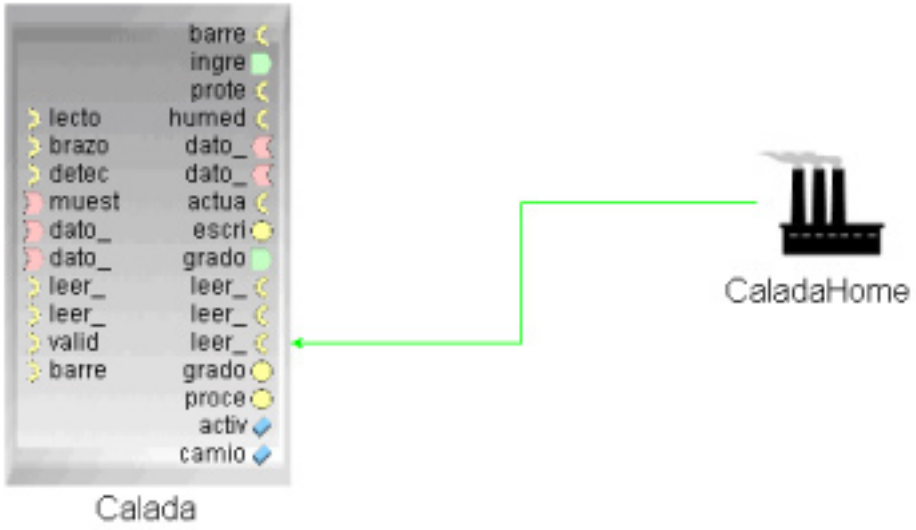
• **BrazoMecanico**: este componente modela el comportamiento de un brazo mecánico que toma muestras de cereal.



El archivo generado a partir del intérprete IDL es el siguiente:

```
#ifndef BRAZOMECANICO_IDL
#define BRAZOMECANICO_IDL

#include <Components.idl>
#include "ElevacionDeGranos.idl"

module ElevacionDeGranos
{
  component BrazoMecanico
  {
   publishes ElevacionDeGranos::DatoDisponible dato_
disponible;

    provides ElevacionDeGranos::Senal senal;

    attribute boolean activo;
  };

  home BrazoMecanicoHome
    manages BrazoMecanico
  {
  };
};

#endif // BRAZOMECANICO_IDL
```

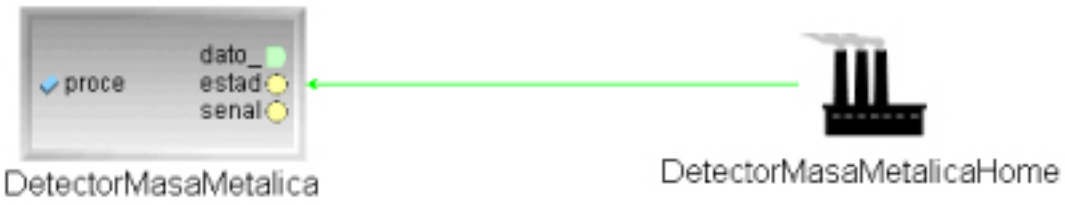
- **Calada**: este componente modela la gestión del puesto de Calada.



El archivo generado a partir del intérprete IDL es el siguiente:

```
#ifndef CALADA_IDL
#define CALADA_IDL
#include <Components.idl>
#include "ElevacionDeGranos.idl"
module ElevacionDeGranos
{
  component Calada
  {
    consumes ElevacionDeGranos::DatoDisponible dato_proteinas;
    consumes ElevacionDeGranos::DatoDisponible dato_humedad;
  consumes ElevacionDeGranos::DatoDisponible muestra_disponible;
    consumes ElevacionDeGranos::DatoDisponible dato_masa;
    consumes ElevacionDeGranos::DatoDisponible dato_tarjeta;
   publishes ElevacionDeGranos::DatoDisponible grado_calculado;
   publishes ElevacionDeGranos::DatoDisponible ingresar_rubros;
    provides ElevacionDeGranos::LeerDato grado;
    provides ElevacionDeGranos::DatosVisteo escribir_rubros;
    provides ElevacionDeGranos::Senal proceso;
    uses ElevacionDeGranos::Accion barrera_salida;
    uses ElevacionDeGranos::Accion barrera_entrada;
    uses ElevacionDeGranos::ActualizarEstadoCamion actualizar_estado;
    uses ElevacionDeGranos::ValidarCamion validar_camion;
    uses ElevacionDeGranos::LeerDato leer_proteinas;
    uses ElevacionDeGranos::LeerDato leer_temp;
    uses ElevacionDeGranos::Senal proteinometro;
    uses ElevacionDeGranos::LeerDato leer_humedad;
    uses ElevacionDeGranos::Senal humedimetro;
    uses ElevacionDeGranos::Senal brazo_mecanico;
    uses ElevacionDeGranos::LeerDato leer_masa;
    uses ElevacionDeGranos::Senal detector_masa;
    uses ElevacionDeGranos::LeerDato leer_tag;
    uses ElevacionDeGranos::Senal lector_tarjeta;
    attribute boolean activo;
    attribute boolean camion_en_proceso;
  };
  home CaladaHome
    manages Calada
  {
  };
};

#endif // CALADA_IDL
```

- **DetectorMasaMetalica**: este componente modela el comportamiento de un detector de masa metálica.

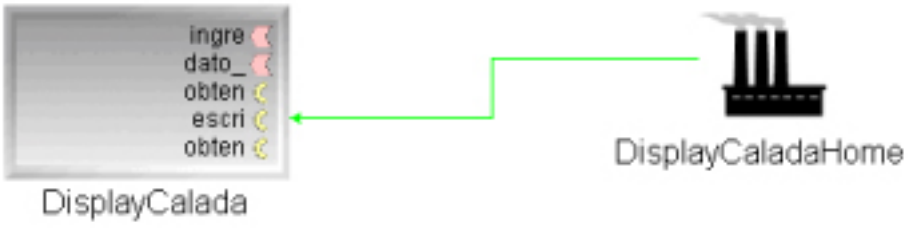


El archivo generado a partir del intérprete IDL es el siguiente:

```
#ifndef DETECTORMASAMETALICA_IDL
#define DETECTORMASAMETALICA_IDL

#include <Components.idl>
#include "ElevacionDeGranos.idl"

module ElevacionDeGranos
{
  component DetectorMasaMetalica
  {
   publishes ElevacionDeGranos::DatoDisponible dato_
disponible;

    provides ElevacionDeGranos::LeerDato estado;

    provides ElevacionDeGranos::Senal senal;

    attribute boolean procesando;
  };

  home DetectorMasaMetalicaHome
    manages DetectorMasaMetalica
  {
  };
};

#endif // DETECTORMASAMETALICA_IDL
```

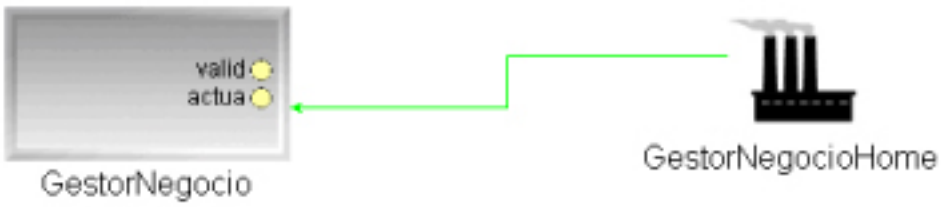
• **DisplayCalada**: este componente modela la terminal de ingreso y visualización de datos del puesto de Calada.

El archivo generado a partir del intérprete IDL es el siguiente:



```
#ifndef DISPLAYCALADA_IDL
#define DISPLAYCALADA_IDL

#include <Components.idl>
#include "ElevacionDeGranos.idl"

module ElevacionDeGranos
{
  component DisplayCalada
  {
     consumes ElevacionDeGranos::DatoDisponible ingresar_rubros;

    consumes ElevacionDeGranos::DatoDisponible dato_disponible;

    uses ElevacionDeGranos::LeerDato obtener_grado;

    uses ElevacionDeGranos::DatosVisteo escribir_rubros;

    uses ElevacionDeGranos::LeerDato obtener_dato;
  };

  home DisplayCaladaHome
    manages DisplayCalada
  {
  };
};

#endif // DISPLAYCALADA_IDL
```

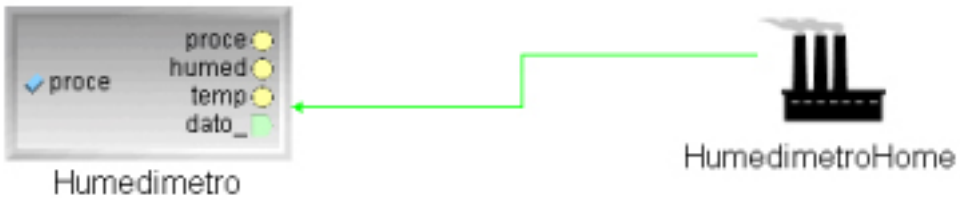
• **GestorNegocio**: este componente modela la gestión de negocio de la Planta de Elevación de Granos.



El archivo generado a partir del intérprete IDL es el siguiente:

```
#ifndef GESTORNEGOCIO_IDL
#define GESTORNEGOCIO_IDL

#include <Components.idl>
#include "ElevacionDeGranos.idl"

module ElevacionDeGranos
{
  component GestorNegocio
  {
    provides ElevacionDeGranos::ActualizarEstadoCamion
actualizar_estado_camion;

      provides ElevacionDeGranos::ValidarCamion vali-
dar_camion;
  };

  home GestorNegocioHome
    manages GestorNegocio
  {
  };
};

#endif // GESTORNEGOCIO_IDL
```

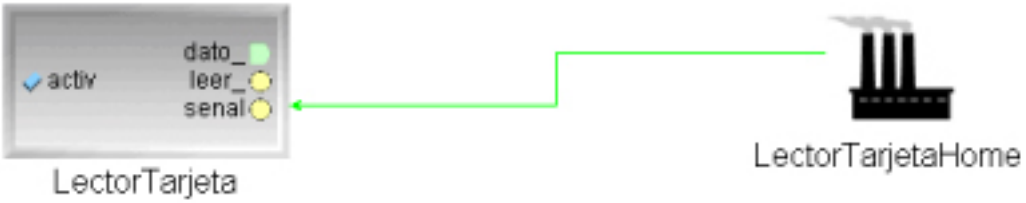
- **Humedimetro**: este componente modela el comportamiento del humedímetro.



El archivo generado a partir del intérprete IDL es el siguiente:

```
#ifndef HUMEDIMETRO_IDL
#define HUMEDIMETRO_IDL

#include <Components.idl>
#include "ElevacionDeGranos.idl"

module ElevacionDeGranos
{
  component Humedimetro
  {
   publishes ElevacionDeGranos::DatoDisponible dato_
disponible;

    provides ElevacionDeGranos::LeerDato temp;

    provides ElevacionDeGranos::LeerDato humedad;

    provides ElevacionDeGranos::Senal procesar;

    attribute boolean procesando;
  };

  home HumedimetroHome
    manages Humedimetro
  {
  };
};

#endif // HUMEDIMETRO_IDL
```

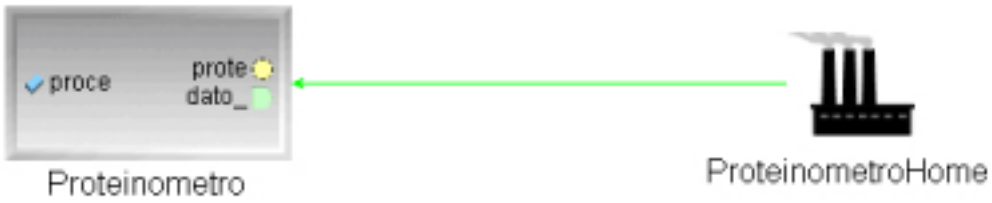
- **LectorTarjeta**: este componente modela el comportamiento del lector de tarjetas de proximidad.



El archivo generado a partir del intérprete IDL es el siguiente:

```
#ifndef LECTORTARJETA_IDL
#define LECTORTARJETA_IDL

#include <Components.idl>
#include "ElevacionDeGranos.idl"

module ElevacionDeGranos
{
  component LectorTarjeta
  {
   publishes ElevacionDeGranos::DatoDisponible dato_
disponible;

    provides ElevacionDeGranos::LeerDato leer_dato;

    provides ElevacionDeGranos::Senal senal;

    attribute boolean activo;
  };

  home LectorTarjetaHome
    manages LectorTarjeta
  {
  };
};

#endif // LECTORTARJETA_IDL
```

- **Proteinometro**: este componente modela el comportamiento del proteinómetro.



El archivo generado a partir del intérprete IDL es el siguiente:

```
#ifndef PROTEINOMETRO_IDL
#define PROTEINOMETRO_IDL

#include <Components.idl>
#include "ElevacionDeGranos.idl"

module ElevacionDeGranos
{
  component Proteinometro
  {
   publishes ElevacionDeGranos::DatoDisponible dato_
disponible;

    provides ElevacionDeGranos::LeerDato proteinas;

    attribute boolean procesando;
  };

  home ProteinometroHome
    manages Proteinometro
  {
  };
};

#endif // PROTEINOMETRO_IDL
```

6.4. PASO 2: Ensamblado y Empaquetamiento

El segundo paso Implica la definición de paquetes de módulos binarios de software a través de meta-datos. Los meta-datos definen como se interconectan los ports de los componentes, como son ensamblados y empaquetados.

Dentro de un modelo "ImplementationArtifacts", se definen para cada componente los artefactos CMM, sus bibliotecas, dependencias y archivos de configuración necesarios para la ejecución de los componentes en la plataforma seleccionada.

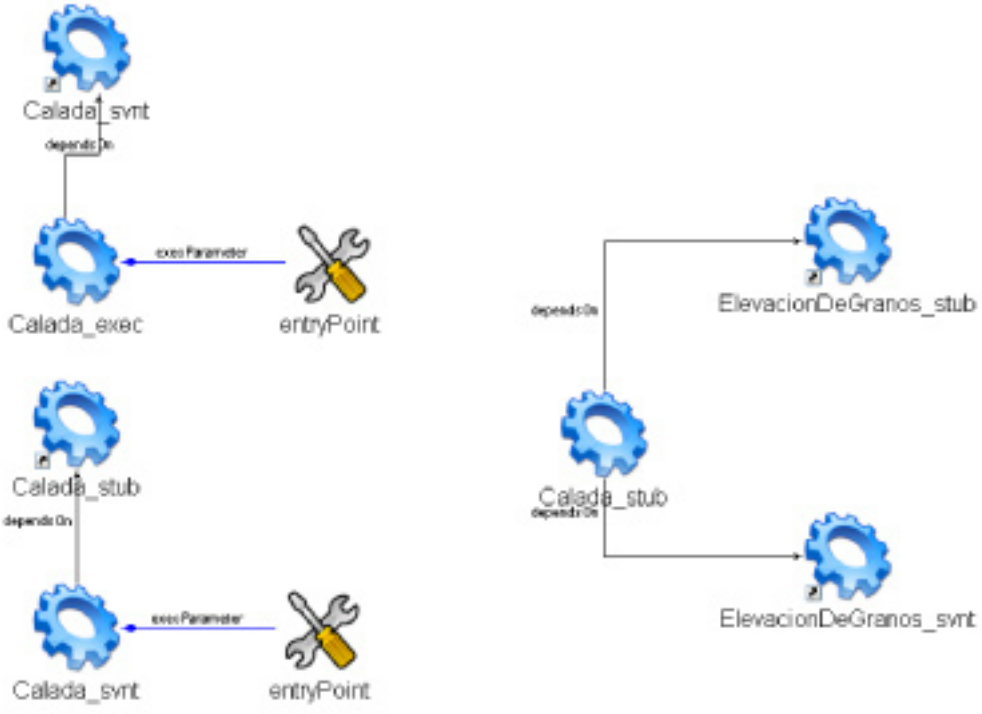


Fig.32 – Artefactos de implementación para el componente Calada.

En la figura 32 se muestra el modelo de implementación para el componente Calada, el cual se componen de tres bibliotecas: el servidor de ejecución, el stub[100] y el servant[101]. También se especifica los parámetros de configuración del servidor de ejecución a través del objeto entryPoint.
La definición de la implementación de los componentes se especifica en un modelo “ComponentImplementations” y el ensamblado en un modelo “ComponentAssembly”, allí es donde definimos las conexiones entre los ports de los componentes del modelo.
El ensamblado de la solución se muestra en la figura 33, el componente principal es el componente Calada, el cual gestiona todo el proceso de Calada de camiones.
El proceso se inicia al invocar el facet **proceso** del componente Calada, una vez invocado **proceso**, el componente invoca el facet **senal** del componente LectorTarjeta a través del receptable **lector_tarjeta** de Calada. Cuando LectorTarjeta detecta una tarjeta genera un evento **dato_disponible** que esta conectado al sink de Calada **dato_tarjeta**, la recepción de este evento genera una llamada al facet **leer_data** de LectorTarjeta que devuelve el ID de la tarjeta leída.
Una vez obtenida el ID de la tarjeta, se realiza una llamada al facet **validar_camion** con el ID de la tarjeta como parámetro y este devuelve 1 si el ID del camión es valido o cero en caso contrario. SI el camión es validado se procede a abrir la barrera de entrada llamando el facet **accion** de BarreraEntrada. Luego se activa el dispositivo detector de masa de metálica llamando al facet **senal** de DetectorMasaMetalica para detectar cuando el camión paso completamente por la barrera, DetectorMasaMetalica envía un evento **dato_disponible** que esta conectado al sink de Calada **dato_masa**, recibido el evento Calada llama al facet **accion** de BarreraEntrada para bajar la barrera.
Cuando DetectorMasaMetalica envía el evento también indica que el camión esta posicionado para la obtención de las muestras por intermedio del brazo mecánico. El componente Calada llama al facet **senal** del componente BrazoMecanico, BrazoMecanico envía un evento **dato_disponible** que esta conectado al sink de Calada **muestra_disponible**.
Con las muestras ya obtenidas del camión se procede a medir las características del cereal, para eso se invoca a los facets **procesar** de los componentes Proteinometro y Humedimetro, que generan un evento **dato_disponible** cuando terminan de analizar los granos. Recibido el evento el componente Calada llama a los facets correspondientes en los componentes Proteinometro y Humedimetro para obtener los valores detectados.

100. Stub: esqueleto común a todas las implementación de los contenedores CCM.
101. Servant: componente que instancia los componentes en el servidor.

Por último se ingresan los datos obtenidos en el componente GestorNegocio llamando al facet **actualizar_estado_camion**, se levanta la barrera de salida y se espera recibir nuevamente el evento desde el componente DetectorMasaMetalica con el cambio de estado (es decir no detecta masa metálica) que indica que el camión abandonó la plataforma, y se debe cerrar la barrera.

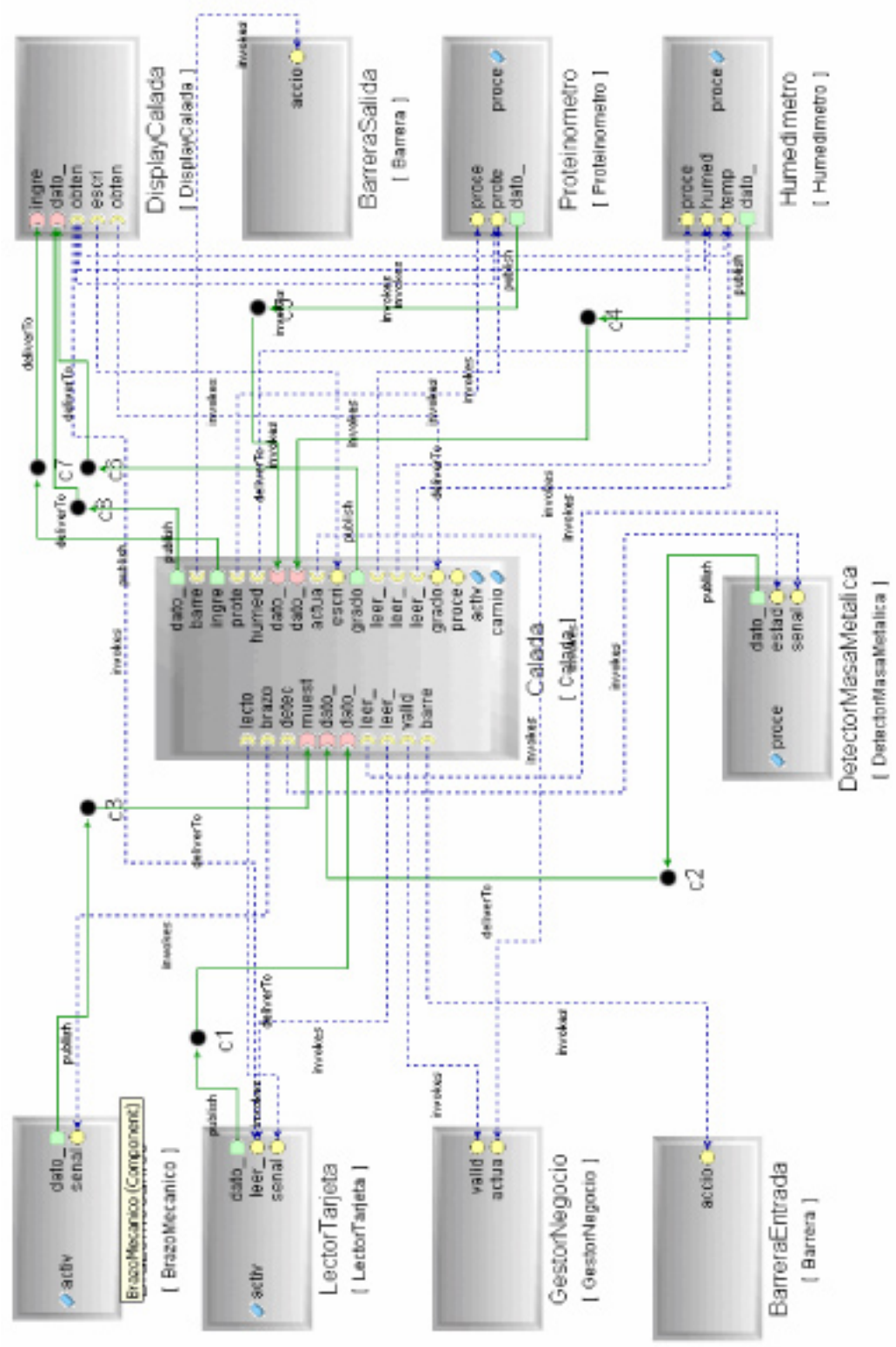


Fig.33 – Ensamblado de componentes del modelo.

En la figura 34 se puede apreciar como se asocia a un componente monolítico: BrazoMecanico (identificado por el icono) las bibliotecas utilizadas y archivos de configuración de la interfaz BrazoMecanico.

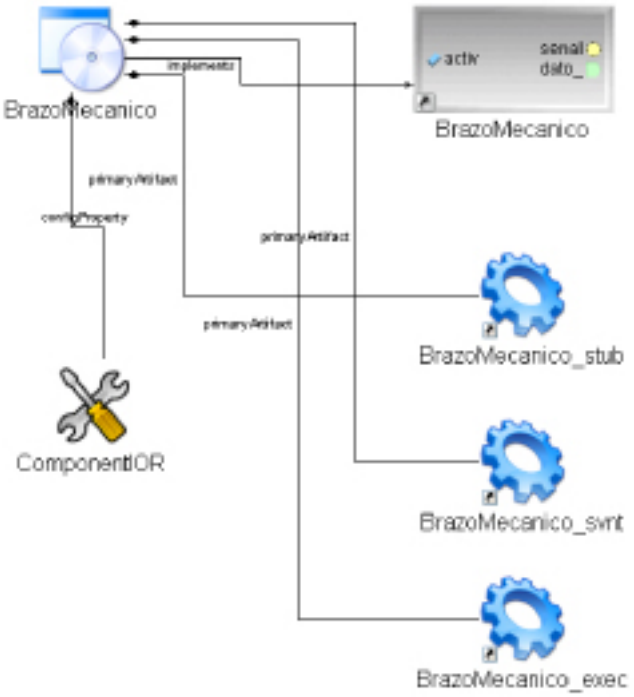


Fig.34 – Componentes de la implementación de la interfaz BrazoMecanico.

Una vez terminado el ensamblado se ejecuta el interprete CIDL el cual genera los archivos que especifican los componentes CCM que implementan las interfaces del modelo.

Los archivos generados a partir del intérprete CIDL son los siguientes:

```
#ifndef CALADA_CIDL
#define CALADA_CIDL

#include "Calada.idl"
composition session Calada_Impl
{
  home executor CaladaHome_Exec
  {
    implements ElevacionDeGranos::CaladaHome;
    manages Calada_Exec;
  };
};


#endif // CALADA_CIDL

#ifndef PROTEINOMETRO_CIDL
#define PROTEINOMETRO_CIDL

#include "Proteinometro.idl"
composition session Proteinometro_Impl
{
  home executor ProteinometroHome_Exec
  {
    implements ElevacionDeGranos::ProteinometroHome;
    manages Proteinometro_Exec;
  };
};


#endif // PROTEINOMETRO_CIDL

#ifndef LECTORTARJETA_CIDL
#define LECTORTARJETA_CIDL

#include "LectorTarjeta.idl"
composition session LectorTarjeta_Impl
```

```
{
  home executor LectorTarjetaHome_Exec
  {
    implements ElevacionDeGranos::LectorTarjetaHome;
    manages LectorTarjeta_Exec;
  };
};


#endif // LECTORTARJETA_CIDL

#ifndef HUMEDIMETRO_CIDL
#define HUMEDIMETRO_CIDL

#include "Humedimetro.idl"
composition session Humedimetro_Impl
{
  home executor HumedimetroHome_Exec
  {
    implements ElevacionDeGranos::HumedimetroHome;
    manages Humedimetro_Exec;
  };
};


#endif // HUMEDIMETRO_CIDL

#ifndef GESTORNEGOCIO_CIDL
#define GESTORNEGOCIO_CIDL

#include "GestorNegocio.idl"
composition session GestorNegocio_Impl
{
  home executor GestorNegocioHome_Exec
  {
    implements ElevacionDeGranos::GestorNegocioHome;
    manages GestorNegocio_Exec;
  };
};
```

```
#endif // GESTORNEGOCIO_CIDL

#ifndef DISPLAYCALADA_CIDL
#define DISPLAYCALADA_CIDL

#include "DisplayCalada.idl"
composition session DisplayCalada_Impl
{
  home executor DisplayCaladaHome_Exec
  {
    implements ElevacionDeGranos::DisplayCaladaHome;
    manages DisplayCalada_Exec;
  };
};


#endif // DISPLAYCALADA_CIDL

#ifndef DETECTORMASAMETALICA_CIDL
#define DETECTORMASAMETALICA_CIDL

#include "DetectorMasaMetalica.idl"
composition session DetectorMasaMetalica_Impl
{
  home executor DetectorMasaMetalicaHome_Exec
  {
    implements ElevacionDeGranos::DetectorMasaMetalicaHome;
    manages DetectorMasaMetalica_Exec;
  };
};


#endif // DETECTORMASAMETALICA_CIDL

#ifndef BRAZOMECANICO_CIDL
#define BRAZOMECANICO_CIDL

#include "BrazoMecanico.idl"
composition session BrazoMecanico_Impl
```

```
{
  home executor BrazoMecanicoHome_Exec
  {
    implements ElevacionDeGranos::BrazoMecanicoHome;
    manages BrazoMecanico_Exec;
  };
};


#endif // BRAZOMECANICO_CIDL

#ifndef BARRERA_CIDL
#define BARRERA_CIDL

#include "Barrera.idl"
composition session Barrera_Impl
{
  home executor BarreraHome_Exec
  {
    implements ElevacionDeGranos::BarreraHome;
    manages Barrera_Exec;
  };
};


#endif // BARRERA_CIDL
```

Una vez terminado con el ensamblado se especifica el empaquetamiento de los componentes en un modelo "ComponentPackages", la idea es definir los paquetes de software que implementan los componentes monolíticos o compuestos que realizan las interfaces del modelo.
La figura 35 muestra como se modela el paquete Barrera (identificado por el icono ) que implementa el componente monolítico Barrera y realiza la interfaz Barrera.

Fig.35 – Empaquetamiento del componente Barrera.

El próximo paso es ejecutar el intérprete Descriptor de Empaquetamiento, el cual genera los siguientes archivos:

- Descriptor de Componentes CORBA (.ccd): contiene información acerca de las interfaces y ports de los componentes.
- Descriptor de Implementación de Componentes (.cid): contiene información de la implementación de los componentes. Que pueden ser monolíticos o compuestos, en el caso de los compuestos también especifica las dependencias y conexiones entre los componentes.
- Descriptor de Artefactos de Implementación (.iad): contiene información de los artefactos de implementación incluyendo las dependencias entre los artefactos.
- Descriptor de Paquetes de Componentes (.cpd): contiene información acerca del agrupamiento de las interfaces del mismo componente dentro de los paquetes de los componentes.
- Descriptor de Configuración de Paquetes (.pcd): contiene información acerca de la configuración específica de los paquetes de los componentes.

Archivo generado a partir del intérprete Descriptor de Empaquetamiento del componente Calada:

```
<?xml version=”1.0” encoding=”UTF-8” standalone=”no” ?>
<Deployment:ComponentImplementationDescription xmlns:
Deployment=”http://www.omg.org/Deployment” xmlns:xmi=”http://
www.omg.org/XMI” xmlns:xsi=”http://www.w3.org/2001/XMLSchema-
instance” xsi:schemaLocation=”http://www.omg.org/Deployment De-
ployment.xsd”>

    <UUID>_ElevacionDeGranos_ComponentImplementations_Calada_Ca-
lada</UUID>

  <implements href=”Calada.ccd”/>

  <monolithicImpl>
    <primaryArtifact>
      <name>Calada_stub</name>
      <referencedArtifact href=”Calada_stub.iad”/>
    </primaryArtifact>
    <primaryArtifact>
      <name>Calada_svnt</name>
      <referencedArtifact href=”Calada_svnt.iad”/>
    </primaryArtifact>
    <primaryArtifact>
      <name>Calada_exec</name>
      <referencedArtifact href=”Calada_exec.iad”/>
    </primaryArtifact>
  </monolithicImpl>

  <configProperty>
    <name>ComponentIOR</name>
```

```
    <value>
      <type>
        <kind>tk_string</kind>
      </type>
      <value>
        <string>Calada.ior</string>
      </value>
    </value>
  </configProperty>

</Deployment:ComponentImplementationDescription>
```

6.5. PASO 3: Configuración

La configuración radica en definir los parámetros propios de la plataforma y del Middleware elegidos. Para esta tarea se utiliza el metamodelo OCML de GME que nos permite definir los parámetros propios del ORB de CIAO. La figura 36 muestra la configuración del ORB de un componente.

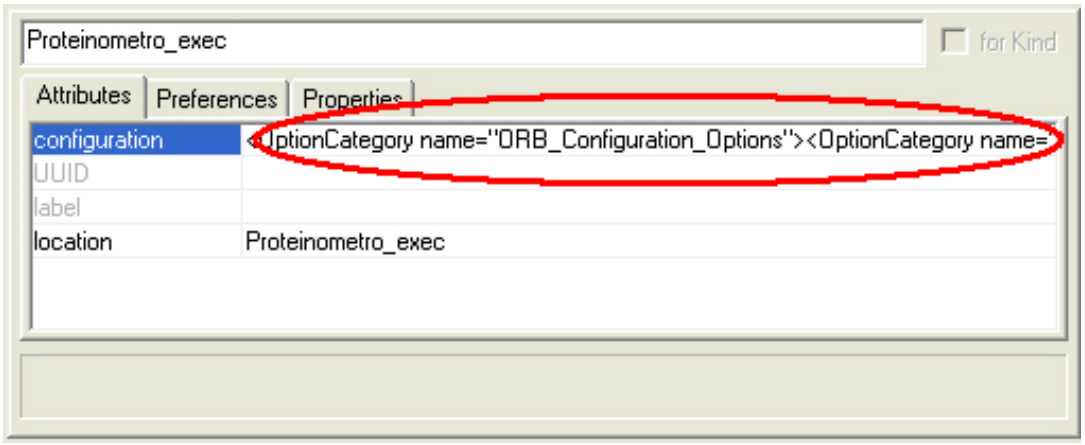


Fig.36 – Configuración de los parámetros del ORB.

6.6. PASO 4: Planificación del Despliegue

La planificación del despliegue incluye el modelado de los artefactos del entorno de la plataforma de ejecución (nodos, conexiones, etc.) y la decisión de cómo desplegar los componentes de la solución en paquetes a través de los diferentes sitios de la plataforma de destino.

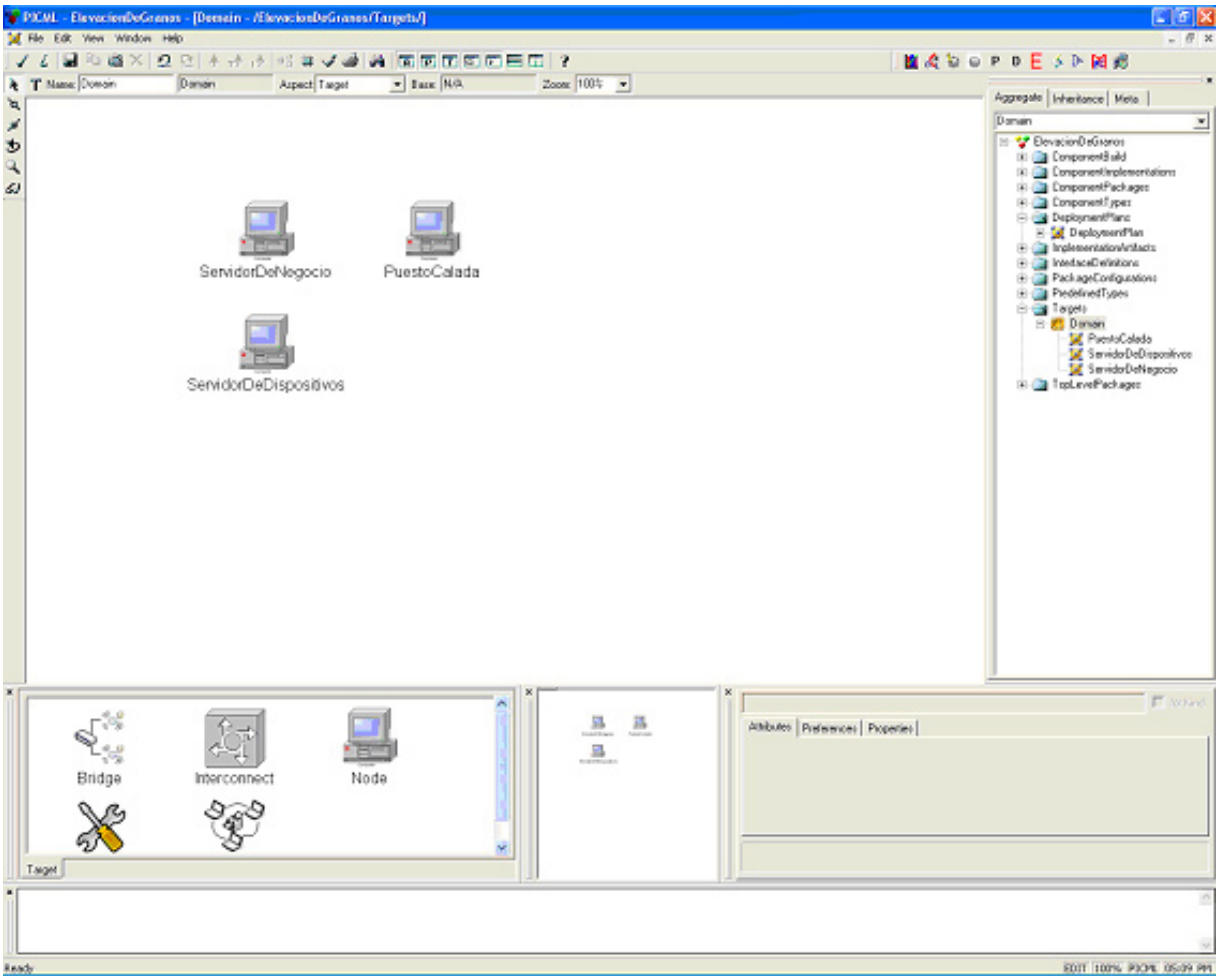


Fig.37 – Nodos de destino del despliegue de los paquetes.

En la figura 37 se observan los tres nodos creados para nuestra solución: ServidorDeNegocio, Servidor-DeDispositivos y PuestoCalada.
Se debe crear un modelo “Target” y un “Domain” para especificar los nodos destinos de nuestra solución. Ahora podemos especificar el despliegue de los componentes del modelo para cada uno de los nodos de destino arriba definidos. Para eso creamos un modelo “DeploymentPlan” y entonces se selecciona un nodo y se eligen los componentes que se desplegaran en él. En la figura 38 se puede apreciar el despliegue de componentes para el nodo ServidorDeDispositivos.

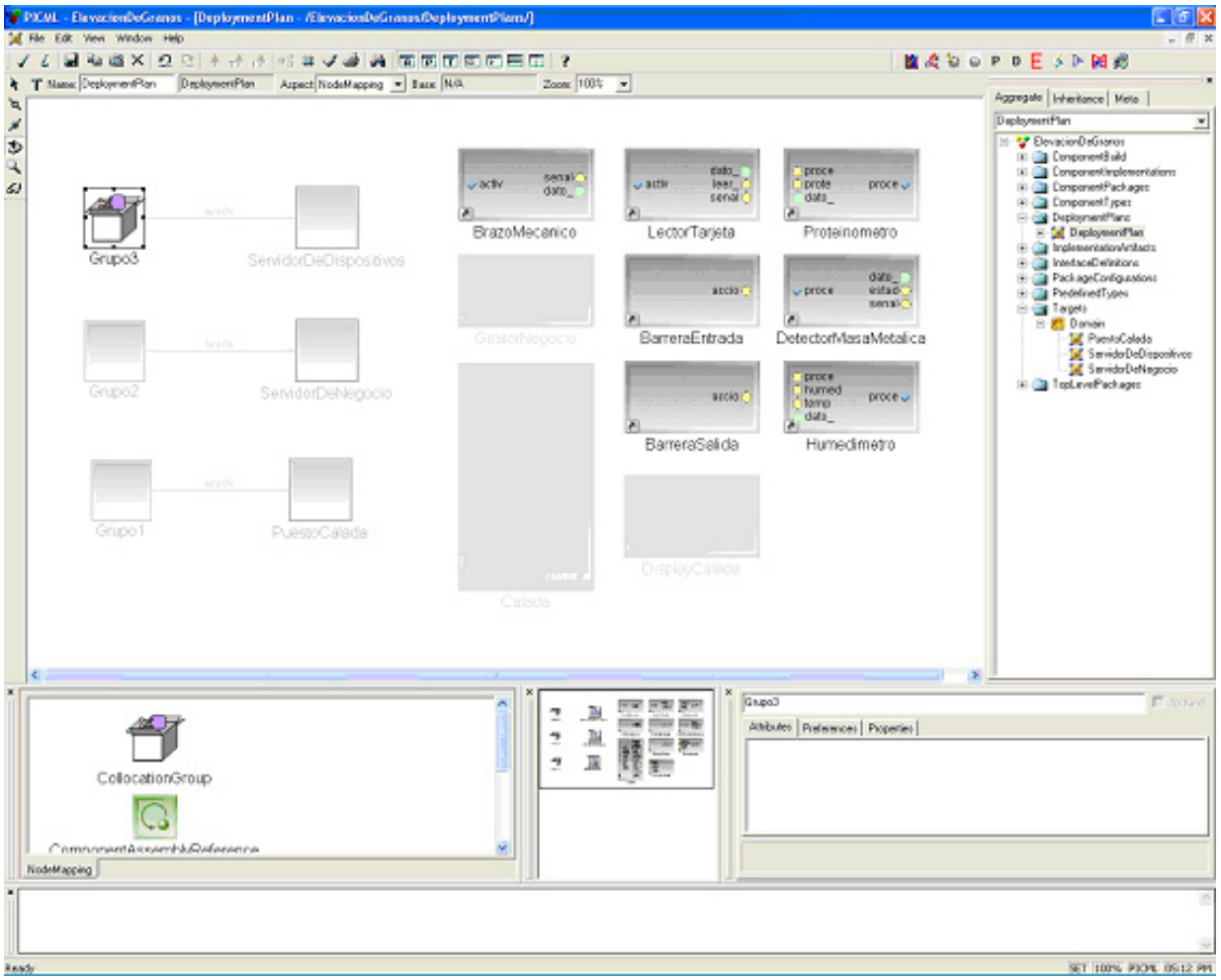


Fig.38 – Componentes a desplegare en el nodo ServidorDeDispositivos.

Luego se ejecuta el intérprete Generador Descriptor de Dominios, el Generador Descriptor del Plan de Despliegue y el Generador del Plan de Despliegue Plano generándose los siguientes archivos:

- Descriptor de Componentes del Dominio (.cdd): contiene información acerca del entorno de destino en donde los componentes de la aplicación serán desplegados.
- Descriptor del Paquete Principal (.tpd): contiene información del elemento de mayor jerarquía del paquete.
- Plan de Despliegue de Componentes (.cpd): contiene información acerca de los componentes a desplegar y las conexiones entre ellos.

6.7. PASO 5: Preparación del Despliegue y Puesta en Marcha

La preparación de los componentes para el despliegue y la puesta en marcha se realiza a través de DAnCE. CoSMIC realiza una síntesis utilizando el interprete “Generador del Plan de Despliegue Aplanado” y genera un archivo que luego usará la implementación de DAnCE de la plataforma destino para realizar el despliegue y la puesta en marcha de los componentes. En el CD que acompaña al trabajo se incluye el archivo generado por este intérprete.

6.8. PASO 6: Análisis y Benchmarking

Las pruebas de rendimiento se realizan a través del BGML, se crea un modelo “ComponentAnalysis” el cual permite ejecutar una serie de pruebas sobre la aplicación desplegada. Una vez terminado el modelo de pruebas se ejecuta el intérprete del BGML que genera el código para la plataforma del CCM seleccionada.

El archivo generado a partir del intérprete BGML es el siguiente:

```
// -*- C++ -*-
// ****  Code generated by the Benchmark Generation
Modeling Language (BGML) Interpreter  ****
// Benchmark Generation Modeling Language (BGML)
// has been developed by
//       Institute for Software Integrated Systems
//       Vanderbilt University
//       Nashville, TN
//       USA
//       http://www.isis.vanderbilt.edu/
//
// Information about BGML is available from:
//     http://www.dre.vanderbilt.edu/cosmic

#ifndef BENCHMARK_DISPLAYCALADA_OBTENER_DATO_H
#define BENCHMARK_DISPLAYCALADA_OBTENER_DATO_H

#include "BGML_Task_Base.h"
#include "DisplayCalada_exec.h"

template <typename T>
class Benchmark_DisplayCalada_obtener_dato : public
BGML_Task_Base
{
public:
   Benchmark_DisplayCalada_obtener_dato (T* remote_
ref);
  ~Benchmark_DisplayCalada_obtener_dato ();

  // Static entry point for benchmarking
  int static run (T* remote_ref);

  int svc (void);
```

6.9. Comparativa: MIC vs. Desarrollo Tradicional Orientado a Objetos

La comparación se debe plantear desde dos perspectivas diferentes: por un lado la utilización del CCM y por otro la utilización de CoSMIC como herramienta para modelar la aplicación CCM.

Con respecto a la utilización de un Middleware orientado a Componentes como CCM las ventajas que podemos apreciar están enumeradas en el capitulo 2.1.5. Por ejemplo en el modelo del ensamblado de componentes de la aplicación ElevacionDeGranos se especifican todas las conexiones entre los componentes del sistema, y la síntesis del modelo genera un archivo de meta-datos que luego será la entrada de una infraestructura de ejecución (DAnCE) que automaticamente interconectará a los componentes

una vez realizada la puesta en marcha de la aplicación. La interconexión entre los componentes del actual sistema se realizó a través de la implementación de un nivel de abstracción menor: programación de sockets.
En CCM se puede especificar a través de un modelo el plan de despliegue de los componentes y su puesta en marcha, en el desarrollo O.O. se puede documentar en un diagrama UML los nodos y como se desplegarán los componentes pero no hay manera de automatizar el despliegue y la puesta en marcha de la aplicación, falta una infraestructura de ejecución.
En la aplicación actual el despliegue se realiza en forma manual al igual que la puesta en marcha de todos los componentes del sistema, dichos procesos son muy tediosos y propensos a errores, según una estadística del actual sistema el 30% de los errores que surgen al liberar una nueva versión de la aplicación son errores referentes al despliegue y a la puesta en marcha.
La diferencia en la utilización del MIC respecto al proceso tradicional de desarrollo se centra principalmente en el nivel de abstracción con el cual se modela el sistema.
Esta diferencia en el nivel de abstracción también produce un cambio en los actores que intervienen en el proceso de desarrollo de la aplicación. En el caso nuestro, el nivel de abstracción conseguido permite a un usuario del dominio específico de las plantas de elevación de granos crear un modelo que soporte los requerimientos por él mismo solicitados –o por otros usuarios-. Aquí apreciamos como el actor que crea el modelo no es el arquitecto de SW o el diseñador de SW sino el usuario especialista en el dominio en cuestión. Cabe aclarar que el DSML permite el modelo de aplicaciones CCM, por lo que el experto del dominio –en este caso- además de poseer un conocimiento del dominio deberá tener un conocimiento del CCM.
Al cambiar al actor que modela el sistema por el usuario experto del dominio en teoría debería aumentar la satisfacción de los usuarios respecto a las expectativas del sistema, pues serían menos propenso a errores de interpretación los requerimientos funcionales y no funcionales de la aplicación. Hacemos mención que los requerimientos de los usuarios en el desarrollo tradicional son transformados por los arquitectos y diseñadores en diagramas de clases, etc. y los diagramas mencionados son a su vez transformados en código por los programadores.
Este nivel de abstracción también nos permite independizar el modelo de la solución de su implementación, esto aumenta la vida del modelo –en teoría, el modelo siempre sería válido- y permite que evolucione en forma iterativa, pues lo independiza de la tecnología subyacente. Si lo comparamos con el método tradicional, tenemos que el modelo no esta independizado de la implementación, esto acorta la vida del modelo pues al cambiar de tecnología deberemos cambiar todo o alguna parte del modelo. Por ejemplo CoSMIC nos permite modelar cualquier aplicación CCM, pero actualmente solo tiene implementados los traductores para la plataforma CIAO. En definitiva la gran ventaja de este enfoque es que nos permite separar la definición del modelo de la definición de cómo se implementará el modelo.
Con respecto al modelo de datos no existe ninguna diferencia entre el MIC y el proceso tradicional, eso se debe a que el modelo de datos por lo general ya se encuentra independizado de la tecnología. En nuestro caso el modelo de datos se mantuvo prácticamente igual y no debió ser modificado en forma completa.
La actividad de análisis y diseño del método tradicional en el MIC no existe tal cual la conocemos, pues como ya dijimos anteriormente el nivel de abstracción es mayor.
En el caso de estudio, el análisis y diseño no se desarrolla a nivel de clases como en el método tradicional sino a nivel de conceptos propios del dominio específico, que luego serán transformados por los intérpretes a modelos propios de la plataforma en particular, en este caso archivos XML que describen la aplicación CCM (en el desarrollo tradicional el código es un modelo).
Un aspecto en donde MIC saca una clara ventaja al desarrollo tradicional es en el soporte a la evolución del modelo, con MIC una modificación en el proceso de negocio, solo implicaría un cambio en el modelo, luego habría que ejecutar nuevamente los intérpretes. A lo sumo habría que modificar el metamodelo y los intérpretes si el cambio es radical. Sin embargo en la mayoría de los casos bastara con la modificación del modelo. En el caso del desarrollo tradicional un cambio en el proceso de negocio puede requerir la ejecución de un proyecto solo para implementar los cambios, pues se deben modificar los requerimientos, el modelo de análisis, el modelo de diseño, el modelo de implementación y el modelo de despliegue entre otros, en cambio en el MIC como ya dijimos implica solo cambiar el modelo y ejecutar nuevamente los intérpretes.
Lo expuesto en el párrafo anteriormente significa que una aplicación desarrollada utilizando el enfoque del MIC tendrá un alto grado de mantenibilidad[102], esto es, será más fácil de mantener y su ciclo de vida será más extenso.

102. Mantenibilidad: Propiedad de un sistema que representa la cantidad de esfuerzo requerida para conservar su funcionamiento normal o para restituirlo una vez se ha presentado un evento de falla

7. Conclusión

La utilización de GME a través de los metamodelos implementados por CoSMIC y el uso de los interpretes dan como resultado para el caso de estudio del presente trabajo -después de la ejecución de 7 intérpretes- la generación de 120 archivos específicos de la plataforma destino –en este caso CIAO-.
Aproximadamente automatizan la escritura de 1200 líneas de código –hay que aclarar que lo generado no es solamente código fuente, sino que también genera archivos que describen los diferentes modelos de la plataforma-, pero lo más importante es que nos permite modelar una aplicación utilizando un modelo del dominio específico de las aplicaciones distribuidas, es decir nos proporciona un nivel de abstracción en donde es posible separar la implementación, de la funcionalidad requerida por la aplicación.
El futuro de la ingeniería del software parece apuntar hacia este camino, ya sea perfeccionando el enfoque del MIC u otros similares, el fin es llevar la construcción de software primero a la estandarización y luego al simple armado de bloques lógicos de componentes o servicios de software pre-armados, como si fuera una línea de producción industrial, en donde el especialista del dominio será el responsable de definir y armar el modelo –a través de un metamodelo específico del dominio- y el rol programador o actor técnico sea solo el de integrador y el responsable del despliegue y de la configuración de la solución.
Para esto es necesario que las herramientas de modelado soporten la definición del modelo desde diferentes perspectivas o aspectos, ya sea el modelo estructural, el modelo de implementación, como también el modelo de despliegue o de comportamiento. Esto es el principal problema que deberán sortear y el segundo es la automatización a través de otra herramienta de los intérpretes que sintetizan el modelo en otro modelo de menor abstracción.
La aplicación del MIC en el caso de estudio nos muestra en forma concreta como a partir de un modelo de alto nivel podemos llegar a una solución, en nuestro ejemplo se utiliza un traductor para la plataforma CIAO pero eso no quita que se pueda construir un traductor para por ejemplo J2EE; de esta manera se logra la independencia de la solución respecto de su implementación. Esto es una clara ventaja respecto al desarrollo convencional de software. Pues el modelo pasa a jugar otro papel en el ciclo de desarrollo de software, el modelo en sí es la parte principal de todo el proceso, el modelo tiene un valor por si mismo.
El enfoque del MIC resuelve gran parte de los problemas relativos a acortar la brecha existente entre el dominio del problema y el dominio de la solución: brindando al especialista del dominio específico la posibilidad de definir un modelo utilizando el vocabulario propio del dominio del problema y un alto nivel de abstracción.
Con respecto a las desventajas encontradas en la utilización del MIC podemos señalar la complejidad y el costo del desarrollo de los traductores, se deben desarrollar uno más traductores de modelo para cada domino específico, si no se consigue una estandarización de dominios, habrá que evaluar el costo / beneficio de la creación de los traductores. Por otro lado el fin de MIC es crear un meta-lenguaje por cada dominio específico, por lo que habrá multitud de lenguajes disponibles, esto último va en contra de la filosofía del UML por ejemplo, en donde se buscó un lenguaje lo suficientemente flexible para poder describir todo tipo de dominios y que fuera comprendido por todos los involucrados en el desarrollo de una aplicación de software.

7.1. Futuras Líneas de Investigación

Se presentan dos posibles extensiones al trabajo de esta tesina. Una referida a la creación de metamodelos de dominios específicos empresariales y de intérpretes para probar la efectividad y eficacia de este enfoque sobre dominios empresariales.
Y otra referida al estudio de la creación de estándares para la especificación de los diferentes dominios con el objetivo de no tener que reinventar la rueda en la definición de los metamodelos cuando los dominios son semejantes.

8. Bibliografía

8.1. Libros, Papers y Referencias

[KAGLEDE2003] Karsai G., Agrawal A., Ledeczi A, “A Metamodel-Driven MDA Process and its Tools”, WISME, UML 2003 Conference, San Francisco, CA, October, 2003,
[KANN2004] Charles W. Kann, “Creating Components: Object Oriented, Concurrent, and Distributed Computing in Java”, ISBN: 0849314992, Auerbach Publications, 2004. Edición digital.

[TANEN2002] Andrew S. Tanenbaum, Maarten van Steen, “Distributed Systems: Principles and Paradigms (Hardcover)”, ISBN: 0130888931, Prentice Hall; 1st edition (January 15, 2002).
[TABU2001] Zahir Tari, Omran Bukhres, “Fundamentals of Distributed Object Systems: The CORBA Perspective“, ISBN: 0471200646, 2001 John Wiley & Sons, Inc.
[BAHA2003] Jean Bacon, Tim Harris, “Operating Systems: Concurrent and Distributed Software Design“, ISBN: 0321117891, Addison Wesley, Marzo 2003. Edición digital.
[SCHMIDT, HUSTON, 2003] Douglas C. Schmidt, Stephen D. Huston, “C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks “, ISBN: 0201795256, Addison Wesley, Octubre 2002. Edición digital.
[SCHSCH2002] R. Schantz and D. Schmidt, “Middleware for Distributed Systems”, Wireless Sensor Networks: A Survey, Computer Networks, Vol. 38, pp 393-422, 2002.
[SPRINKLE2002] Jonathan Sprinkle, “Model-Integrated Computing”, Vanderbilt University 2002.
[EVANS2003] Eric Evans, “Domain-Driven Design: Tackling Complexity in the Heart of Software”, ISBN: 0321125215, Addison Wesley; 1st edition (August 20, 2003).
[MIC2001] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, Gábor Karsai, “Composing Domain-Specific Design Environments”, Volume 34 , Issue 11 (November 2001), table of contents Pages: 44 – 51, Year of Publication: 2001, ISSN:0018-9162, IEEE.
[MIC2003] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Balachandran Natarajan, “Using Model-Integrated Computing to Compose Web Services for Distributed Real-time and Embedded Applications”, Institute for Software Integrated Systems Vanderbilt University, 2003.
[MIC-2-2003] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty, “Model-Integrated Development of Embedded Software”, Proceedings of the IEEE, vol. 91, no. 1, pp. 145--164, Jan. 2003.
[GME2004] Maroti M, Karsai G., Ledeczi A., “Composition and Cloning in Modeling and Meta-Modeling”, Proceedings of IEEE Transactions on Control Systems Technology, 2004, 12, 2.
[GMEMAN2004] Manual de Usuario de GME. http://www.isis.vanderbilt.edu/Projects/gme.
[CCM2004] Nanbor Wang, “COMPOSING SYSTEMIC ASPECTS INTO COMPONENT-ORIENTED DOC MIDDLEWARE”, WASHINGTON UNIVERSITY SEVER INSTITUTE OF TECHNOLOGY DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERIN,G 2004
[CIAO2005] Gan Deng, Jaiganesh Balasubramanian, William Otte, Douglas C. Schmidt, and Aniruddha Gokhale, “DAnCE: A QoS-enabled Component Deployment and Conguration Engine”, Proceedings of the 3rd Working Conference on Component Deployment, Grenoble, France, November 28-29, 2005.
[COSMIC2005] Krishnakumar Balasubramanian, Arvind S. Krishna, Emre Turkay, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt, “Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems”, the International Journal of Embedded Systems, special issue on Design and Verification of Real-Time Embedded Software, April 2005.
[QINGYAO2003] Qing Li and Carolyn Yao, “Real-Time Concepts for Embedded Systems” ISBN:1578201241, CMP Books © 2003 (294 pages)
[COSMIC2005-2] “Model-Driven Development of Distributed Real-time and Embedded Systems”, Douglas C. SCHMIDT, Krishnakumar BALASUBRAMANIAN, Arvind S.KRISHNA, Emre TURKAY, and Aniruddha GOKHALE, Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, USA, June 2006.

8.2. Sitios de Internet

• Institute for Software Integrated Systems - http://www.isis.vanderbilt.edu

9. Software Utilizado

- APF - Análise de Pontos de Função. Versión 2.4.7.0 - Copyrighted by MCsoft 2005 -http://www.ivanmecenas.hpg.ig.com.br/apf.htm.
- GME. Version 5.9.21 - Copyrighted by Vanderbilt University.2005 - http://www.isis.vanderbilt.edu/Projects/gme.
- CIAO. Version 0.4.8 - Copyrighted by Douglas C. Schmidt and his research group at Washington University, University of California, Irvine, and Vanderbilt University, Copyright (c) 1993-2006 - http://deuce.doc.wustl.edu/Download.html.
- CoSMIC – Version 0.4.8 - Copyrighted by Douglas C. Schmidt and his research group at Washington University, University of California, Irvine, and Vanderbilt University, Copyright (c) 1993-2006 - http://www.dre.vanderbilt.edu/cosmic/html/download.html.