SELECCIÓN DE UN MÉTODO FORMAL DE ESPECIFICACIÓN COMO HERRAMIENTA DE DISEÑO DE UN SISTEMA DISTRIBUIDO BASADO EN OBJETOS.

Raúl Gómez González
Magister en Ingeniería Informática
Departamento de Matemáticas
Universidad de La Serena
Tel. Fax +56 51 226662
Casilla 599
La Serena - CHILE
e-mail: rgomez@elqui.cic.userena.cl

Raúl Monge Anwandter
Dr. Ingeniero en Informática
Departamento de Informática
Universidad Técnica Federico Santa María
Tel. +56 32 626364
Casilla 110 - V
Valparaíso - CHILE
email: rmonge@inf.utfsm.cl

ABSTRACT

El problema de desarrollar un sistema de soporte para la interacción de objetos en la programación de sistemas distribuidos basado en objetos es complejo, por lo cual se requiere utilizar una herramienta que permita trabajar con un alto nivel de abstracción y verificar la correctitud del diseño y su realización. Con este fin se plantea como objetivo de este trabajo escoger un método de especificación formal, lo que se realiza en dos etapas:

La primera identifica, utilizando un enfoque basado en el análisis de la bibliografía y de las publicaciones de experiencias realizadas, las características y propiedades tanto de los sistemas a especificar, como de diversos métodos formales, a fin de determinar a un nivel teórico cuáles de estos últimos permiten representaciones correctas de las primeras. Los métodos que resultan seleccionados en esta etapa son dos: Lógica Temporal de Acciones (TLA) y Cálculo de Sistemas de Comunicación con su extensión *Abacus* (CCS/*Abacus).*

La segunda etapa consiste en desarrollar especificaciones de un sistema que posee las características identificadas previamente, en este caso una versión simplificada del sistema de correo electrónico *Grapevine*, mediante los métodos seleccionados en la etapa anterior, a fin de verificar en forma práctica cuál de ellos es el más adecuado para el propósito indicado, resultando seleccionado definitivamente CCS/*Abacus*, y además se obtiene experiencia para desarrollar este tipo de especificaciones; con ello se alcanza el objetivo planteado inicialmente.

Como resultado adicional se obtiene algunas conclusiones acerca de la complejidad que presenta la aplicación de los métodos formales y las situaciones en que es conveniente utilizarlos.

I. PLANTEAMIENTO DEL PROBLEMA Y DE LOS OBJETIVOS

Uno de los aspectos importantes del problema de diseñar un sistema de soporte para la interacción de objetos en la programación de sistemas distribuidos basados en objetos consiste en proporcionar mecanismos de comunicación eficientes y confiables que permitan la interacción entre objetos localizados en forma distante. El soporte de comunicación en un sistema distribuido está relacionado con diversos servicios: administración de objetos, servicio de nombres, administración de conexiones, y administración de grupos. Esto significa que el desarrollo de dicho sistema de soporte requiere diseñar un conjunto de servicios distribuidos, lo cual es una tarea compleja.

En este contexto, el presente trabajo se plantea como objetivo encontrar una herramienta de diseño apropiada, la cual, para satisfacer las características mencionadas, debe permitir trabajar con un alto nivel de abstracción y verificar la correctitud del diseño y su realización. Estos requerimientos son satisfechos por los métodos formales de especificación de sistemas y como existe una diversidad de ellos, es necesario seleccionar aquel que muestre ser el más adecuado para escribir especificaciones que presenten esas propiedades. En este contexto, la selección se realiza en dos etapas:

1. Se identifica y realiza un análisis teórico, basado en la bibliografía y en las publicaciones acerca de experiencias realizadas, tanto de las características asociadas con la especificación de sistemas distribuidos basados en objetos, como de algunos métodos formales representativos, a fin de determinar cuáles de estos últimos permiten representaciones correctas de las primeras, lo cual se usa como criterio para llevar a cabo una primera selección.
2. Los métodos que resultan elegidos son sometidos a una verificación práctica de sus capacidades, en base a la especificación de un sistema de correo electrónico, cuyos requerimientos y especificación no formal se describen en [Lis85]. De esta segunda etapa se espera obtener, además del método seleccionado definitivamente, la experiencia requerida para construir tales especificaciones.

Este trabajo corresponde a un resumen de una tesis completa de Magister en Ingeniería Informática [Gom96], y algunos resultados parciales de ella se han presentado en diversos congresos de Informática.

II. ESPECIFICACIÓN DE SISTEMAS DISTRIBUIDOS BASADOS EN OBJETOS.

Para especificar este tipo de sistemas, debe considerarse las propiedades relevantes que se derivan, por un lado, de la distribución, y, por otro, del enfoque de objetos.

1. DISTRIBUCIÓN. Los aspectos más relevantes que han surgido de sistemas con estas características son [Wei93]:

- **Interfaces y Comportamiento.** La interfaz de un módulo en un sistema distribuido, y por lo tanto concurrente, incluye, además de los puntos de llamado y retorno, los puntos intermedios en los cuales puede interactuar con otras hebras* concurrentes. Los módulos de estos sistemas pueden ser activos, con hebras de actividad de trasfondo, cuyos efectos deben ser modelados en una especificación.
- **Seguridad y Vivacidad.** En la especificación de sistemas distribuidos, se debe distinguir entre las propiedades de seguridad y las de vivacidad. Las propiedades de *seguridad* colocan las restricciones para que el sistema permanezca en un buen estado. Las propiedades de *vivacidad* establecen lo que el sistema debe hacer. Una propiedad de seguridad establece restricciones sobre los resultados

* threads

que se permite que retorne un procedimiento y los efectos laterales que puede tener.

- **Rendimiento / Funcionalidad.** Al diseñar un sistema, se debe buscar un equilibrio entre rendimiento y funcionalidad: mayor funcionalidad tiende a empobrecer el rendimiento y viceversa.
- **Fallas.** Los sistemas distribuidos son concurrentes y el problema principal que agregan es la posibilidad de falla parcial: una parte del sistema puede fallar y recuperarse, mientras el resto del sistema sigue en ejecución.
- **Comunicación.** Además de la posibilidad de fallas independientes, los sistemas distribuidos también tienen demoras de comunicación. A veces estos dos aspectos fuerzan al diseñador a proporcionar una confiabilidad más débil que la que se puede dar a un sistema de un solo sitio.

2. ENFOQUE DE OBJETOS. Un sistema es *basado en objetos* si soporta objetos como sus entidades [Weg87]. Se modela en la forma de un conjunto de objetos que interactúan, de tal forma que en la especificación es necesario tener una **noción precisa de objeto** y un **modelo de interacción de objetos**. Los objetos son unidades que encapsulan su estado y su comportamiento, este último expresado a través de los métodos, y son capaces de operar independientemente y de cooperar concurrentemente. Tanto el estado como los métodos pueden ser accedidos por otros objetos a través de su interfaz. Otros dos aspectos importantes a considerar al modelar este tipo de sistemas son:

- **Ubicación.** Los objetos en el sistema pueden estar distribuidos de diferentes maneras, lo cual depende de requerimientos tales como los servicios que proporcionan o su grado de interacción. Dicha organización puede ser estática o dinámica; en este último caso, se puede agregar nuevos objetos, como también mover o destruir los objetos existentes.
- **Concurrencia.** Los servicios que proporciona un objeto pueden ser solicitados concurrentemente por los otros objetos, o bien puede haber concurrencia al interior de los objetos, al intentar ejecutarse varios métodos a la vez.

3. EL MODELO DE ESPECIFICACIÓN. En la especificación de un sistema, la complejidad se reduce por descomposición y abstracción, dividiendo el sistema en módulos. Lo más importante es especificar el comportamiento de los módulos y las interfaces entre ellos, para indicar cómo interactúan, y no la implementación interna de cada módulo. Las especificaciones de los módulos proveen *abstracción:* esconden detalles de la implementación de un módulo que no son relevantes para el trabajo de los otros módulos, ya que describen solamente *qué* necesita hacer un módulo, y no *cómo* debe hacerlo. En la especificación de sistemas distribuidos basados en objetos, los módulos a especificar son los objetos que lo componen, y dicha especificación debe basarse en un modelo que represente las diversas características que se han descrito para este tipo de sistemas.

Existen una serie de modelos de sistemas distribuidos basados en objetos, cada uno de los cuales coloca énfasis en algunas características particulares. Algunos de estos modelos son: Fäustle [Fäu92] y Cell [Kon91], que abordan el problema de la ubicación de los objetos; BOS [HauJun91], que trata la comunicación entre los objetos; Actores [Agh86] y Ehrich-Sernadas [EhrSer95], que se preocupan de la concurrencia y donde el primero de ellos es muy conocido y utilizado. El modelo de especificación escogido es el proporcionado por el sistema *Argus* [Lis85], debido a la claridad y a la riqueza de conceptos que presenta en cuanto a la representación de objetos y la interacción entre ellos, que resuelven los problemas asociados con la

comunicación y la ubicación de los objetos, la concurrencia tanto interna como externa, y la tolerancia a fallas.

En *Argus*, un sistema distribuido está compuesto de un grupo de *guardianes*, que son objetos que encapsulan y controlan el acceso a uno o más recursos. Un guardián dispone de un conjunto de operaciones, denominadas *métodos*, que son invocadas por los otros guardianes para hacer uso de los recursos. El guardián ejecuta los métodos, sincronizándolos y controlando su acceso. Internamente, un guardián contiene objetos de datos y procesos. Los *procesos* ejecutan las invocaciones a los métodos y realizan las tareas de trasfondo. Algunos de los objetos de datos son globales y constituyen el estado del guardián, y son compartidos por diferentes procesos; otros objetos son locales a los procesos individuales. Los guardianes se comunican invocando a los métodos usando un mecanismo basado en paso de mensajes. Las invocaciones a los métodos son independientes de la localización de los guardianes que los ejecutan. Un guardián puede sobrevivir a las caídas del nodo en que reside, debido a que su estado consiste de objetos *estables*, que sobreviven a las fallas, y de objetos *volátiles*; después de una caída y de la subsecuente recuperación del nodo de un guardián, el sistema de soporte recrea el guardián con los objetos estables y luego comienza un proceso para restaurar los objetos volátiles, después de lo cual el guardián puede reiniciar las tareas de trasfondo y responder a las invocaciones a los métodos.

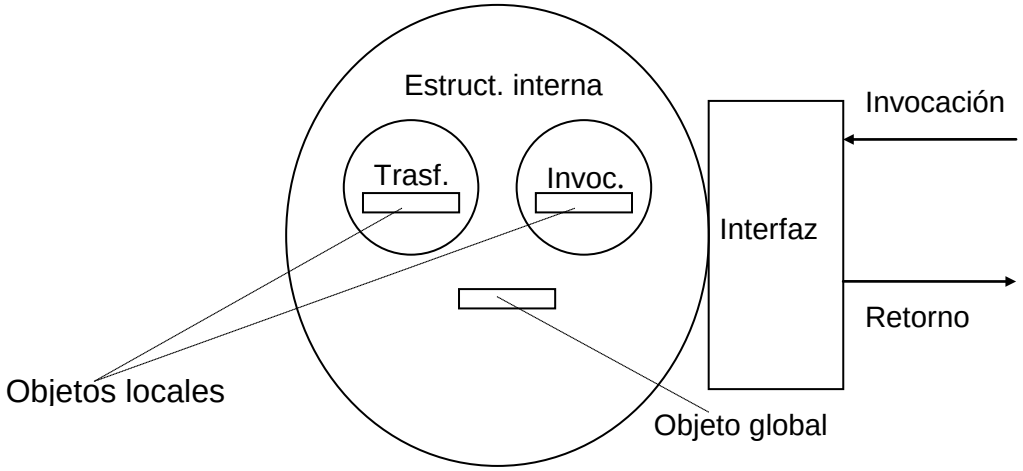


Figura 1: *Modelo de un guardián.*

El modelo permite que cada operación se ejecute como una *acción atómica*, que se distingue por tener las siguientes dos propiedades: la *indivisibilidad*, en que una acción nunca se traslapa (o contiene) con la ejecución de otra acción, y la *recuperabilidad*, en que el efecto global de la actividad es "todo o nada", es decir, o bien todos los objetos permanecen en su estado inicial, o todos cambian a su estado final; una acción que se completa se dice que se *compromete*; en caso contrario, la acción *aborta.* Una forma de implementar la propiedad de indivisibilidad, proporcionando un alto grado de concurrencia, es mediante la *serializabilidad*, en que el efecto de ejecutar un grupo de acciones es el mismo que si se ejecutaran secuencialmente en algún orden; para evitar que una acción observe o interfiera los estados intermedios de otra acción, se debe sincronizar el acceso a los objetos compartidos.

Las acciones se pueden dividir en partes, denominadas *subacciones*, de tal forma que una acción puede contener muchas subacciones, algunas de las cuales se pueden ejecutar secuencialmente y otras concurrentemente. Las subacciones se estructuran en forma jerárquica y, junto con los guardianes, son mecanismos para tratar el problema de las fallas, como también para introducir concurrencia dentro de una

acción. Esta estructura no se puede observar desde el exterior, es decir, la acción completa satisface las propiedades de la atomicidad.

III. CARACTERÍSTICAS DE LOS MÉTODOS DE ESPECIFICACIÓN FORMAL ANALIZADOS.

Se define un *comportamiento* de un módulo como una secuencia $\langle s_0,s_1,s_2,...\rangle$ de estados o como una secuencia $\langle \alpha_0,\alpha_1,\alpha_2,...\rangle$ de acciones. En el presente trabajo se construirá especificaciones que utilizan la *semántica basada en comportamientos*, que es aquella en la que el significado de un módulo es el conjunto de todos los posibles comportamientos que se le permite que exhiba dicho módulo [Lam86].

Los *métodos formales* usados en el desarrollo de software se basan en la Matemática para especificar, desarrollar, y verificar sistemas en forma rigurosa y precisa [Win90], y consisten de un *lenguaje de especificación formal* y de *técnicas de demostración*. Cuando de una especificación se escribe otra especificación, en un nivel de abstracción más bajo, se dice que la segunda especificación *implementa* a la primera. Una implementación es *correcta* respecto de una especificación si es un subconjunto de ella.

En base a la clasificación de los métodos formales que se realiza en [Win90], se incluyen, para ser analizados, seis de ellos. De entre los métodos orientados al modelo de escogieron Redes de Petri, Método de Weihl y Lampson, Cálculo de Sistemas de Comunicación, y Z; de los métodos orientados a las propiedades se escogieron Lógica Temporal de Acciones y Método del Axioma de Transición. A continuación se da una breve descripción de los aspectos principales de estos métodos y un comentario que analiza sus capacidades para representar las características de los sistemas bajo estudio, y que, junto a otras consideraciones importantes, sirven posteriormente como criterios para la comparación que se realiza en esta primera etapa del trabajo.

1. MÉTODO DE WEIHL Y LAMPSON [Wei93]. Una especificación se describe como "trazas" de acciones atómicas, las que se definen en términos de máquinas de estados finitos, escritas en Lógica de Primer Orden. La *interfaz* entre un módulo y sus clientes consiste de una colección de *acciones* con nombre, que son compartidas por los clientes y el módulo.

Una implementación se considera *correcta* si cada comportamiento que produce está en el conjunto definido por la especificación. La correctitud de una implementación se prueba en dos etapas: Primero, se demuestra la existencia de algunos *invariantes*, que son propiedades que son verdaderas para cada estado posible. La segunda etapa consiste en demostrar que la implementación simula la especificación, usando para ello una *función de abstracción*, que explica cómo interpretar cada estado de la implementación en términos de la especificación.

Comentarios: Este método es eficiente y preciso en lo que respecta a su lenguaje de especificación y a sus técnicas de demostración. Permite realizar composiciones y descomposiciones de las especificaciones, las que se pueden escribir en diferentes niveles de abstracción. Proporciona representaciones adecuadas de las diversas características del comportamiento de los sistemas distribuidos, pero la literatura nada dice acerca de cómo definir objetos y sus interacciones. No es mapeable a un lenguaje de programación y no dispone de soporte automatizado. Presenta una mediana madurez en cuanto a publicaciones y aplicaciones de otros autores.

2. MÉTODO DEL AXIOMA DE TRANSICIÓN [Lam89]. Una especificación se divide es las propiedades de seguridad y de vivacidad. La especificación de una propiedad de seguridad se hace utilizando el modelo de Máquina de Estados Finitos, en que las acciones se dividen en las de interfaz y las internas. En un módulo se distinguen el

estado de la interfaz y el estado interno, representados por las *funciones de estado*, que cambian debido a las acciones que se ejecutan; hay un conjunto de reglas, denominadas *axiomas de transición*, que describen cómo cada acción cambia las funciones de estado. La especificación de las propiedades de vivacidad se escriben en Lógica Temporal, mediante axiomas que indican cómo pueden cambiar eventualmente los valores de las funciones de estado.

En cuanto a la correctitud de una implementación, la prueba se hace separando las propiedades de seguridad de las de vivacidad. Para probar las propiedades de seguridad de la especificación, se demuestra que cada acción de la implementación que cambia las funciones de estado de ésta, lo hace tal como lo permite alguna acción de la especificación, utilizando Lógica simple. Para probar que se satisfacen las propiedades de vivacidad de la especificación, se usa tanto las propiedades de seguridad como las de vivacidad de la implementación, mostrando cómo cambian las funciones de estado, usando Lógica Temporal.

Comentarios: Este método dispone de un lenguaje de especificación y de técnicas de demostración adecuadas. Permite componer y descomponer especificaciones; es apropiado principalmente para niveles de abstracción altos. Presenta representaciones adecuadas de los diversos aspectos relacionados con la especificación de sistemas distribuidos, pero no presenta facilidades para estructurar objetos y sus interaciones. No es mapeable a un lenguaje de programación y no dispone de soporte automatizado. No es un método maduro, aunque ha sido referenciado y aplicado por algunos autores.

3. LÓGICA TEMPORAL DE ACCIONES (TLA) [Lam94] [AbaLam94]. Establece las fórmulas temporales como acciones. Un programa se representa por una fórmula Φ, que siempre puede escribirse como la conjunción de un predicado que especifica los valores iniciales de las variables, un predicado que indica las etapas en que las funciones de estado no cambian (etapas "tartamudeantes"), y una fórmula que expresa los requerimientos de imparcialidad, que significa que si cierta operación es posible, entonces el programa debe ejecutarlo eventualmente. $\Phi \wedge \Psi$ es la fórmula que representa la ejecución concurrente de los programas Φ y Ψ.

TLA tiene un conjunto de reglas para probar propiedades temporales de los programas. No hay distinción entre un programa y una propiedad. La fórmula Φ se puede considerar como una especificación de cómo se desea que se comporte un programa. Para demostrar que un programa Ψ implementa a Φ, se demuestra que Ψ satisface la propiedad Φ, utilizando dichas reglas.

TLA+ [Lam95] es un lenguaje formal para definir y asegurar la validez de las fórmulas de TLA. El elemento básico de TLA+ es el *módulo*, que permite escribir la especificación de un sistema en una forma estructurada.

Comentarios: TLA presenta una sólida teoría, basada en Lógica, que involucra un lenguaje de especificación y un sistema de reglas de inferencia, todo bajo una misma notación y significado, con las que se puede escribir las especificaciones y realizar las demostraciones en forma clara y precisa. Permite componer y descomponer especificaciones de una manera natural, utilizando operadores lógicos conocidos. Su teoría permite conceptualizar y representar en forma muy adecuada las diversas características asociadas a la especificación de sistemas distribuidos. En cuanto al modelado de objetos, en la literatura no se presenta explícitamente una solución, pero se visualiza que es posible realizarlo mapeando los módulos de TLA+ en objetos. No es traducible directamente a un lenguaje de programación, y se está construyendo herramientas para automatizarlo. Como sus desarrollos son muy recientes, no ha alcanzado una madurez en cuanto a ser referenciado y aplicado por otros autores. El lenguaje de especificación TLA+ permite escribir las especificaciones de TLA de una

forma estructurada y bien organizada, de tal manera que las especificaciones obtenidas resultan más fáciles de leer.

4. MÉTODO Z [Dar88] [Som92]. Utiliza un lenguaje basado en Matemática Discreta. Una especificación se presenta como una colección de esquemas desplegados gráficamente, que describen las entidades del sistema y las relaciones entre ellas. Usa un cálculo para combinar esquemas y construir esquemas complejos en forma incremental, y para describir las interacciones entre las entidades. El efecto de incluir un esquema A en el esquema B es que este último hereda los elementos definidos por A. Las técnicas de demostración se basan en Matemática Discreta.

Comentarios: Z es un método que presenta un lenguaje y un formato de especificación muy atractivos, y permite realizar demostraciones sobre ellas. Se puede componer y descomponer especificaciones, pero éstas son de bajo nivel de abstracción. Proporciona representaciones adecuadas para algunos aspectos de la especificación de sistemas distribuidos, y es posible utilizar los esquemas para modelar los objetos y sus interacciones. Dispone de herramientas automatizadas, y ha sido aplicado y referenciado por diversos autores.

5. REDES DE PETRI [Mer94] [Mur89] [PapCas92] [Vos94]. Una Red de Petri es un grafo bipartido que presenta un modelo del comportamiento de sistemas concurrentes, representándolo por dos tipos de nodos: las *ubicaciones*, que contienen *marcas* que representan el estado del sistema, y las *transiciones*, que se activan de acuerdo a las *reglas de disparo*. La red tiene una marcación inicial, y cuando una regla de disparo activa una transición, se obtiene una nueva marcación. Repitiendo este procedimiento en forma sucesiva, se obtiene una secuencia de marcaciones y transiciones, que describen el comportamiento del sistema.

Asociado al grafo que representa una especificación, existe un conjunto de fórmulas que permiten efectuar el análisis de las propiedades de dicha especificación, mediante diversas técnicas. A fin de poder expresar algunas propiedades de los sistemas, se ha desarrollado varias extensiones de las Redes de Petri, entre las que se cuentan las Redes Temporizadas, Redes Estocásticas, y Redes de Alto Nivel.

Comentarios: Este método presenta un lenguaje gráfico muy apropiado, con el que se construye especificaciones que se traducen a un conjunto de fórmulas matemáticas que permiten realizar análisis de ellas en base a las técnicas de prueba de que dispone. Para sistemas grandes, las especificaciones crecen en forma explosiva, y su análisis se hace muy complejo. El problema de componer y descomponer especificaciones está resuelto sólo por un tipo de Redes de Alto Nivel, denominadas Redes de Petri Coloreadas [Mur89]. El método proporciona representaciones muy adecuadas de las características de los sistemas distribuidos, y puede modelar objetos también sólo en el caso de las Redes de Petri Coloreadas; para otros tipos de redes, esto se transforma en un problema complejo. Existen diversas herramientas automatizadas, que ayudan a manejar parte de los problemas de explosividad y complejidad. Es un método maduro, existe mucha investigación en el tema, y a pesar de las dificultades que presenta, es ampliamente utilizado, principalmente por sus capacidades visuales.

6. CÁLCULO DE SISTEMAS DE COMUNICACIÓN [Mil86] [Mil89]. CCS es un cálculo que permite expresar y manipular sistemas concurrentes, sustentado en una teoría algebraica y enriquecida con una componente lógica, que está basada en dos ideas centrales:

- La primera es la *observación*: Se describe un sistema en forma tal que se pueda determinar exactamente lo que ve un observador externo. Dos sistemas son equivalentes a la observación si dicho observador no puede diferenciar entre ellos.
- La observación externa no impide el estudio de la estructura de los sistemas. Éstos se construyen con agentes independientes que se comunican, y la *comunicación sincronizada* es la segunda idea central, lo que se realiza como un acto indivisible, denominado *transacción*.

El comportamiento de un sistema se representa por un árbol de derivaciones, y se denota por una expresión, que está sujeta a leyes ecuacionales. Dicha expresión se puede manipular mediante las reglas que rigen el cálculo, ya sea para agregar o eliminar detalles, lográndose con ello representaciones a diferentes niveles de abstracción. Esto permite escribir especificaciones y sus implementaciones, bajo una misma notación.

Las leyes que componen el cálculo permiten ejecutar análisis de las propiedades del sistema especificado, y la técnica proporcionada por la equivalencia de observación permite demostrar que una implementación es correcta con respecto a una especificación.

Abacus [Nie90] [NiePap90] es un lenguaje de especificación ejecutable, que extiede CCS a fin de proporcionar un modelo basado en objetos. Realiza un mapeo entre objetos actores [Agh86] y agentes, y la interacción de objetos utiliza el modelo de paso de valores.

Comentarios: CCS proporciona una teoría que explica todos los aspectos de los sistemas distribuidos, basada en álgebra y en lógica, y proporciona un lenguaje de especificación con gran capacidad expresiva, y de técnicas de prueba completas y rigurosas, todo dentro de un esquema sólido y coherente. Dispone de mecanismos claros para componer y descomponer especificaciones, las cuales pueden estar en cualquier nivel de abstracción. *Abacus* es un lenguaje que extiende CCS para permitir el modelado de objetos y sus interacciones. Las especificaciones en CCS se pueden mapear al lenguaje de programación paralelo ***M***; además, *Abacus* es ejecutable y utilizable como una herramienta de prototipeo para construir y probar especificaciones. CCS es un método maduro, pues ha sido aplicado para especificar muchos sistemas complejos, y ha sido referenciado y analizado por diversos autores.

IV. COMPARACIÓN DE LOS MÉTODOS EN BASE A UN ANÁLISIS TEÓRICO.

La comparación de los seis métodos bajo estudio se realiza utilizando una versión modificada del esquema descrito en [KarCas93] de la siguiente forma:

1. CRITERIO DE COMPARACIÓN. Del estudio realizado a los diversos aspectos asociados con la especificación de sistemas distribuidos basados en objetos y de las propiedades deseables de los métodos formales, se escogió dieciocho características para ser utilizadas como criterio se selección, y que se clasifican en cuatro grupos:

i) Características básicas: Explican la base matemática y las técnicas de especificación y de prueba proporcionadas por el método: Lenguaje de especificación. Procedimiento (etapas a cumplir). Técnicas de demostración de las propiedades de una especificación. Técnicas de demostración de la correctitud de una implementación. Composición y descomposición del sistema. Niveles de abstracción.

ii) Características de representación del comportamiento de sistemas distribuidos: Describen la base teórica y los modelos que utiliza el método para representar las diversas características asociadas al comportamiento de los sistemas distribuidos: Teoría propuesta. Comportamiento. Concurrencia. Tolerancia a fallas. Comunicación. Propiedades de vivacidad. Propiedades de seguridad.

iii) Características de representación de objetos: Describen los modelos que utiliza el método para representar los objetos y sus interacciones: Modelo de objeto. Interacción de objetos.

iv) Características de uso: Describen aspectos acerca de las facilidades que presenta el método en cuanto al mapeo de sus expresiones a constructos de un lenguaje de programación, la existencia de herramientas automatizadas para la especificación y las demostraciones, además de la madurez que tiene en cuanto a aplicaciones importantes en que se ha usado y publicaciones en que es referenciado: Mapeo a un lenguaje de programación. Soporte automatizado. Madurez.

2. VALORES DE ESTIMACIÓN. Se hizo una evaluación de cada método en cuanto a su capacidad para satisfacer cada una de las propiedades descritas, en base a la siguiente estimación:

- *Alto*: El método satisface bien la propiedad.
- *Mediano*: El método satisface la propiedad en forma parcial.
- *Escaso*: El método no satisface la propiedad, o lo hace en forma muy débil.

3. MATRIZ DE COMPARACIÓN. Se construyó una matriz que contrasta los métodos con las propiedades enumeradas, donde se observa que los métodos las satisfacen en forma homogénea dentro de cada grupo al que pertenecen, por lo cual se decidió construir una nueva matriz, en la que se contrasta los métodos con los grupos de propiedades. Esta matriz, que da una visión más global del comportamiento de los métodos, es la siguiente:

	Básicas	**Comportamiento**	**Objetos**	**Uso**
Weihl y Lampson	Alto	Alto	Escaso	Escaso
Axioma de Transición	Alto	Alto	Escaso	Escaso
TLA	Alto	Alto	Mediano	Mediano
Z	Mediano	Mediano	Mediano	Alto
Redes de Petri	Mediano	Mediano	Mediano	Alto
CCS	Alto	Alto	Alto	Alto

Tabla 1: *Matriz de comparación de los seis métodos.*

4. CONCLUSIONES. La clasificación realizada da como resultado que los métodos que ocupan los dos primeros lugares son CCS/*Abacus* y TLA. Esto se justifica por el hecho que ambos métodos desarrollan sólidas teorías sobre sistemas distribuidos, en las que se basan para proporcionar esquemas en que se incluyen las semánticas, los lenguajes de especificación, y las técnicas de demostración, como un todo coherente, a diferencia de otros métodos, tales como el de Weihl y Lampson y el del Axioma de Transición, que no desarrollan teorías propias, y combinan elementos de diverso tipo, tanto en los lenguajes como en las técnicas de demostración. Las ventajas de CCS/*Abacus* sobre TLA radican en su madurez y en su capacidad explícita para representar objetos y sus interacciones.

Las Redes de Petri, que son muy aceptadas como método de especificación, ocupan el tercer lugar en esta clasificación, debido a que hay aspectos que son soportados solamente por ciertas clases de ellas; entre esos aspectos se tiene: composición y descomposición de las especificaciones de un sistema, y demostraciones de las propiedades de las especificaciones y de la correctitud de las implementaciones.

El método Z tiene capacidades limitadas para representar las características del comportamiento de sistemas distribuidos, pero es una herramienta de especificación muy adecuada para sistemas secuenciales.

Es importante destacar que los lenguajes de especificación formal orientados a objetos TROLL [Jun-91], [Har-94] y OASIS [Pas92], que no fueron incluidos en este análisis, debido a que no presentan técnicas de demostración, son herramientas poderosas para especificar sistemas distribuidos, incorporan todos los conceptos aportados por dicho enfoque, y proporcionan diversos modelos de sistemas distribuidos orientados a objetos [HarJun91], [EhrSer95]. TROLL presenta una gran expresividad, está basado en desarrollos teóricos rigurosos [HarSaa93], y dispone de herramientas de soporte automatizado.

V. VERIFICACIÓN PRÁCTICA DE LAS CAPACIDADES DE LOS MÉTODOS.

Una vez escogidos los métodos CCS/*Abacus* y TLA, que corresponde al término de la primera etapa del presente trabajo, se verifica en la práctica sus capacidades y se selecciona el más adecuado, lo que se lleva a cabo comparando las especificaciones obtenidas usando dichos métodos, de una versión simplificada del sistema de correo electrónico *Grapevine* [Bir82], especificada de manera no formal en [Lis85]. Las especificaciones se construyeron de la forma que se indica a continuación.

1. DESCRIPCIÓN Y MODELO DE *GRAPEVINE.* El sistema *Grapevine* tiene una interfaz simple. Cada usuario tiene un nombre único (*user_id*) y un *mailbox*. Las localizaciones de los mailboxes están escondidas para el usuario. Se puede enviar correo a un usuario entregando su user_id y un mensaje, el que se agrega al mailbox del usuario. Se puede leer correo entregando al sistema el user_id del usuario; todos los mensajes del mailbox del usuario se remueven del sistema y retornan al llamador. Por último, hay una operación para agregar nuevos usuarios al sistema, y para extender dinámicamente el sistema de correo electrónico.

El sistema se diseña según el modelo utilizado en *Argus*, y se implementa mediante tres tipos de guardianes: mailers, maildrops, y registries:

- Los *mailers* actúan como el front end del sistema: todo el uso de él ocurre a través de invocaciones a los métodos de los mailers. A fin de obtener alta disponibilidad, se usa muchos mailers, por ejemplo, uno en cada nodo físico.
- Un *maildrop* contiene los mailboxes de algún subconjunto de usuarios. Los maidrops individuales no se replican, sino que se usan múltiples maildrops distribuidos, a fin de reducir la contención e incrementar la disponibilidad, de tal manera que la falla de un nodo físico no deje no disponibles a todas los mailboxes.
- Los *registries* proporcionan los mapeos desde los user_id a los maildrops. Se utiliza registries replicados para incrementar la disponibilidad; se necesita a lo más un registry accesible para enviar o leer correo. Cada registry contiene el mapeo completo para todos los usuarios. Además, cada registry conoce la ubicación de todos los otros registries.

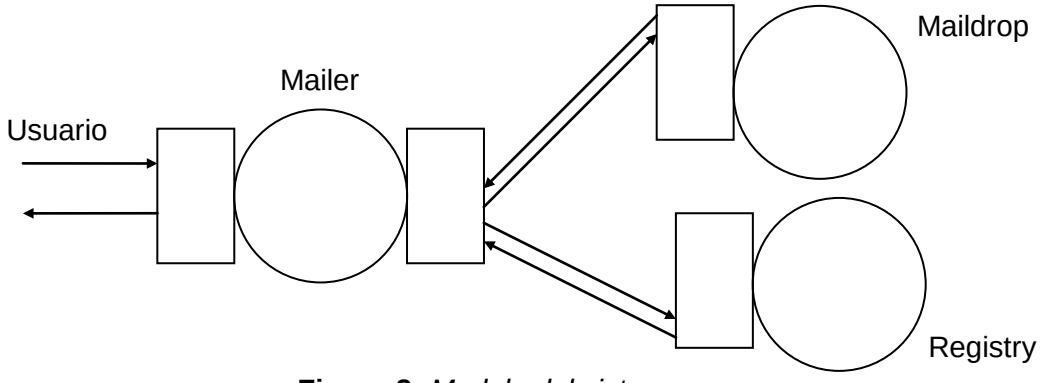


Figura 2: *Modelo del sistema.*

2. EL MÉTODO DE ESPECIFICACIÓN. La especificación del sistema *Grapevine* consiste en describir los comportamientos, mediante acciones atómicas, de los tres tipos de guardianes: mailers, registries, y maildrops, los cuales, como se los ha definido, son objetos, y por consecuencia obedecen a un modelo común de estructura y de interacción, diferenciándose entre ellos en cuanto a los métodos y variables que implementan. Considerando esta comunalidad, y siguiendo el enfoque de Reflexión Computacional planteado en [Mae87], la especificación del sistema se lleva a cabo de la siguiente forma:

- Se desarrolla primero un modelo general de un guardián, el que se denomina Metasistema, y que consiste en describir en forma general su interfaz, su estructura interna, y las restricciones que debe satisfacer para que se comporte de manera correcta. En la especificación de la interfaz, que debe incluir la descripción de la comunicación entre los guardianes, se debe considerar el hecho de que un guardián puede recibir múltiples solicitudes en forma concurrente, y también que el medio de comunicación es no confiable. En cuanto a la estructura interna del guardián, se debe especificar sus métodos y sus variables de estado; cada método puede ser accedido una sola vez al mismo tiempo, pero pueden estarse ejecutando varios métodos a la vez; las variables son accedidas solamente por los métodos, en la misma forma que se accede a estos últimos. Las restricciones que debe satisfacer el guardián corresponden a las propiedades de seguridad y de vivacidad, y también, para que la especificación sea completa, se debe incluir el comportamiento del ambiente.
- Una vez que se ha modelado el Metasistema, se especifica cada tipo de guardián en particular, agregándole las descripciones de sus métodos y variables de estado propios de cada uno de ellos. De esta forma se obtiene la especificación completa del sistema.

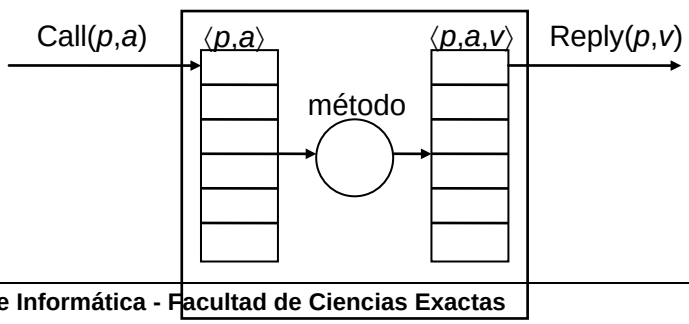
3. LAS ESPECIFICACIONES. Las especificaciones obtenidas son extensas y se muestra como ejemplo la del método *AddRegistryMailer* del guardián mailer; consiste en que éste recibe una solicitud de agregar un nuevo registry al sistema en un nodo *home* indicado, y para realizarla, el mailer debe invocar al método *NewRegistry* de un guardián registry.

a. Especificación en TLA+.

El estado de un guardián se representa mediante dos variables internas globales:

- *incall*: Contiene todas las invocaciones pendientes a los métodos del guardián. Consiste del conjunto de pares $\langle p,a \rangle$, tal que el proceso externo *p* emitió una invocación con argumento *a*, que está esperando a ser procesada.
- *done*: Contiene los resultados de las invocaciones a los métodos del guardián, que ya han sido ejecutadas. Consiste del conjunto de triples $\langle p,a,v \rangle$, tal que se realizó una invocación por el proceso *p* con argumento *a*, y produjo el valor *v*.

Estas variables permiten representar la concurrencia de múltiples invocaciones a los métodos de un guardián y se pueden graficar de la siguiente forma:



incall *done*

Figura 3: *Variables de estado de un guardián.*

La especificación es la siguiente.

$$\begin{aligned}
&\mathit{AddRegistryMailer}(p) =^{\text{def}} \\
&\quad \exists\ \mathit{home} : \wedge\ \langle p,\langle \text{"}\mathit{Re.NewRegistry}\text{"},\mathit{home}\rangle\rangle \in \mathit{incall} \\
&\qquad\qquad \wedge\ \mathit{incall}' = \mathit{incall} \setminus \{\langle p,\langle \text{"}\mathit{Re.NewRegistry}\text{"},\mathit{home}\rangle\rangle\} \\
&\qquad\qquad \wedge \vee\ \mathit{done}' = \mathit{done} \cup \{\langle p,\langle \text{"}\mathit{Re.NewRegistry}\text{"},\mathit{home}\rangle,\text{"}\mathbf{OK}\text{"}\rangle_{\mathit{best}}\rangle \\
&\qquad\qquad\quad \vee\ \mathit{done}' = \mathit{done} \cup \{\langle p,\langle \text{"}\mathit{Re.NewRegistry}\text{"},\mathit{home}\rangle,\text{"}\mathbf{Unavailable}\text{"}\rangle_{\mathit{best}}\}
\end{aligned}$$

La especificación indica que el mailer, al recibir una invocación, la coloca en *incall*, y cuando puede ejecutarla, la retira de esa variable y el resultado lo coloca en *done.* El resultado de la ejecución del método puede ser exitoso si el método *NewRegistry* está disponible, y en caso contrario entrega el mensaje "**Unavailable**". El registry *best* es la referencia actual de acceso al sistema.

b. Especificación en CCS/*Abacus.*

Para ejecutar los métodos, un guardián *Id* se implementa internamente por dos tipos de agentes:

- Un agente *Métodos(Id)*, que controla el acceso a los métodos.
- Un conjunto de agentes $\mathit{Método}_i$, i=1,...,n, que ejecutan los métodos.

De esta forma, *Id* puede recibir y ejecutar invocaciones en forma concurrente.

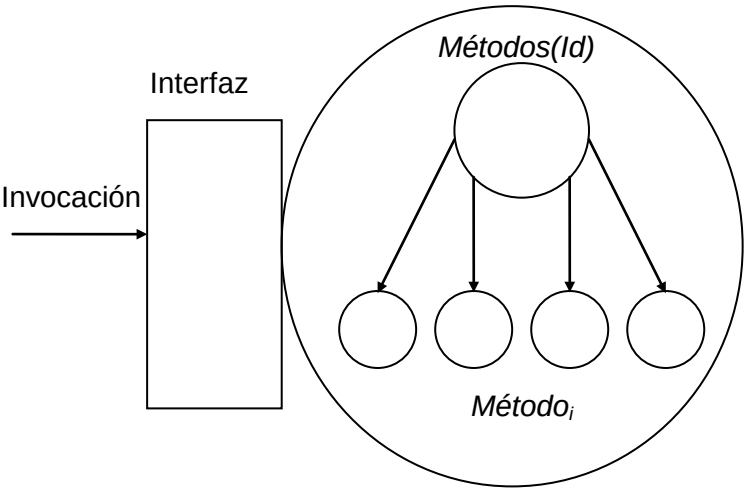


Figura 4: *Implementación de los métodos de un guardián.*

Una especificación se escribe modelando la comunicación entre agentes. Los mensajes se denotan de la siguiente forma:

- Los mensajes *Call* desde un agente *Client* a un agente *Server* son de la forma [*Server*,*Msg*,*Client*].
- Los mensajes *Reply* a un agente *Client* son de la forma [*Client,Reply*].

La especificación del método *AddRegistryMailer* se realiza describiendo el comportamiento del agente que tiene el mismo nombre, de la siguiente forma:

AddRegistryMailer:=[*AddRegistryMailer*,*home*,*Métodos*(*Mailer*)]?
agregaRegistry
agregaRegistry:=
[*guardián*(*Mailer*),*Registry*(*NewRegistry*(*home*,*best*),*AddRegistryMailer*]!
([*AddRegistryMailer*,"**OK**"]?[*Métodos*(*Mailer*),"**OK**"]!
AddRegistryMailer
+[*AddregistryMailer*,"**Unavailable**"]?(*agregaRegistry*
+ [*Métodos*(*Mailer*),*Msg*("**Unavailable**")]!*AddRegistryMailer*)

El guardián mailer ha recibido una solicitud de agregar un nuevo registry al sistema; el mailer realiza esto a través del agente *Métodos(Mailer)*, que controla el acceso a sus métodos, el que solicita al agente *AddRegistryMailer* del mailer, que agregue ese registry en el nodo *home*, después de lo cual este último agente ejecuta la acción *agregaRegistry* En ella, el agente *AddRegistryMailer* solicita a la interfaz *guardián*(*Mailer*) que invoque al método *NewRegistry* del guardián registry, con argumento *home* y a través del registry *best*. El resultado de esta invocación puede ser exitosa, en cuyo caso se lo comunica a *Métodos(Mailer)*, dando término a la solicitud inicial, o bien el método no está disponible, en cuyo caso se intenta ejecutar nuevamente una cierta cantidad de veces, después de lo cual el agente *AddRegistryMailer* entrega como resultado a *Métodos(Mailer)* un mensaje indicando la no disponibilidad.

4. ANÁLISIS DE LAS ESPECIFICACIONES. Una vez desarrolladas las especificaciones, se realiza un análisis de ellas, y se obtiene las siguientes conclusiones relevantes para el propósito de este trabajo:

TLA+ es un lenguaje declarativo, mientras que CCS/*Abacus* es operativo.

- Una diferencia relevante entre estos métodos consiste en que en TLA+ se describe cómo son afectadas las variables de estado por la ocurrencia de las acciones, mientras que en CCS/*Abacus* se define qué comunicaciones externas deben ejecutar los agentes para que se produzcan las acciones. Esto conduce a que una especificación TLA+ se enfoca más hacia la descripción de lo que ocurre internamente en un módulo, mientras que CCS/*Abacus* establece lo que ocurre al exterior de los agentes. Esto se traduce en que en en este último método, para explicar algunos comportamientos, debe crearse agentes, lo que, por un lado, permite establecer aspectos acerca de la arquitectura del sistema, pero, por el otro, puede ser algo artificioso, en el sentido que puede no corresponder a la arquitectura real del sistema.

Como una consecuencia de lo anterior, se tiene que una comunicación en TLA+ se representa mediante el uso de acciones y variables internas a los guardianes, a diferencia de CCS/*Abacus*, que lo hace mediante agentes externos de comunicación.

CCS/*Abacus* permite representar en forma explícita tanto la concurrencia interna como externa de los guardianes, lo que se hace a través del operador & (composición concurrente). Por su parte, la especificación en TLA+ representa en

forma explícita la concurrencia interna a través del operador $\vee$ (disjunción), y para expresar la concurrencia externa mantiene siempre habilitadas las acciones *Call* y *Reply*.

Tanto TLA+ como CCS/*Abacus* permiten representar la atomicidad de las acciones, como también las subacciones. En TLA+ esto se logra mediante combinaciones adecuadas del operador $\wedge$ (conjunción), de tal manera que una acción se compromete si satisface todas las propiedades requeridas para ella y en caso contrario aborta. En CCS/*Abacus* se maneja dichos conceptos a través de las comunicaciones entre los agentes, de tal forma que cuando un agente compromete o aborta una acción, recién envía la respuesta al agente solicitante.

CCS/*Abacus* representa en forma explícita el encapsulamiento de las variables y el acceso a los métodos de un objeto, lo cual no es logrado por TLA+. En este sentido, CCS/*Abacus* permite representar adecuadamente la estructura de los objetos, como también la interacción entre ellos.

La incertidumbre introducida por las fallas y las demoras en la comunicación se modelan en TLA+ con el operador "eventualidad", con el que se logra expresar que una entidad alguna vez en el futuro alcanzará cierto estado deseado. Las fallas y recuperaciones se modelan introduciendo nuevas acciones: *Fail* y *Retry*, respectivamente. En CCS/*Abacus*, la incertidumbre se modela estableciendo las secuencias de acciones que se deben ejecutar ante cada alternativa de falla, demora, o recuperación.

TLA+ tiene la ventaja de poder expresar las propiedades de vivacidad, seguridad y las reglas del ambiente en forma apropiada, mientras que CCS/*Abacus* permite representar bien las propiedad de seguridad y las reglas del ambiente, pero sólo logra expresar en forma parcial la propiedad de vivacidad. La capacidad parcial de CCS/*Abacus* para expresar las propiedades de vivacidad es el punto en que algunos autores encuentran una debilidad en el método, y esto coincide con el resultado obtenido en este trabajo. Sin embargo, en [Lad95] se menciona la existencia de una versión temporal de CCS, denominada Modal Mu-Calculus, que sí permite establecer completamente la propiedad de vivacidad.

En ambas especificaciones se establece que el sistema opera correctamente bajo algunas suposiciones acerca del ambiente. El modelo de interacción entre los guardianes utilizado en los dos casos afirma que si cada guardián satisface su especificación, entonces se comporta correctamente en conjunto con los otros guardianes. El modelo es circular, pues la especificación de un guardián requiere que se comporte correctamente si su ambiente lo hace, y su ambiente consiste de todos los otros guardianes. A pesar de esta circularidad, el modelo permite obtener especificaciones robustas. Los dos métodos bajo análisis pueden expresar de una forma adecuada esta propiedad; TLA+ lo lleva a cabo mediante fórmulas temporales y CCS/*Abacus* la expresa utilizando Lógica de Procesos (***PL***).

5. MATRIZ DE COMPARACIÓN. Con el fin de alcanzar el objetivo de este trabajo, se compara los métodos TLA+ y CCS/*Abacus*, tal como en la primera etapa del presente trabajo, mediante una modificación del método descrito en [KarCas93], construyendo una matriz que utiliza como criterio de decisión once características y propiedades que son claves para la especificación de sistemas distribuidos en base a objetos, clasificadas en tres grupos:

i. Características de representación del comportamiento de sistemas distribuidos:

1. Concurrencia interna y externa.
2. Fallas y recuperaciones.

3. Incertidumbre ante fallas y demoras en la comunicación.
4. Atomicidad de las acciones.
5. Propiedad de seguridad.
6. Propiedad de vivacidad.
7. Comunicación.

ii. Características de representación de objetos.

8. Modelo de objetos.
9. Interacción de objetos.
10. Arquitectura del sistema.

iv. Representación del ambiente.

11. Modelado del ambiente.

Se hace una evaluación del método en cuanto a su capacidad para satisfacer cada una de las propiedades descritas, mediante los mismos valores de estimación antes utilizados, en base a la experiencia obtenida en las especificaciones desarrolladas, obteniéndose la siguiente matriz:

	TLA+	**CCS/*Abacus***
Concurrencia	Alto	Alto
Fallas	Alto	Alto
Incertidumbre	Alto	Alto
Atomicidad	Alto	Alto
Seguridad	Alto	Alto
Vivacidad	Alto	Mediano
Comunicación	Mediano	Alto
Modelo de objetos	Mediano	Alto
Interacción de objetos	Mediano	Alto
Arquitectura	Bajo	Alto
Ambiente	Alto	Alto

Tabla 2: *Matriz detallada de comparación de la especificaciones.*

En la matriz se puede observar que los métodos satisfacen las propiedades en forma homogénea dentro del grupo al que pertenecen, por lo cual se decide construir una nueva matriz, en la que se contrasta los métodos con los grupos de propiedades. Esta matriz da una visión más global acerca del comportamiento de los métodos y es la siguiente:

	TLA+	**CCS/*Abacus***
Sistemas distribuidos	Alto	Alto
Objetos	Mediano	Alto
Ambiente	Alto	Alto

Tabla 3: *Matriz resumida de comparación de la especificaciones.*

6. RESULTADOS. Tal como indica esta última matriz, el método que satisface de forma más adecuada las características y propiedades relevantes para la especificación formal de sistemas distribuidos basados en objetos es CCS con su

extensión *Abacus.* La ventaja que éste tiene sobre TLA+ radica en su capacidad para modelar objetos y sus interacciones. En cuanto a la expresión del comportamiento de sistemas distribuidos, existe un equilibrio, existiendo un aspecto en contra de cada método: TLA+ no expresa de manera explícita la comunicación externa, y CCS/*Abacus* no puede expresar completamente la propiedad de vivacidad, aunque esta última es una dificultad más relevante para el problema de la especificación, y es el único aspecto que se detectó en que CCS/*Abacus* presenta debilidad, aunque, tal como se mencionó anteriormente, esto es superado por la versión temporal de CCS. Así, para el propósito de especificar sistemas distribuidos en base a objetos, es preferible utilizar el método CCS/*Abacus*. En todo caso, TLA+ es un método riguroso y coherente, que desarrolla una cantidad de elementos conceptuales y metodológicos que permiten especificar y analizar formalmente los comportamientos de estos sistemas. Puede afirmarse que TLA+ es un método adecuado para analizar y desarrollar conceptos acerca de sistemas distribuidos en una forma abstracta, sin considerar las implementaciones posteriores. En cambio, CCS/*Abacus* está enfocado hacia la construcción de especificaciones más concretas y pensando en el comportamiento real de dichos sistemas, esto último sin bajar de nivel de abstracción.

VI. CONCLUSIONES.

El problema de modelar sistemas distribuidos basados en objetos es complejo; requiere tratar sus diversas características y hacer representaciones adecuadas de ellas. En base al estudio que se ha realizado, se concluye que los métodos formales proporcionan mecanismos que permiten satisfacer todos estos requerimientos, de una manera consistente y precisa; además, proporcionan técnicas para demostrar propiedades de las especificaciones y la correctitud de las especificaciones.

Esta conclusión es consistente con el hecho de que el presente trabajo se basó en la idea de que los métodos formales son adecuados para especificar sistemas distribuidos basados en objetos, y que se propuso como objetivo determinar cuál de ellos, de entre un grupo representativo, permite llevar a cabo esta tarea de una mejor forma, considerando las diversas características de dicho tipo de sistemas. El objetivo propuesto se alcanzó, pues se logró determinar, después de un detallado análisis, que el método CCS, en conjunto con su extensión *Abacus*, es el más adecuado para construir para la especificación requerida. También se logró el objetivo de obtener experiencia para escribir especificaciones de este tipo de sistemas.

Pero del desarrollo del trabajo, y considerando sobre todo las dos especificaciones realizadas, se obtiene también, como una conclusión relevante, que si bien los métodos formales son herramientas que disponen de una gran riqueza expresiva y de análisis, son altamente complejas, tanto en su utilización como en la comprensión de las especificaciones; para usar un método formal se requiere de un elevado conocimiento de Matemática no trivial. Esta complejidad se ve incrementada cuando se intenta llevar a cabo demostraciones acerca de las especificaciones y sus implementaciones, a tal punto que se hace imprescindible el uso de herramientas automatizadas. El conocimiento especializado que requieren conduce a que sea difícil su aplicabilidad en el desarrollo de software.

Otro aspecto que se puede inferir del trabajo es que, si bien las especificaciones formales son rigurosas y precisas, y permiten demostrar que satisfacen diversas características relevantes para un sistema, pueden no ser completas y también pueden formalizar aspectos que no corresponden a los requerimientos. Para evitar estos problemas debería desarrollarse demostraciones de cada requerimiento, lo cual incrementaría considerablemente el tiempo y la complejidad del diseño.

A pesar de estas dificultades que presentan los métodos formales, se debe destacar que su desarrollo ha conducido a realizar aportes teóricos importantes para la

Informática, al proponer teorías y modelos rigurosos acerca de los sistemas y sus propiedades; además, las especificaciones formales permiten deducir características relevantes de la estructura y el comportamiento de los sistemas, lo cual tiene un valor preponderante en el conocimiento acerca de ellos.

En vista de estas consideraciones, puede pensarse en los métodos formales como elementos importantes en el estudio de los sistemas informáticos, pero no como herramientas de utilización masiva para el desarrollo de software, salvo, tal vez, aquel de seguridad crítica.

Así, es necesario relajar la rijidez de los métodos formales con el fin de facilitar el proceso de diseño. Existen métodos no formales, los que, aplicados cuidadosamente, permiten obtener especificaciones correctas y confiables, por lo que, en base a todas las consideraciones anteriores acerca de los métodos formales, se concluye que son más adecuados que estos últimos para diseñar sistemas. Sin embargo, los métodos formales siguen siendo herramientas útiles para tratar algunos aspectos del diseño que requieren ser investigados con mayor precisión, como, por ejemplo, poder establecer si una especificación satisface ciertas propiedades. De esta forma, aún cuando los métodos formales no son recomendables para todas las aplicaciones, hay situaciones en que sí son necesarios, tal como se acaba de indicar.

REFERENCIAS

[AbaLam94] Abadi Martin, Lamport Leslie; "Conjoining Specifications"; Systems Research Center, SRC Report; 136 Lytton Ave., Palo Alto, CA 94301; July 1994.

[Agh86### Agha Gul; "Actors: A Model of Concurrent Computation in Distributed Systems"; The MIT Press; 1986.

[Dar88] Darrel Ince; "Z and System Specification"; *Information and Software Technology*; April 1988.

[EhrSer95] Ehrich Hans D., Sernadas Amílcar; "Local Specification of Distributed Families of Sequential Objects"; *Recent Trends in Data Type Specification*, Artesiano E., Reggio G., Tarlecki A., eds.; Springer LNCS (submitted); 1995.

[Fäu92] Fäustle Michael; "An Ortogonal Distribution Language for Uniform Object-Oriented Languages"; Technical Report, Friedrich-Alexander University, Erlangen-Nürberg; 1992.

[Gom96] Gómez Raúl; "Especificación Formal de un Sistema Concurrente y Distribuido basado en Objetos"; Tesis de Magister en Ingeniería Informática, Universidad Técnica Federico Santa María, Valparaíso, Chile, 1996.

[Har-94] Hartmann Thorsten et al.; "Revised Version of the Modeling Language TROLL: TROLL Version 2.0"; Technische Universität Braunschweig; April 1994.

[HarJun91] Hartmann Thorsten, Jungclaus Ralf; "Abstract Description of Distributed Object Systems"; Object-Based Concurrent Computing, *Proc. ECOOP´91 Workshop*, Geneva, 1991, Tokoro M., Nierstrasz O., Wegner P. (eds.), Springer-Verlag, Berlin, LNCS 612; 1991.

[Jun-91] Junclaus Ralf et al.; "Object-Oriented Specification of Information Systems: The TROLL Languaje"; Technique Universität Brauchweig;December 1991.

[KarCas93] Karam, Gerald and Casselman, Ronald; "A Cataloging Framework for Software Development Methods"; *IEEE Computer*, February 1993.

[Kon91] Konstantas, Dimitri; "Cell: A Model for Strongly Distributed Object Based Systems"; Working Paper; 1991.

[Lad95] Ladkin Peter; "Formal But Livery Buffers in TLA+"; Technische Fakultät, Universität Bielefeld, D-33501; April 1995.

[Lam86] Lamport Leslie; "A Formal Basis for the Specification of Concurrent Systems"; in *Distributed Operatings Systems: Theory and Practice, NATO ASI Series,* Vol. F28; Parker, Y. et al. (eds.); SpringerVerlag, Berlin, Heidelberg; 1987.

[Lam89] Lamport Leslie; "A Simple Approach to Specifyng Concurrent Systems"; *Communications of the ACM*, vol. 32, Nº 1; January 1989.

[Lam94] Lamport Leslie; "The Temporal Logic of Actions"; *ACM Transactions on Programming Languages and Systems*; 1994.

[Lam95] Lamport Leslie; "TLA+"; DEC Systems Research Center, SRC Report; 136 Lytton Ave., Palo Alto, CA 94301; Mar. 1995.

[Lis85### Liskov Barbara; "The Argus Languaje and System"; *Distributed Systems*, Lecture Notes in Computer Science; Springer-Verlag; 1985.

[Mae87] Maes Pattie; "Concepts and Experiments in Computational Reflextion"; *OOSPLA'87 Proceedings*; October 1987.

[Mer94] Merceron Agathe; "Concurrent Behaviour of Petri Nets"; Escuela de Ingeniería, Universidad de Chile, Chile; 1994.

[Mur89] Murata Tadao; "Petri Nets: Properties, Analysis and Applications"; *IEEE Proceedings*, vol. 77, Nº4; April 1989.

###Mil86### Milner Robin; "A Calculus of Communicating Systems"; *LFCS Report Series*, Department of Computer Science, University of Edinburg; 1986.

###Mil89### Milner, Robin; "*Communication and Concurrency*"; Prentice-Hall; 1989.

[Nie90] Nierstrasz, Oscar; "A Guide to Specyfing Concurrent Behaviour with *Abacus*"; ACM.Published in, *SIGPLAN Notices*, Proccedings OOSPLA/ECOOP'90, Ottawa; Oct. 1990.

[NiePap90] Nierstrasz Oscar, Papathomas Michael; "Viewing Objects as Patterns of Communicating Agents"; in *SIGPLAN Notices* OOSPLA/ECOOP'90, Otawa; October 1990.

[PapCas92] Papelis Yannis, Casavant Thomas; "Specification and Analysis of Parallel/Distributed Software and Systems by Petri Nets With Transition Enabling Functions"; *IEEE Transactions of Software Engineering*, vol. 18, Nº 3; March 1992.

[Pas92] Pastor Oscar; "Diseño y Desarrollo de un Entorno de Producción Automática de Software Basado en el Modelo de Objetos"; Tesis Doctoral, Universidad Politécnica de Valencia; 1992.

[Som92] Sommerville Ian; "*Software Engineering* "; Addison-Wesley; 1992.

[Ste87] Stein Lynn; "Delegation Is Inheritance", *OOSPLA´87 Proceedings*; ACM; October 1987.

[Vos94] Voss Klaus; "Net Modelling of Distributed Database Systems"; Gesellchauft für Mathematik und Datenverabeitung, Sankt Augustin, Germany; 1994.

[Wei93] Weihl William; "Specification of Concurrent and Distributed Systems"; in *Distributed Systems*, ed.Mullender Sape; Addison-Wesley; 1993.

[Win90] Wing Jeannette; "A Specifier`s Introduction to Formal Methods"; *IEEE Computer*, September 1990.