Cálculo para el modelado formal de arquitecturas de software basadas en componentes: $cálculo - \rho_{arq}$

[Formal Modeling for Component-Based Software Architectures: $\rho_{arq} - calculus$]

Fecha de recepción: 28 de Marzo de 2009

Fecha de aceptación: 28 de Abril de 2009

Henry Alberto Diosa [1]

Juan Francisco Díaz Frías [2]

Carlos Mauricio Gaona Cuevas [3]

Resumen

El *cálculo* $- \rho_{arq}$ es una notación formal para especificar aspectos estructurales y dinámicos de arquitecturas de software basadas en componentes. La idea subyacente es partir de una configuración estática inicial que sólo es susceptible de pasar a una nueva configuración por medio de la aplicación de reglas de la semántica operacional del formalismo desarrollado. Para dar más alcance semántico a los diagramas de componentes en UML 2.x se propone una extensión estereotipada de la notación gráfica. En consecuencia, se logra el diseño de configuraciones de componentes traducibles al *cálculo* $- \rho_{arq}$. Adicionalmente, esta notación permite representar aspectos dinámicos de manera consistente con otras propuestas de modelado de sistemas distribuidos basados en componentes de software.

Palabras claves: *arquitecturas de software, cálculo* $- \rho_{arq}$*, semántica operacional, UML 2.x, componentes de software, sistemas distribuidos*

Abstract

The $\rho_{arq} - calculus$ is a formal notation to specify structural and dynamic aspects of component-based software architectures. The underlying idea is to leave of an initial static configuration that is only susceptible of passing to a new configuration by means of the application of the operational semantics rules of the $\rho_{arq} - calculus$. We propose a graphic notation that gives more semantic reach to diagrams of components in UML 2.x.. As a result, this notation allows to represent dynamic aspects in a consistent way with other modeling proposals of component-based software in distributed systems.

Keywords: *Software architectures,* $\rho_{arq} - calculus$*, operational semantics, UML 2.x, software components, distributed systems*

[1] Ingeniero de Sistemas de la Universidad Nacional de Colombia. Magíster en Teleinformática de la Universidad Distrital Francisco José de Caldas. Ph.D. en Ingeniería con Énfasis en Ciencias de la Computación de la Universidad del Valle. Actualmente se desempeña como profesor de planta en la Universidad Distrital en Bogotá D.C.

[2] Matemático e Ingeniero de Sistemas y Computación de la Universidad de Los Andes. Ph.D. en Informática de la Universidad de París XI. Actualmente es profesor de planta en la Universidad del Valle en la Escuela de Ingeniería de Sistemas y Computación (Cali), Colombia.

[3] Ingeniero Industrial de la Universidad Industrial de Santander. Ph.D. Computer Science de la Universidad de Massachussetts Lowell. Actualmente es profesor de planta en la Universidad del Valle en la Escuela de Ingeniería de Sistemas y Computación (Cali), Colombia.

1. INTRODUCCIÓN

El modelado formal de arquitecturas de software es un tema de investigación abierto y se ha trabajado de manera intensa desde la última década del siglo pasado. La mayoría de las propuestas buscan obtener especificaciones arquitecturales de software que permitan el chequeo de propiedades estáticas y dinámicas.

Wright es un lenguaje de descripción de conexión arquitectural [Allen and Garlan, 1997] que aporta una notación y teoría subyacente para una semántica explícita de la conexión arquitectural. La propuesta de UniCon (*Universal Connector*) [Shaw and Others, 1995] hace consideraciones sobre la manera de soportar abstracciones arquitecturales , localizando y codificando las formas en que interactúan componentes y distinguiendo entre los varios empaquetamientos de los mismos, que requieren diferentes formas de interacción. Rapide [Bryan and Mann, 1995] es un lenguaje de descripción arquitectural orientado a objetos, concurrente y basado en eventos desarrollado por la Universidad de Stanford; es relevante la adopción de un modelo de ejecución en el que la concurrencia, sincronización, flujo de datos y temporización en un prototipo son representados explícitamente. Un enfoque puramente sintáctico soportado en teoría de conjuntos fue presentado por Thomas R. Dean y James R. Cordy [Dean and Cordy, 1995]. Especificación y análisis formal de arquitecturas de software como reacciones aplicadas sobre soluciones de moléculas regidas por reglas explícitamente establecidas se propuso con la máquina química abstracta de Berry y Boudol [Berry and Boudol, 1992]. Acme [David Garlan and Wile, 1997] es un lenguaje que soporta transformaciones de especificaciones arquitecturales entre lenguajes de descripción arquitectural heterogéneos. Un lenguaje de ligado declarativo que puede usarse para definir la composición jerárquica de componentes interconectados es Darwin [Jeff Magee and Kramer, 1995]. La distribución de dichos componentes es tratada de manera ortogonal a la estructuración del sistema, soporta la especificación tanto de estructuras estáticas como de estructuras dinámicas que evolucionan durante la ejecución. La propuesta del Instituto para Investigación en Software de la Universidad de California en Irvine es *xADL 2.0* [Taylor et al., 2002], consiste en un marco de trabajo basado en XML para el desarrollo de lenguajes de descripción arquitectural. Otro esfuerzo digno de resaltar es el de Medvidovic y sus colaboradores, ellos proponen utilizar UML [Medvidovic et al., 2002] en la especificación de arquitecturas de software; sus aportes son interesantes respecto a las debilidades halladas en esta herramienta semiformal de modelado para el nivel de abstracción abordado.

Este trabajo propone la sintaxis y la semántica operacional de un formalismo para especificar arquitecturas de software basadas en componentes denominado *cálculo* $-\ \rho_{arq}$ como una extensión del *cálculo* $-\ \rho$ [Niehren and Muller, 1995, Smolka, 1994a, Smolka, 1994b] para el nivel de abstracción arquitectural. Teniendo en cuenta el uso frecuente de UML 2.x [Object Management Group, 2007, págs. 159-160], se hace una propuesta de estereotipado para hacer traducibles los modelos de software basados en componentes a expresiones formales que permiten modelar el flujo de ejecución en una arquitectura.

2. SINTAXIS DEL $CÁLCULO - \rho_{ARQ}$

El formalismo propuesto consta de un conjunto de entidades sintácticas significativas desde la perspectiva del modelado arquitectural de software; tales entidades se compendian en la Tabla 1.

2.1 Símbolos

Se cuenta con un alfabeto infinito de variables y un alfabeto infinito de nombres. Las variables son lugares para cargar nombres; es decir, no hay otros valores diferentes a los nombres. Tanto nombres como variables son indiferenciadamente denominados referencias. Términos como $\overline{x}$ equivalen a una secuencia como $(x_1, x_2, ..., x_n)$.

2.2 Expresiones

En el *cálculo* $-\ \rho_{arq}$ las expresiones representan configuraciones arquitecturales que al encapsularlas pueden representar componentes

- La expresión $\top$(**Null**) se equipara a un **componente nulo** que no ejecuta acción alguna.
- La **composición** expresa ejecución concurrente de E y F. Esta expresión será altamente útil para modelar componentes que se ejecutan concurrentemente en una arquitectura.
- La expresión $E^{(int)}$ permite representar la parte interna o encapsulada de un componente. Esta entidad sintáctica permite separar la parte pública o externa de un componente (que permite alambrarlo con otros componentes para lograr determinadas configuraciones arquitecturales) y la parte interna, que puede estar implementada en el marco de un paradigma diferente al basado en componentes y que solamente interesa desde la perspectiva de ejecución exitosa o no de dicho cuerpo interno que puede activarse por obtener valores de entrada a través de las interfaces públicas de requerimiento de servicios.
- **El combinador de selección condicionada**[4] es una generalización útil del condicional, tiene la forma:

  $$if\ (C_1)\ \cdots\ (C_n)\ else\ G$$

  donde $C_k ::= \exists\overline{x}(\phi_k\ then\ E_k)$ con $k = 1 \ldots n$. Las cláusulas $(C_1)\ \cdots\ (C_n)$ pueden pensarse como computaciones en competencia; si la guarda de una cláusula es satisfecha se libera el cuerpo de la misma. Si ocurre lo contrario, ésta es descartada. Si ninguna cláusula puede ser satisfecha se activa el argumento G del *else*, esto último es un camino para el manejo de fallas.

  Cuando existe una sóla cláusula no cuantificada, se tiene el condicional convencional $if\ \phi\ then\ E\ else\ G\ fi$; por esta razón, no es necesario contarlo como entidad sintáctica aparte.

  Esta entidad sintáctica introduce no-determinismo cuando se cuenta con más de una cláusula que puede ser satisfecha desde el contexto.

[4]Esta representación de ejecución alternativa de componentes en el *cálculo* $-\ \rho_{arq}$ es una derivación de lo que se denominó **combinador de disyunción condicionada o vigilada** propuesta en las primeras versiones extendidas del *cálculo* $-\ \gamma$ [Smolka, 1994a, Smolka, 1994b].

SÍMBOLOS		
x, y, z		variables
a, b, c		nombres
$u, v, w ::= x \mid a$		referencias
EXPRESIONES		
$E, F, G ::=$	$\top$	**Null**
$\mid$	$E \wedge F$	**Composición**
$\mid$	$E^{(int)}$	**Parte interna o encapsulada del componente** E
$\mid$	$if(C_1 \cdots C_n)\ else\ G$	**Combinador de selección condicionada**
$\mid$	$x :: \overline{y}/E$	**Abstracción**
$\mid$	$x\overline{y}/E$	**Aplicación**
$\mid$	τ/E	**Reacción interna**
$\mid$	$\exists wE$	**Declaración**
$\mid$	$x : \overline{y}/E$	**Replicación de abstracción**
$\mid$	$E^{\top}$	**Ejecución exitosa del componente** E
$\mid$	$E^{\perp}$	**Ejecución no exitosa del componente** E
$\mid$	$OSO(E)\ do\ F\ else\ G$	**On Success Of**
$\mid$	$!OSO(E)\ do\ F\ else\ G$	**Observación repetida**
$\phi, \psi ::=$	$\dot{\top}$	**Verdad Lógica**
$\mid$	$\dot{\perp}$	**Falsedad Lógica**
$\mid$	$x = y$	**Restricción ecuacional**
$\mid$	$\phi \dot{\wedge} \psi$	**Conjunción de restricciones**
$\mid$	$\dot{\exists} x\phi$	**Cuantificador existencial**

Tabla 1: Sintaxis del $cálculo - \rho_{arq}$
Fuente: [Niehren and Muller, 1995];[Smolka, 1994a] ; [Smolka, 1994b] ; [Milner, 1999] y este trabajo.

- La **abstracción** se puede interpretar como el componente E con entrada $\overline{y}$ a lo largo de x; es decir, recibir una entidad simbólica a lo largo de x que podrá reemplazar a $\overline{y}$ en el componente E, siempre y cuando esa entidad simbólica recibida sea libre en el ámbito del componente E, donde es reemplazada.

- La **aplicación** $x\overline{y}/E$ se interpreta como enviar $\overline{y}$ a lo largo de x y continuar con la ejecución de E. Esta entidad sintáctica se asimila al prefijo de salida usado por Milner en el $cálculo - \pi$ [Milner, 1999] y varía el concepto de aplicación expresado en el $cálculo - \rho$ [Niehren and Muller, 1995] al asociar el envío de un mensaje a lo largo de un canal asociado a un componente, en este caso E. La expresión $x\overline{y}/\top$ se abreviará con $x\overline{y}$.

- Una **reacción interna** representada por τ/E no tiene su contraparte explícita en el $cálculo - \rho$ original; aún así, desde la perspectiva de chequeo de modelos es importante contar con esta entidad sintáctica para posibles reducciones en el número de estados a abordar en los análisis de propiedades dinámicas, donde puede requerirse especificar ciertas transiciones como reacciones internas para reducir el número de reacciones observables [Antonia Bertolino and Muccini, 2003].

- La **declaración** $\exists wE$ introduce una referencia w con alcance E.

- La **replicación de abstracción** $x : \overline{y}/E$ se puede expresar de la forma:

  $x : \overline{y}/E \equiv x :: \overline{y}/E \ \wedge \ x : \overline{y}/E$

  que permite ejemplificar componentes; es decir, se genera una nueva **abstracción** lista para reaccionar y se queda listo para replicar otra si es necesario.

  No obstante, la definición de las reglas de reducción en el $cálculo - \rho_{arq}$ mostrarán que una de ellas puede reemplazar esta entidad sintáctica (**regla aplicación**).

- Los términos $E^{\top}$ y $E^{\perp}$ permiten reflejar el éxito o fallo en la ejecución de componentes de software, respectivamente.

- La expresión $OSO(E)\ do\ F\ else\ G$ permite resolver la observación de la ejecución del componente E; si el componente es exitoso en su ejecución se resuelve a la expresión arquitectural F, en caso contrario se resuelve a G. Esta expresión puede permitir observaciones sucesivas de éxitos o fracasos en la ejecución de un componente si se expresa como $!OSO(E)\ do\ F\ else\ G$

  Las demás expresiones tienen la misma semántica del $cálculo_{\rho}$ original.

3. LA SEMÁNTICA OPERACIONAL

El $cálculo - \rho_{arq}$ mantiene la propiedad del $cálculo - \rho$ respecto a que restricciones, aplicaciones, abstracciones y condicionales pueden combinarse libremente por medio de composición y declaración.

La Tabla 2 presenta los axiomas de congruencia estructural; variables ligadas son introducidas como argumentos formales de las abstracciones y por las declaraciones. [5]

Los axiomas ACI (Asociatividad, Conmutatividad e Identidad), $Intercambio$, $Alcance$ y $Equiv.Restricciones$ son los convencionalmente propuestos en el $cálculo_{\rho}$ original.

[5]Las variables que no están ligadas se denomina variables libres. $\mathcal{FV}(E)$ y $\mathcal{BV}(E)$ se usan para denotar el conjunto de variables libres y variables ligadas, respectivamente, en E.

$(\alpha - conversión)$	Cambio de referencias ligadas por referencias libres
(ACI)	$\wedge$ es asociativa, conmutativa y satisface $E \wedge \top \equiv E$
$(Intercambio)$	$\exists x \exists y E \equiv \exists y \exists x E$
$(Alcance)$	$\exists x\ E \wedge F \equiv \exists x (E \wedge F) \quad si\ x \notin \mathcal{FV}(F)$
$(Equiv.Restricciones)$	$\phi \equiv \psi \quad si\ \phi \biguplus_{\Delta} \psi$ y $\mathcal{FV}(\phi) = \mathcal{FV}(\psi)$
$(Replicación\ de\ observación)$	$!OSO(E)\ do\ F\ else\ G \equiv OSO(E)\ do\ F\ else\ G \wedge !OSO(E)\ do\ F\ else\ G$
$(Éxito/Fracaso\ Observacional)$	$[v/w]E^{(int)} \equiv \dot{\top} \wedge if\ [\ (\dot{\top}\ then\ E^{\top})\ ,\ (\dot{\top}\ then\ E^{\bot})\ ]\ else\ (\ \top\ )$

Fuente [Niehren and Muller, 1995] y este trabajo

Tabla 2: Axiomas de congruencia estructural $cálculo - \rho_{arq}$

El axioma de *Replicación de observación* permite efectuar observaciones sucesivas de ejecución de un componente, cuando sea conveniente hacerlo.

La vista puramente observacional de los componentes en una arquitectura de software, lleva a proponer el axioma de $Éxito/Fracaso\ Observacional$ que usa el combinador de selección condicionada para modelar el posible resultado no-determinista de efectuar reemplazos de las entradas en un componente y luego ejecutarlo. En este caso se puede reducir a una ejecución exitosa del componente simbolizada por $E^{\top}$ o a una ejecución no exitosa del componente, simbolizada por $E^{\bot}$.

Para los propósitos de hacer más fácil la lectura de este documento, se abusa de la notación para reflejar el uso de este axioma de aquí en adelante, usando:

$[v/w]E^{(int)} \overset{suc(E)}{\equiv} E^{\top}$; que se puede interpretar como ejecución exitosa de E al reemplazar w por v en éste

y,

$[v/w]E^{(int)} \overset{unsuc(E)}{\equiv} E^{\bot}$; que se puede interpretar como ejecución no exitosa de E al reemplazar w por v en éste.

La Tabla 3 presenta las reglas de reducción que representan la semántica operacional[6]

Como se puede observar, las reglas $A_{\rho_{arq}}$ y $Comb_{\rho_{arq}}$ son condicionadas por restricciones de contexto en ϕ, en cada caso si se liberan las posibles reducciones internas se obtendría:

- En $A_{\rho_{arq}}$ (**aplicación**) se ejecutarían concurrentemente una abstracción con una replicación que ejemplifica una aplicación; en este caso, se enviaría $\overline{z}$ a lo largo de x y se seguiría ejecutando F, que reaccionaría con la abstracción ejemplificada, la cual recibe $\overline{z}$ a través de x, la replicación queda lista para ejemplificar más abstracciones y se da el reemplazo $[\overline{z}/\overline{y}]$ en E. Desde esta perspectiva, la regla de reducción **aplicación** en el ρ_{arq} incluye **replicación** y haría innecesario la entidad sintáctica para este fin. Esta regla de reducción puede modelar llamadas de procedimientos por pasar parámetros actuales para reemplazar los formales. Obviamente, las condiciones laterales establecen que la restricción determina el **enlace** que une un componente (aplicación) con otro (abstracción) y que se debe cumplir la $\alpha - conversión$.

- La regla de reducción de combinación de restricciones (C_{ρ}) merece especial atención porque restricciones a nivel global de la arquitectura podrían obtener más información o ampliarse por este medio (Operador $Tell$, tal como se propone en el modelo de computación por restricciones [Saraswat, 1992]).

- La regla $Comb_{\rho_{arq}}$ dispara la ejecución de un E_k si la restricción de contexto es suficientemente fuerte y permite deducir desde ϕ la guarda ψ_k del condicional respectivo. Ahora, si existe simultáneamente, en la expresión del combinador, otra cláusula con un ψ_j con $j \neq k$ deducible desde ϕ se presenta una situación de no-determinismo que debe ser manejada. Se puede percibir en esta regla de reducción la posibilidad de escogencia de un componente de un grupo de componentes candidatos. Se debe usar un adecuado diseño de guardas de cláusulas para evitar no-determinismo.

  De otra parte, si la restricción de contexto arquitectural no es lo suficientemente fuerte para deducir desde ésta alguno de las guardas de las cláusulas, se generaría el comportamiento alterno general del componente F, que en algunos casos podría ser el de manejo de fallas o errores, esto es consecuencia de que algún ψ_k no sea consistente con la restricción de contexto ϕ. Estas dos reglas de reducción, desde la perspectiva arquitectural, se pueden asociar a restricciones del contexto global de la arquitectura de referencia que controlan (si son suficientemente fuertes) el comportamiento de componentes/conectores de la misma en tiempo de ejecución.

- La propuesta de la regla $Ejec_{\tau}$ o de éxito/fracaso observacional se debe a que el interés de este trabajo no va más allá de especificar arquitecturas de software como ensamble de componentes vistos como cajas negras con puertos de entrada y de salida; cualquier entrada a un componente que se hace efectiva mediante el reemplazo de sus parámetros de entrada por valores obtenidos como servicios provistos por otros componentes se expresan como $[v/w]E^{(int)}$ (v *que reemplaza a* w *en el componente* E con $v \notin \mathcal{VF}(E^{(int)})$ o alfaconversión), la representación del éxito o fracaso de la ejecución se hará usando la regla de reducción denominada $Ejec_{\tau}$ o de $Éxito/Fracaso\ Observacional$ porque es producto de lo que ocurre internamente en el

[6]El operador de reemplazo $[\overline{z}/\overline{y}]$ requiere implícitamente que $\overline{z}$ y $\overline{y}$ tengan la misma longitud y que $\overline{y}$ sea lineal; es decir, que todos los elementos de $\overline{y}$ sean distintos entre sí.

$(A_{\rho_{arq}}) \quad \phi \wedge x:\overline{y}/E \wedge x'\overline{z}/F \longrightarrow \phi \wedge x:\overline{y}/E \wedge [\overline{z}/\overline{y}]E^{(int)} \wedge F \quad si\ \phi \models_\Delta x = x', \mathcal{V}(\overline{z}) \cap \mathcal{BV}(E^{(int)}) = \emptyset$
$(C_{\rho_{arq}}) \quad \phi_1 \wedge \phi_2 \longrightarrow \psi \qquad si\ \phi_1 \hat{\wedge} \phi_2 \models\!\!\models_\Delta \psi$
$(Comb_{\rho_{arq}}) \quad \phi \wedge if\ (C_1) \ldots (C_n)\ else\ F\ fi \longrightarrow \begin{cases} E_k, & si\ \phi \models_\Delta \psi_k \\ F, & si\ \phi \models_\Delta \neg\psi_k\ ; \forall k = 1,2,\ldots,n \end{cases}$ $Con\ \ C_k ::= \exists\overline{x}(\psi_k\ Then\ E_k)\ ;\ k = 1,2,\ldots,n$
$(Ejec_\tau)$ (a) $[OSO(E)\ do\ F\ else\ G] \wedge E^\top \longrightarrow F$, *debido a que hay ejecución exitosa del componente*   (b) $[OSO(E)\ do\ F\ else\ G] \wedge E^\perp \longrightarrow G$, *debido a que no hay ejecución exitosa del componente*

Fuente [Niehren and Muller, 1995] y este trabajo

Tabla 3: Reglas de reducción del $cálculo - \rho_{arq}$

$\frac{E \longrightarrow F}{\exists x E \longrightarrow \exists x F}$	$\frac{E \longrightarrow F}{E \wedge G \longrightarrow F \wedge G}$	$\frac{E_1 \equiv E_2 \quad E_2 \longrightarrow F_2 \quad F_1 \equiv F_2}{E_1 \longrightarrow F_1}$

Tabla 4: Reducciones sobre los componentes

componente y que no es visible al observador externo del mismo.

Como se puede observar, una expresión **On Success Of** (regla OSO) en composición con un componente que se ejecuta de manera exitosa reduce a la ejecución de una expresión arquitectural F, que puede representar el caso exitoso del flujo arquitectural en la configuración; en otro sentido, si la observación de la ejecución del componente resulta en un caso de fallo, se genera la ejecución del componente G que se puede asociar al manejo de errores.

Los axiomas y esta nueva regla de reducción permiten especificar formalmente el avance en la ejecución de una arquitectura, como se verá más adelante.

3.1 Reducciones en componentes

Ahora, luego de acceder a los componentes $E, F, G, ...$ (satisfacer las restricciones) se podrían dar reducciones que respetan las reglas de inferencia presentadas en la Tabla 4.

3.2 La ausencia de estado en el $cálculo - \rho_{arq}$

Ni en las entidades sintácticas ni en la semántica operacional del $cálculo - \rho_{arq}$ se usa el concepto de celda introducido en el $cálculo - \rho$ original. Esto es consecuencia de considerar un componente como una entidad sin estado, que se ciñe más a un modelo de computación declarativo que hace una vista observacional, regida por principios de abstracción que modelan un componente como una caja negra que recibe entradas y garantiza producir las mismas salidas ante los mismos valores de dichas entradas.

4. ASPECTOS DINÁMICOS Y DE CONFIGURACIÓN ARQUITECTURALES

Con el objeto de expresar configuraciones arquitecturales, se propone a continuación una notación gráfica consistente y traducible al $cálculo - \rho_{arq}$. Esta propuesta se inspira en tres fuentes principales:

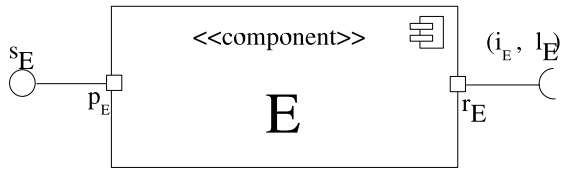

Figura 1: Notación gráfica de componente

1. La traducción de la notación visual del lenguaje de configuración Darwin al $cálculo - \pi$ [Jeff Magee and Kramer, 1995].

2. La notación gráfica de componentes de software propuesta en UML 2.0 [Object Management Group, 2007] con algunas mejoras para hacerla más formal y completa respecto al tema de arquitecturas de software y traducibilidad a una especificación formal en el $cálculo - \rho_{arq}$.

4.1 La notación de componente

Para el problema de especificación que se ha abordado, un componente se puede considerar como un proceso encapsulado que sólo expone al exterior sus puertos de entrada y sus puertos de salida; siempre se podrán alambrar componentes conectando puertos de entrada de uno con los puertos de salida de otro y cuidando que exista compatibilidad entre las interfaces que representan dichos puertos.

Un componente se representa como lo muestra la Figura 1, un rectángulo rotulado con << *component* >> en la parte superior y con el nombre del componente en el centro del mismo; opcionalmente, un ícono en la esquina superior derecha que en anteriores versiones de UML se interpretaba como un componente. Cuadrados que sobresalen de los bordes representan puertos públicos [7] y poseen un nombre de acceso con el subíndice indicando el componente al

[7] Si un puerto es protegido, se colocará el cuadrado en la parte interna de la frontera del rectángulo

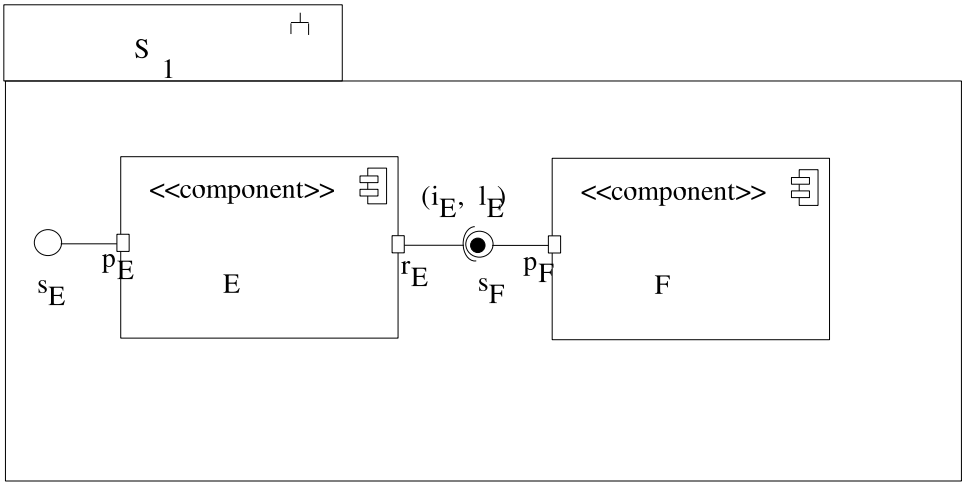

Figura 2: Notación gráfica de ensamble de componentes

que pertenece; los puertos se pueden interpretar como los puntos o nombres de acceso que se usan para acceder a un servicio ofrecido o a una locación que requiere alguno, en el ejemplo de la Figura 1 se usan las letras p y r con un subíndice que indica el componente correspondiente (ejemplo: p_E). A cada servicio sólo corresponderá un nombre de acceso y si hay más de un servicio ofrecido por el componente se puede igualmente numerar el subíndice (ejemplo: p_{E1}); un componente puede tener varios puertos con una interfaz de entrada o de salida asociada a cada uno, la semántica de dichas interfaces se explica a continuación.

4.1.1 Interfaz de salida (*Provided Interface*)

Una interfaz de salida se representa con una línea recta continua saliendo de un puerto y terminando en un círculo cerrado no relleno; dicha interfaz de salida se denominará también lugar de salida o de provisión de servicio. Éste tendrá el nombre o referencia del servicio que se provee, identificado por una s con un subíndice que indica el componente al que corresponde (ejemplo: s_E), si hay más de un servicio a referenciar se puede numerar en el subíndice (ejemplo: s_{E1}).

4.1.1.1 Especificación formal en el *cálculo* $-\rho_{arq}$..

Para el componente E que se presenta en la Figura 1 se denominará $PROV_E(p,s)$ y se definiría formalmente por:

$$PROV_E(p,s) \stackrel{def}{=} p_E : x/xs_E \equiv p_E :: x/xs_E \wedge p_E : x/xs_E$$

correspondiente a la entidad sintáctica replicación y se interpretaría como: *se espera a lo largo de p_E un valor de una locación a la cual se enviará el servicio s_E*; al usar replicación es posible atender muchos clientes que requieran este servicio.

4.1.2 Interfaz de entrada (Required Interface)

Consiste en una línea recta continua saliendo del rectángulo y terminada en un semicírculo abierto; también se denominará **lugar requisitor de servicio** o **lugar de entrada**, éste llevará sobre el semicírculo abierto una pareja expresada como (i_E, l_E) que indica una ubicación l_E que espera recibir un valor que pueda reemplazar el parámetro i_E en el componente E.

4.1.2.1 Especificación formal en el *cálculo* $-\rho_{arq}$.

Para el componente que se presenta en la Figura 1 se denominará $REQ_E(r,l,i)$ y se definiría por:[8]

$$REQ_E(r,l,i) \stackrel{def}{=} \exists l_E[(r_E :: y/yl_E) \wedge (l_E :: i_E/E^{(int)})]$$

que se puede interpretar como: *se espera a lo largo de r_E un nombre de acceso a un servicio, que al ser aplicado a la ubicación l_E que es local a la expresión (o sea al componente que pertenece), externalice la ubicación para recibir a lo largo de ésta el servicio solicitado*, que al ser recibido a lo largo de l_E podrá reemplazar el parámetro de entrada i_E en $E^{(int)}$, que representa la parte interna del componente[9].

Por consiguiente, desde la perspectiva de definición de componente, se puede plantear que sería suficientemente representado por sus interfaces públicas de entrada y salida actuando concurrentemente con la abstracción que represente la incorporación de servicios al mismo; es decir, hasta ese punto habría visibilidad para un observador del componente. En el caso que se viene ejemplificando, el componente E se puede definir como:

$$E \stackrel{def}{=} PROV_E(p,s) \wedge REQ_E(r,l,i)$$

[8]La posibilidad de introducir valores a un componente a través de parámetros expuestos por medio de interfaces públicas se modela diferenciando la parte pública de un componente E, que especifica las interfaces públicas, de la parte interna de un componente, $E^{(int)}$. Este enfoque evita el uso de recursión y establece claramente la diferencia entre lo externo (público) y lo interno de un componente. El nivel de abstracción arquitectural, que se ha venido tratando en este trabajo, supone que la parte interna de un componente puede estar implementada en diversos paradigmas de programación y está completamente encapsulada.

[9]Como se puede observar, el puerto de requerimiento de servicio queda inhabilitado al usar la expresión abstracción y no la replicación de abstracción. Esto se hace para simplificar el análisis. En trabajos futuros se abordará la reconexión dinámica dejando habilitados los puertos de requerimiento para seguir recibiendo entradas

4.2 La notación de ensamble de componentes y representación de aspectos dinámicos

Un componente adquiere utilidad práctica cuando interactúa con otros componentes, para esto debe ocurrir que servicios ofrecidos en un lugar sean conectados y concretados en una ubicación que los requiere. La notación gráfica que representará, de ahora en adelante, el ensamble de componentes será similar a la que ilustra la Figura 2.

Como se puede observar en la Figura 2, se tienen dos componentes:

$$E \stackrel{def}{=} [(p_E : x/xs_E)] \wedge \exists l_E[\,(r_E :: y/yl_E) \wedge (l_E :: i_E/E^{(int)})]$$

$$F \stackrel{def}{=} (p_F : z/zs_F)$$

4.2.0.2 *Especificación formal en el* $cálculo - \rho_{arq}$.

Para conectar activamente E y F se requiere un conector que logre dar acceso al servicio; este conector de ensamble se denominará C_{FE} y se especifica como:

$C_{FE} \stackrel{def}{=} r_E p_F$

que al actuar en forma concurrente (composición), hará que el sistema S_1 representado por esta configuración inicie una dinámica de ejecución, que se puede especificar así:

$S_1 = E \wedge F \wedge C_{FE}$
$= \{(p_E : x/xs_E) \wedge \exists l_E\ [(r_E :: y/yl_E) \wedge (l_E :: i_E/E^{(int)})]\} \wedge \{(p_F : z/zs_F)\} \wedge \{r_E p_F\}$

Ahora, si se aplican los axiomas de congruencia estructural y las reglas de reducción del $cálculo - \rho_{arq}$ sin presencia de restricciones (que se incorporarán más adelante), se tiene[10]:

$$S_1 \stackrel{A_{\rho arq}}{\Longrightarrow} p_E : x/xs_E \wedge [s_F/i_E]E^{(int)}$$

Lo anterior muestra el proceso de ejecución que lleva el servicio(*token*) s_F a la ubicación l_E que lo requiere, es decir, el flujo del *token* indica la secuencia de ejecución en la arquitectura; surge entonces la necesidad de expresar esta secuencia de ejecución.

4.3 Control de ejecución arquitectural

El flujo de *tokens* o servicios en una arquitectura basada en componentes representa su dinámica en tiempo de ejecución; en la anterior sección se pudo observar que al ejecutar F se provee un servicio[11] que es requerido por E; ahora, si E está conectado a su vez a un componente que requiere de sus servicios, se requeriría especificar formalmente el flujo de ejecución. Al considerar E como un componente, no podemos observar más allá de sus entradas y salidas, el flujo de ejecución será especificado a partir de este nivel de observabilidad; por consiguiente, en el ejemplo ilustrativo que nos ocupa, cuando se llega a $[s_F/i_E]E$ lo único que se podría detectar es que el componente provea el servicio normalmente (genera el *token* en sus puntos de salida) o que indique una falla en su ejecución. Si se produce el *token* en su(s) punto(s) de salida o de provisión de servicio, se especifica activando el conector correspondiente para continuar la ejecución de los componentes subsiguientes; en otro caso, se debe entregar un mensaje de error a un componente manejador de errores. Una forma de expresar esta situación es usando la regla $Ejec_\tau$:

$$[s_F/i_E]E^{(int)} \wedge [OSO(E)$$
$$do\ (activar\ conectores\ de\ ensamble\ )$$
$$else\ (activar\ conector\ de\ manejo\ errores)]$$

Se retoma el ejemplo ilustrativo que se viene trabajando, ampliándolo con un posible componente T que requiere un servicio de E para ejecutarse y un componente manejador de errores (ver Figura 3). Cada componente tiene como especificación en el $cálculo - \rho_{arq}$:

$F \stackrel{def}{=} (p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})$
$E \stackrel{def}{=} (p_E : x/xs_E) \wedge (p_{Ee} : v/vs_{Ee}) \wedge \exists l_E[(r_E :: y/yl_E \wedge (l_E :: i_E/E^{(int)})]$
$M \stackrel{def}{=} \exists l_M[r_M : y/yl_M \wedge (l_M : i_M/M^{(int)})]$
$T \stackrel{def}{=} (\exists l_T[r_T :: q/ql_T \wedge (l_T : i_T/T^{(int)})]) \wedge (p_{Te} : n/ns_{Te})$

Antes de proseguir se introducen dos conceptos:

- **Componente fuente**. Es aquel que sólo posee interfaces de salida o de provisión de servicio; en nuestro ejemplo, el componente F.
- **Componente sumidero**. Es aquel que sólo posee interfaces de entrada o requisitoras de servicio. En nuestro ejemplo, es de este tipo el componente M o manejador de errores.

Se puede corroborar que el flujo de ejecución es modelable desde el $cálculo - \rho_{arq}$ usando el ejemplo de configuración arquitectural del sistema S_2 presentado en la Figura 3; suponiendo que el componente fuente F se ejecuta exitosamente[12], la especificación de la configuración quedaría:

$$S_2^{(0)} = F^\top \wedge [\ OSO(F)\ do\ F \wedge C_{FE} \wedge E\ else\ F \wedge C_{FM} \wedge M\}\ ] \wedge [\ OSO(E)\ do\ C_{ET} \wedge T\ else\ C_{EM} \wedge M\ ] \wedge [\ OSO(T)\ do\ S_2 = \acute{e}xito\ else\ C_{TM} \wedge M\ ]$$

Inicialmente se supone que todos los componentes trabajan correctamente; es decir, hay éxito en su ejecución, se obtendría en primera instancia que el componente F es exitoso:[13]

$$S_2^{(0)} \stackrel{Ejec_\tau}{\longrightarrow} [\ F \wedge C_{FE} \wedge E\ ] \wedge [\ OSO(E)\ do\ C_{ET} \wedge T\ else\ C_{EM} \wedge M\} \wedge [\ OSO(T)\ do\ S_2 = \acute{e}xito\ else\ C_{TM} \wedge M\ ]$$

efectuando los reemplazos correspondientes, se obtiene:

[10] Para evitar verbosidad formal y debido al limitado espacio en el documento, se introduce el símbolo relación $\stackrel{A_{\rho arq}}{\Longrightarrow} \equiv \stackrel{A_{\rho arq}}{\longrightarrow} \ldots \stackrel{A_{\rho arq}}{\longrightarrow}$ [Diosa, 2005].

[11] Se debe tener en cuenta que un servicio obtenido se ve representado normalmente por una instancia de un tipo de dato que entra al componente que lo requirió para propósitos específicos de ejecución

[12] Para diferentes momentos, donde la configuración arquitectural varía en cuanto a la activación o no de conectores de ensamble, se representará al sistema con un superíndice para diferenciar uno de otro en la ejecución del mismo sistema.

[13] Al ser F un componente fuente, se puede asimilar a una interfaz de captura de datos certificada para garantizar un funcionamiento correcto

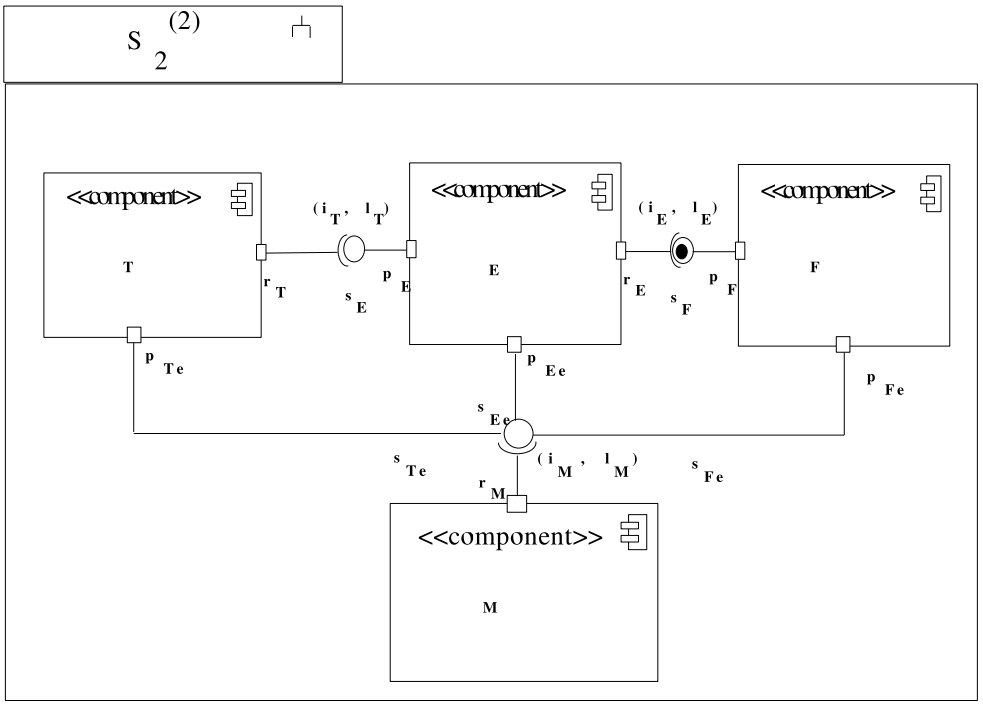

Figura 3: Ensamble complejo de componentes. Momento inicial: captura de datos exitosa

$S_2^{(1)} = \{[(p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [r_E p_F] \wedge [(p_E : x/xs_E) \wedge (\exists l_E[r_E :: y/yl_E \wedge (l_E :: i_E/E^{(int)})]) \wedge (p_{Ee} : v/vs_{Ee})] \} \wedge [\, OSO(E)\ do\ C_{ET} \wedge T\ else\ C_{EM} \wedge M\} \wedge [\, OSO(T)\ do\ S_2 = \acute{e}xito\ else\ C_{TM} \wedge M\,]$

$\stackrel{A_{\rho arq}}{\longrightarrow} \{[(p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [(p_E : x/xs_E) \wedge \exists l_E[(p_F l_E) \wedge (l_E :: i_E/E^{(int)})] \wedge (p_{Ee} : v/vs_{Ee})] \} \wedge [\, OSO(E)\ do\ C_{ET} \wedge T\ else\ C_{EM} \wedge M\} \wedge [\, OSO(T)\ do\ S_2 = \acute{e}xito\ else\ C_{TM} \wedge M\,]$

$\stackrel{A_{\rho arq}}{\longrightarrow} \{[(p_F : z/zs_F) \wedge (l_E s_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [(p_E : x/xs_E) \wedge (p_{Ee} : v/vs_{Ee}) \wedge (l_E :: i_E/E^{(int)})] \} \wedge [\, OSO(E)\ do\ C_{ET} \wedge T\ else\ C_{EM} \wedge M\} \wedge [\, OSO(T)\ do\ S_2 = \acute{e}xito\ else\ C_{TM} \wedge M\,]$

$\stackrel{A_{\rho arq}}{\longrightarrow} \{[(p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [(p_E : x/xs_E) \wedge (p_{Ee} : v/vs_{Ee}) \wedge ([s_F/i_E]E^{(int)})] \} \wedge [\, OSO(E)\ do\ C_{ET} \wedge T\ else\ C_{EM} \wedge M\} \wedge [\, OSO(T)\ do\ S_2 = \acute{e}xito\ else\ C_{TM} \wedge M\,]$

esta primera fase de ejecución coloca el servicio s_F listo para reemplazar el parámetro de entrada i_E en el componente E, lo cual genera ejecución de dicho componente por tener copadas sus entradas; luego, de los axiomas de congruencia estructural de éxito/fracaso observacional y bajo la suposición de que hay ejecución exitosa de E:

$S_2^{(2)} \stackrel{suc(E)}{\equiv} \{[(p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [(p_E : x/xs_E) \wedge (p_{Ee} : v/vs_{Ee}) \wedge (E^{\top})] \} \wedge [\, OSO(E)\ do\ C_{ET} \wedge T\ else\ C_{EM} \wedge M\} \wedge [\, OSO(T)\ do\ S_2 = \acute{e}xito\ else\ C_{TM} \wedge M\,]$

$\stackrel{Ejec_T}{\longrightarrow} \{[(p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [(p_E : x/xs_E) \wedge (p_{Ee} : v/vs_{Ee})] \} \wedge \{ C_{ET} \wedge T\} \wedge [\, OSO(T)\ do\ S_2 = \acute{e}xito\ else\ C_{TM} \wedge M\,]$

Aplicar la nueva secuencia de reducciones lleva el sistema a una nueva configuración, la mostrada en la Figura 4, donde el *token* se ha desplazado al punto de entrada del componente T[14], dicha secuencia, haciendo los reemplazos correspondientes, sería:

$S_2^{(3)} = \{[(p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [(p_E : x/xs_E) \wedge (p_{Ee} : v/vs_{Ee})] \} \wedge [r_T p_E] \wedge \{ \exists l_T[r_T :: q/ql_T \wedge (l_T : i_T/T^{(int)})] \wedge (p_{Te} : n/ns_{Te})]\} \wedge [\, OSO(T)\ do\ S_2 = \acute{e}xito\ else\ C_{TM} \wedge M\,]$

$\stackrel{A_{\rho arq}}{\longrightarrow} \{[(p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [(p_E : x/xs_E) \wedge (p_{Ee} : v/vs_{Ee})] \} \wedge \{ \exists l_T[(p_E l_T \wedge (l_T : i_T/T^{(int)})] \wedge [(p_{Te} : n/ns_{Te})]\} \wedge [\, OSO(T)\ do\ S_2 = \acute{e}xito\ else\ C_{TM} \wedge M\,]$

$\stackrel{A_{\rho arq}}{\longrightarrow} \{[(p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [(p_E : x/xs_E) \wedge (l_T s_E) \wedge (p_{Ee} : v/vs_{Ee})] \} \wedge \{ [(l_T : i_T/T^{(int)}) \wedge (p_{Te} : n/ns_{Te})]\} \wedge [\, OSO(T)\ do\ S_2 = \acute{e}xito\ else\ C_{TM} \wedge M\,]$

$\stackrel{A_{\rho arq}}{\longrightarrow} \{[(p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [(p_E : x/xs_E) \wedge (p_{Ee} : v/vs_{Ee})] \} \wedge \{ [(l_T : i_T/T) \wedge ([s_E/i_T]T^{(int)}) \wedge (p_{Te} : n/ns_{Te})]\} \wedge [\, OSO(T)\ do\ S_2 = \acute{e}xito\ else\ C_{TM} \wedge M\,]$

[14] Un lector atento podrá observar que E en la especificación formal se inhabilita para recibir por sus puntos de entrada; esto se debe a que se usó aplicación por una sóla vez y no la replicación para los puertos de entrada

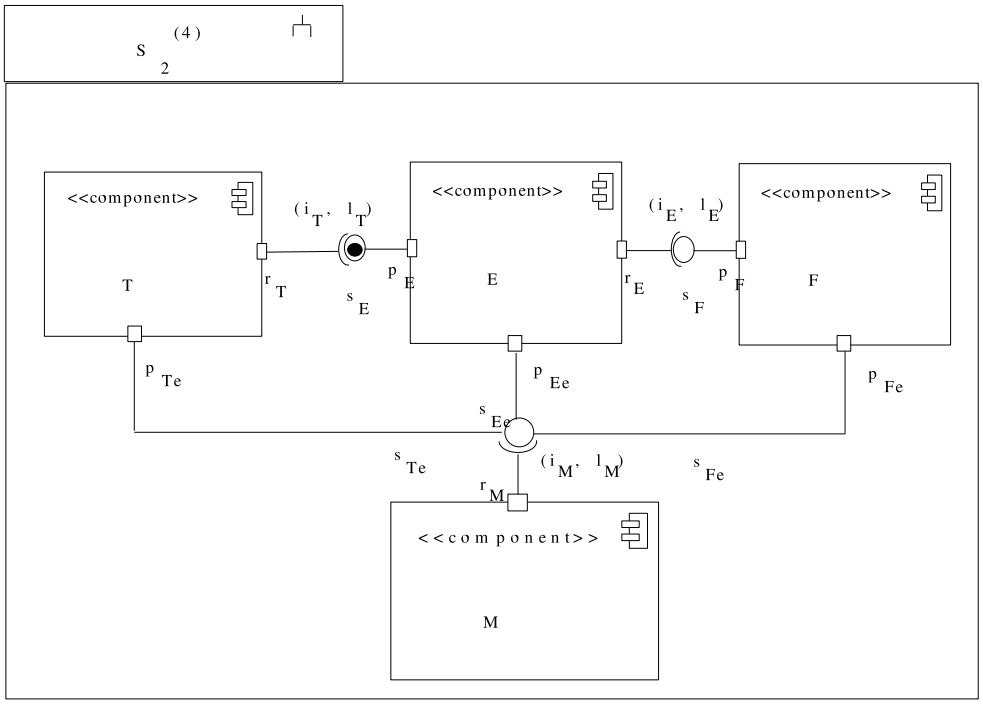

Figura 4: Ensamble complejo de componentes. Momento: activación conector entre E y T

la expresión $([s_E/i_T]T^{(int)})$ indica que se ha provisto el servicio s_E que reemplaza el parámetro de entrada i_T en el componente T. Ahora, el componente T puede ser exitoso, es decir, proveer la funcionalidad requerida al usuario final o fallar, entregar un mensaje al componente manejador de errores. En el primer caso, será visible una ejecución exitosa del sistema S_2 por medio del componente T; en caso contrario, se activará el conector que enlaza el componente con el manejador de errores para proveer a éste del servicio de información de errores s_{Te}, que simplemente puede ser un mensaje de error al mismo. Nos colocaremos en el segundo escenario de error:

$$S_2^{(4)} \overset{unsuc(T)}{\equiv} \{[(p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [(p_E : x/xs_E) \wedge (p_{Ee} : v/vs_{Ee})] \} \wedge \{ [(l_T : i_T/T) \wedge (T^{\perp}) \wedge (p_{Te} : n/ns_{Te})]\} \wedge [\, OSO(T)\ do\ S_2 = \acute{e}xito\ else\ C_{TM} \wedge M\,]$$

$$\overset{Ejec_\tau}{\longrightarrow} \{[(p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [(p_E : x/xs_E) \wedge (p_{Ee} : v/vs_{Ee})] \} \wedge \{ [(l_T : i_T/T) \wedge (p_{Te} : n/ns_{Te})]\} \wedge \{ r_M p_{Te} \wedge \exists l_M[(r_M : y/yl_M) \wedge (l_M : i_M/M^{(int)})]\}$$

$$\overset{A_{\rho arq}}{\longrightarrow} \{[(p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [(p_E : x/xs_E) \wedge (p_{Ee} : v/vs_{Ee})] \} \wedge \{ [(l_T : i_T/T) \wedge (p_{Te} : n/ns_{Te})]\} \wedge \{ \exists l_M[(r_M : y/yl_M) \wedge (l_M : i_M/M^{(int)})]) \wedge \exists l_M[(p_{Te} l_M) \wedge (l_M : i_M/M^{(int)})])\}$$

$$\overset{A_{\rho arq}}{\longrightarrow} \{[(p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [(p_E : x/xs_E) \wedge (p_{Ee} : v/vs_{Ee})] \} \wedge \{ [(l_T : i_T/T) \wedge (p_{Te} : n/ns_{Te}) \wedge (l_M s_{Te})]\} \wedge \{ (r_M : y/\exists l_M[yl_M \wedge (l_M : i_M/M^{(int)})])\}$$

$$\overset{A_{\rho arq}}{\longrightarrow} \{[(p_F : z/zs_F) \wedge (p_{Fe} : w/ws_{Fe})] \wedge [(p_E : x/xs_E) \wedge (p_{Ee} : v/vs_{Ee})] \} \wedge \{ [(l_T : i_T/T) \wedge (p_{Te} : n/ns_{Te})]\} \wedge \{ (r_M : y/\exists l_M[yl_M) \wedge (l_M : i_M/M^{(int)})]) \wedge ([s_{Te}/i_M]M^{(int)})]\} = S_2^{(5)}$$

si se observa con atención, la interfaz de entrada al manejador de errores se replica para seguir atendiendo nuevos envíos de errores. La Figura 5 muestra una instantánea de esta configuración en presencia de error en T.

La regla de reducción $Ejec_\tau$ ha demostrado ser útil para modelar la secuencia de ejecución en una arquitectura basada en componentes. Pero, ¿qué pasa cuando un componente tiene varias interfaces de entrada y no puede ejecutarse hasta obtener todos los servicios de éstas?. En este caso se establece como guarda de la activación del conector o conectores de salida, la conjunción de las aplicaciones exitosas a través de las cuales se reemplazan los parámetros de entrada.

4.4 Configuración jerárquica o componentes compuestos

Ahora se abordarán configuraciones donde existen *mapeos* desde interfaces internas a interfaces externas porque un componente es compuesto.

4.4.1 *Especificación formal en el* $cálculo - \rho_{arq}$

Para formalizar estas configuraciones arquitecturales se deben considerar los puertos como componentes sin procesamiento interno pero con capacidad de redireccionar o reem-

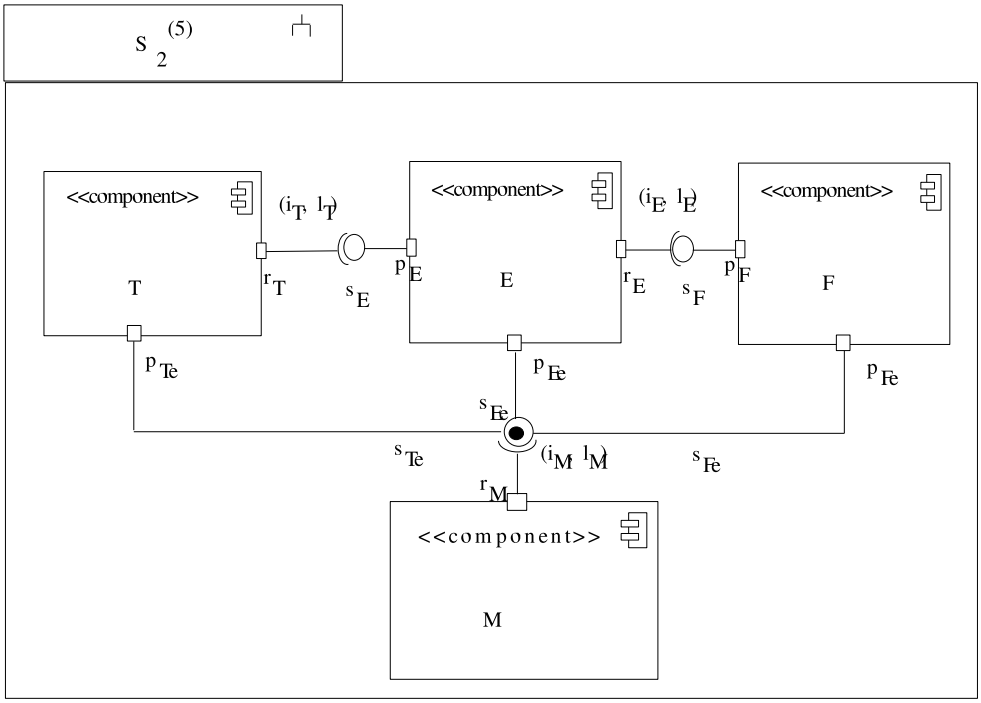

Figura 5: Ensamble complejo de componentes. Momento: error en componente T

plazar un servicio interno por uno obtenido desde lo externo, es decir, no aplica a éstos la ***regla*** $Ejec_\tau$; sin embargo, se introduce un término de redireccionamiento y de activación por defecto del conector de ensamble entre el puerto externo y el interno, por esta razón cuenta con cuatro parámetros representando información interna y externa al componente. Formalmente estarían definidos por una composición entre el formalismo que expresa una interfaz de requerimiento de servicio y una interfaz de provisión de servicio. Así, en la configuración ilustrada en la Figura 6, se definirían como:

$PROV_{jer}(p_W, s_W, r_{WA}, l_{WA}) \stackrel{def}{=} (r_{WA} : x/\exists l_{WA}[xl_{WA} \wedge l_{WA} :: s_W/(p_W : x/xs_W)])$

$\equiv (r_{WA} : x/\exists l_{WA}[xl_{WA} \wedge l_{WA} :: s_W/(p_W : x/xs_W)]) \wedge (r_{WA} :: x/\exists l_{WA}[xl_{WA} \wedge l_{WA} :: s_W/(p_W : x/xs_W)])$

y:

$REQ_{jer}(r_W, l_W, p_{WB}, s_{WB}) \stackrel{def}{=} r_W :: y/\exists l_W[yl_W \wedge l_W :: s_{WB}/(p_{WB} : x/xs_{WB})] \wedge r_B p_{WB}$

El componente compuesto W se podría especificar como:

$W \stackrel{def}{=} REQ_{jer}(r_W, l_W, p_{WB}, s_{WB}) \wedge B \wedge \{ OSO(B)\ do\ C_{BA} \wedge A\ else\ manejar\ fallo\} \wedge \{ OSO(A)\ do\ r_{WA}p_A\ else\ manejar\ fallo\} \wedge PROV_{jer}(p_W, s_W, r_{WA}, l_{WA})$

Se puede demostrar que si W es un componente más de una configuración dada y recibe un *token* en su interfaz de requerimiento, éste hara fluir el servicio hasta su interfaz de salida (ver [Diosa, 2008]).

4.5 Configuración de componentes alternativos o condicionados

Es posible la existencia de un grupo de componentes que ofrecen el mismo servicio y de un componente requisitor que puede escoger uno de ellos de acuerdo a restricciones arquitecturales establecidas desde el contexto en tiempo de ejecución. Este tipo de configuración se notará gráficamente como el ejemplo ilustrado en la Figura 7; aquí el componente sumidero A puede recibir el servicio desde varios proveedores(componentes fuentes): M, ... , N, la variable X en el conector de ensamble indica que el valor que ésta cargue determinará cuál de las opciones será tomada.

4.5.1 *Especificación formal en el cálculo* $-\ \rho_{arq}$

Para este tipo de configuraciones será de gran utilidad el **combinador de selección condicionada**. La especificación formal, partiendo de una ejecución exitosa de A, sería:

$$S_{alt} \stackrel{def}{=} A^\top \wedge [OSO(A)\ do\ \{ [\ if\ \exists X((X = p_M)\ then\ \{OSO(M)\ do\ [\ r_A p_M \wedge M\ ]\ else\ [manejar\ error\ de\ M]\}) , \cdots , ((X = p_N)\ then\ \{OSO(N)\ do\ [\ r_A p_N \wedge N\ ]\ else\ [manejar\ error\ de\ N]\})\ else\ manejar\ error\ fi\ ] \wedge A\}\ else\ \{manejar\ error\ de\ A\ \}$$

las expresiones formales para los componentes serían:

$A \stackrel{def}{=} REQ_A(r_A, l_A, i_A) \equiv r_A :: y/yl_A \wedge l_A :: i_A/A^{(int)}$

$M \stackrel{def}{=} PROV_M(p_M, s) \equiv p_M : x/xs$

$N \stackrel{def}{=} PROV_N(p_N, s) \equiv p_N : x/xs$

Ahora, se supone un escenario donde el contexto determina que el componente a escoger es M y no hay error, es

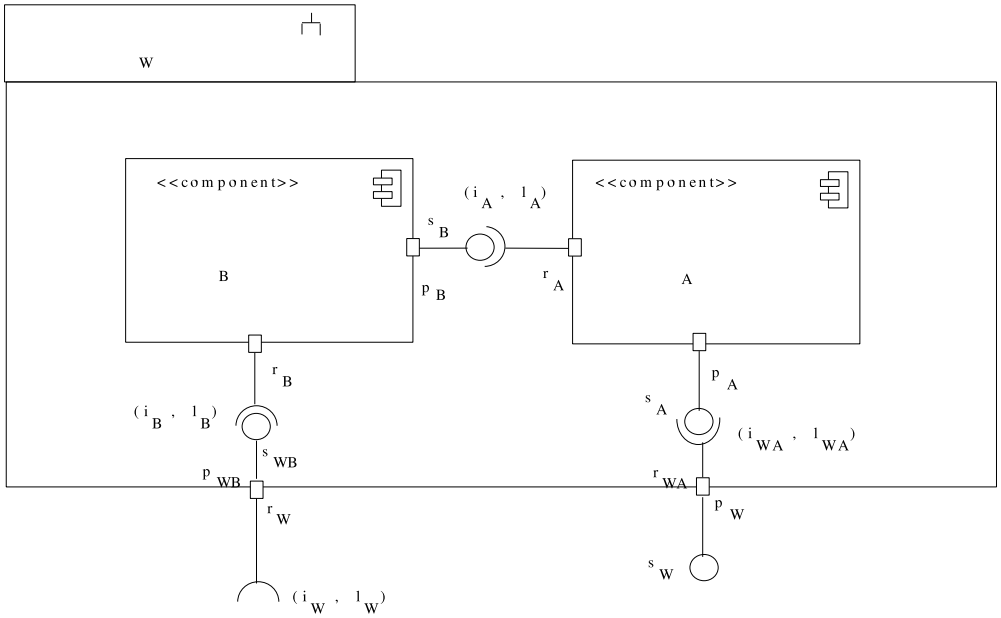

Figura 6: Mapeo de interfaces internas

decir, $X = p_M$ y $M^{\top}$[15]:

$$(M^{\top} \wedge X = p_M \wedge S_{alt}) \xrightarrow{Ejec_{\tau}} (M^{\top} \wedge X = p_M \wedge \{ [\, if\ \exists X((X = p_M)\ then\ \{OSO(M)\ do\ [\, r_A p_M \wedge M\,]\ else\ [manejar\ error\ de\ M]\}) , \cdots , ((X = p_N)\ then\ \{OSO(N)\ do\ [\, r_A p_N \wedge N\,]\ else\ [manejar\ error\ de\ N]\}) \quad else\ manejar\ error\,] \wedge A\})$$

$$\xrightarrow{Comb_{\rho_{arq}}} (M^{\top} \wedge \{OSO(M)\ do\ [r_A p_M \wedge M\,]\ else\ [manejar\ error\ de\ M]\} \wedge A\})$$

$$\xrightarrow{Ejec_{\tau}} \{ r_A p_M \wedge M \} \wedge A$$

Continuando con el ejemplo, se hacen los reemplazos correspondientes:

$$\{ r_A p_M \wedge M \} \wedge A \equiv \{ r_A p_M \wedge p_M : x/xs \} \wedge \{ r_A :: y/yl_A \wedge l_A :: i_A/A^{(int)} \}$$

$$\xrightarrow{A_{\rho_{arq}}} \{ p_M : x/xs \} \wedge p_M l_A \wedge l_A :: i_A/A^{(int)}$$

$$\xrightarrow{A_{\rho_{arq}}} \{ p_M : x/xs \} \wedge l_A s \wedge l_A :: i_A/A^{(int)}$$

$$\xrightarrow{A_{\rho_{arq}}} \{ p_M : x/xs \} \wedge [s/i_A]A^{(int)}$$

quedando la configuración final tal como se espera, el parámetro de entrada del componente A siendo reemplazado por el servicio s provisto por componentes alternativos. De manera análoga se podría activar el componente N en vez de M u otros componentes alternativos.

[15]Obsérvese que la expresión entre paréntesis, a la izquierda, puede corresponder a un **ASK** a un ***Global Store*** de restricciones; que podría ser el medio para gestionar arquitecturas en tiempo de ejecución.

5. CONCLUSIONES

1. El $cálculo - \rho_{arq}$ es un álgebra de procesos que abre otro camino alternativo para formalizar la especificación de sistemas de software a un alto nivel de abstracción como lo es el de configuración arquitectural. Esta herramienta formal permite la representación estática y dinámica de modelos de referencia y arquitecturas de referencia de software.

2. Se asimiló la jerga utilizada en el ámbito de arquitecturas de software a los formalismos del $cálculo - \rho_{arq}$ con una propuesta de estereotipado de la notación gráfica en UML 2.x [Object Management Group, 2007] para representar configuraciones de arquitecturas basadas en componentes.

3. Se ha posibilitado el modelado del flujo de ejecución arquitectural en un sistema de software basado en componentes que consumen y proveen servicios de acuerdo a criterios que el diseñador de software establece. Esta posibilidad de modelar comportamiento abre el camino para explotar el análisis de propiedades dinámicas.

6. TRABAJO FUTURO

1. La noción de observación de este cálculo, soportada en la expresión **On Success Of**, permitirá utilizar la teoría de de **equivalencia de observación** propuesta por Milner [Milner, 1999] [Sangiorgi and Walker, 2003] y la convertibilidad de su semántica operacional a **Sistemas de Transición Rotulados** para evaluar **corrección**. En publicaciones posteriores se divulgarán las bases conceptuales y teóricas de este enfoque.

2. Se viene trabajando en la extensión del marco de trabajo xADL 2.0 para que soporte conceptos del $cálculo-$

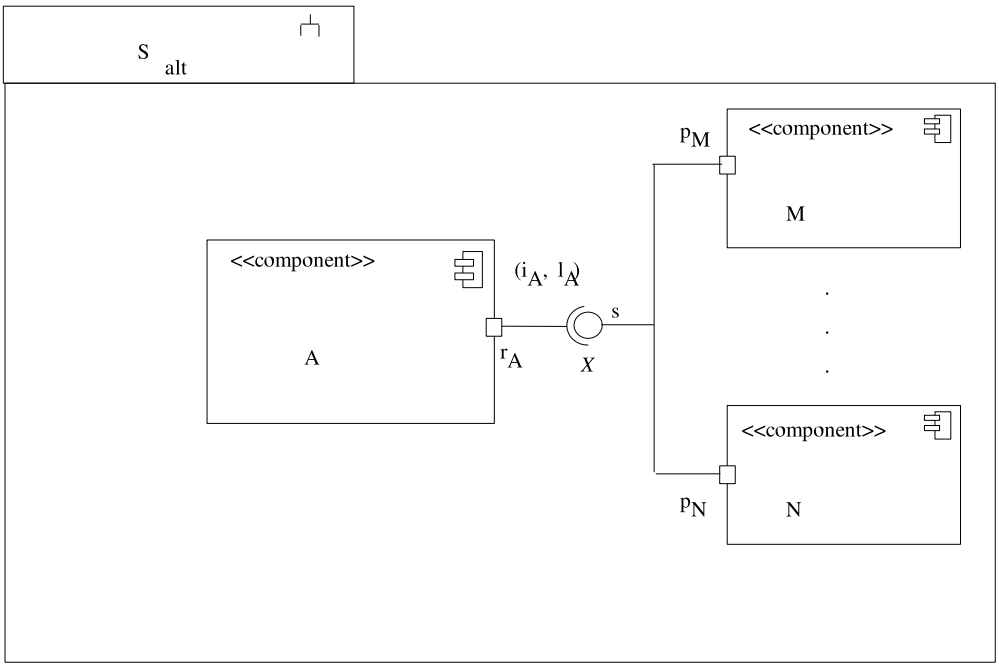

Figura 7: Configuración con componentes alternativos para proveer servicios

ρ_{arq} y hacer traducible especificaciones de modelos arquitecturales de software a formatos gráficos vectoriales SVG [W3C, 2003], en la primera versión de los resultados de este frente de trabajo no se incluye la posible especificación de ejecución arquitectural condicionada. Por este hecho, se viene vislumbrando la posibilidad de traducir y reducir las expresiones sobre una máquina abstracta para reflejar el flujo de ejecución en la especificación en xADL 2.0 traducible a SVG, de esta manera se podría visualizar gráficamente el flujo de ejecución arquitectural.

3. La traducibilidad de diagramas UML 2.x a especificaciones textuales conformes con XMI (XML Metadata Interchange) [W3C, 2007] ha madurado bastante a la fecha actual y posibilita un proyecto similar al que se viene trabajando con xADL 2.0.

4. Se ha detectado que el enfoque notacional ampliado desde la propuesta de UML 2.x [Object Management Group, 2007] se asimila o es análoga a la teoría de **redes de Petri** [Murata, 1989]. Esta coincidencia permite visionar futuros proyectos que apuntalen el uso de esta herramienta para el chequeo de propiedades dinámicas de arquitecturas de software basadas en componentes en el sentido expuesto en este trabajo.

5. Algo novedoso y útil en el tratamiento de especificaciones arquitecturales es el uso de restricciones que permiten condicionar la participación de componentes de software en posibles instancias arquitecturales ejemplificadas desde un modelo de referencia; esto posibilita la activación o desactivación de componentes de software, en tiempo de ejecución, desde un contexto controlado por restricciones almacenadas en un repositorio global. Esta última característica puede abrir las puertas a la gestión de arquitecturas basadas en componentes a partir de un modelo de computación por restricciones; se abre la perspectiva para que conceptos como el de repositorio global de restricciones (como es el caso del Global Store [Roy and Haridi, 2004] en Mozart), que se derivan del sustrato formal que lo soporta, podrán permitir la implementación real de arquitecturas con estas posibilidades.

7. AGRADECIMIENTOS

Se agradece a la Universidad del Valle, particularmente a la Escuela de Ingeniería de Sistemas y Computación, por su apoyo irrestricto al desarrollo de este trabajo. Es digno de mencionar el apoyo que el CENTRO DE INVESTIGACIONES Y DESARROLLO CIENTÍFICO de la Universidad Distrital Francisco José de Caldas brindó a través de la línea de apoyo a proyectos de investigación de maestría y doctorado.

8. REFERENCIAS

Allen, R. and Garlan, D. (1997). A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249.

Antonia Bertolino, P. I. and Muccini, H. (2003). Formal Methods in Testing Software Architectures. In Bernardo, M. and Inverardi, P., editors, *Formal Methods for Software Architectures*, Lecture Notes in Computer Science. Advanced Lectures, pages 122–147. Third International School on Formal Methods for the Design of Computer, Communication and Software Systems:Software Architectures, Springer.

Berry, G. and Boudol, G. (1992). The Chemical Abstract Machine. *Theoretical Computer Sci.*, 96:217–248.

Bryan, D. C. L. J. J. K. L. M. A. J. V. D. and Mann, W. (1995). Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355.

David Garlan, R. T. M. and Wile, D. (1997). ACME: An Architecture Description Interchange Language. CASCON'97.

Dean, T. R. and Cordy, J. R. (1995). A Sintactic Theory of Software Architecture. *IEEE Transactions on Software Engineering*, 21(4):302–313.

Diosa, H. A. (2005). Cálculo formal para especificar aspectos dinámicos de arquitecturas de software. Technical report, Escuela de Ingeniería de Sistemas y Computación. Universidad del Valle, Cali,Colombia.

Diosa, H. A. (2008). Especificación de un Modelo de Referencia Arquitectural de Software a Nivel de Configuración, Estructura y Comportamiento.

Jeff Magee, Naranker Dulay, S. E. and Kramer, J. (1995). Specifying Distributed Software Architectures. Fifth European Software Engineering Conference.

Medvidovic, N., Rosenblum, D. S., Redmiles, D. F., and Robbins, J. E. (2002). Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57.

Milner, R. (1999). *Communicating and Mobile Systems:the π-Calculus*. Cambridge University Press.

Murata, T. (1989). Petri Nets: Properties, Analysis and Applications. volume 77, pages 541–580. IEEE.

Niehren, J. and Muller, M. (1995). Constraints for Free in Concurrent Computation. Asian Computer Science Conference ACSC'95.

Object Management Group (2007). UML 2.1.2 Superstructure Specification.

Roy, P. V. and Haridi, S. (2004). *Concepts, Techniques and Models of Computer Programming*. MIT Press.

Sangiorgi, D. and Walker, D. (2003). *The π-calculus. A theory of Mobile Processes*. Cambridge University Press.

Saraswat, V. A. (1992). *Concurrent Constraint Programming*. The MIT Press.

Shaw, M. and Others (1995). Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335.

Smolka, G. (1994a). A Calculus for Higher-order Concurrent Constraint Programming with Deep Guards. Technical report, Bundesminister fi£¡r Forschung und Technologie.

Smolka, G. (1994b). A Foundation for Higher-order Concurrent Constraint Programming. Technical report, Bundesminister fi£¡r Forschung und Technologie.

Taylor, R. N., van der Hoek, A., and Dashofy, E. M. (2002). An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. *ACM ICSE'02*, 11(1):266–276.

W3C (2003). Scalable Vector Graphics (SVG) 1.1 Specification. W3C Recommendation. http://www.w3.org/TR/2003/REC-SVG11-20030114/.

W3C (2007). XML Metadata Interchange. XMI Specification. http://www.omg.org/spec/XMI/2.1.1/.