Especificación formal de arquitecturas de software basadas en componentes: chequeo de corrección con $cálculo - \rho_{arq}$

[Formal Specification of Component-Based Software Architectures: Correctness Checking with $\rho_{arq} - Calculus$]

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

En esta publicación se propone un método formal para el chequeo de corrección de arquitecturas de referencia basadas en componentes de software. Estos modelos son especificados por medio del $cálculo - \rho_{arq}$. Se hace uso de dos herramientas formales; la primera, el concepto de Sistema de Transición Rotulado (STR, en adelante) ampliado con el concepto de transición condicionada para variables lógicas cuya resolución podría obtenerse desde un *repositorio global* de restricciones. La segunda, la teoría de equivalencia de observación propuesta por Robin Milner y sus colaboradores en la Universidad de Cambridge.

Palabras claves: *arquitectura de referencia, modelo de referencia, $cálculo - \rho_{arq}$, método formal, componentes de software, Sistema de Transición Rotulado, chequeo de corrección, equivalencia de observación.*

Abstract

In this paper we describe a correctness checking formal method of reference architectures against reference models in a component-based software development perspective. These models are specified by means of $\rho_{arq} - calculus$. We use two formal tools; the first, the concept of Labelled Transition System (LTS) enhanced with the conditional-transition concept for logical variables whose resolution may be obtained from a global store of constraints. The second, the observation equivalence theory proposed by Robin Milner and his collaborators in Cambridge University.

Keywords: *Reference architecture, reference model, $\rho_{arq} - Calculus$, formal method, software components, labelled transition systems, correctness checking, observation equivalence*

[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.

[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.

1. INTRODUCCIÓN

Un modelo de referencia es una descripción de las posibles relaciones entre componentes de software para satisfacer los requerimientos de un dominio de problema específico. Una arquitectura de referencia es un subconjunto de servicios del modelo de referencia; dichos servicios pueden ser provistos por componentes de software conectados entre sí y que han sido seleccionados para cumplir requerimientos específicos del sistema; una arquitectura de software es una ejemplificación tecnológica concreta de una arquitectura de referencia [Len Bass and Kazman, 1998]. Es importante contar con técnicas que permitan verificar que una arquitectura de referencia es correcta frente a un modelo de referencia; en otras palabras, la ejemplificación obtenida debe ser altamente conforme con la abstracción de la cual se obtuvo .

Trabajos recientes en métodos formales para la especificación y verificación de ciertas propiedades en arquitecturas de software son variados; Kaveh y Emmerich [Nima Kaveh and Wolfgang Emmerich, 2003] proponen verificación de corrección y análisis de propiedades usando artefactos UML estereotipados como máquinas de estado. Claus Pahl desarrolla un cálculo de composición y sustitución de componentes, basado en el cálculo-π [Milner, 1999] con ligeras extensiones, para establecer un sistema de tipos de puerto y de canal que permiten determinar si hay posibles acoplamientos y sustitución de unos componentes por otros [Pahl, 2001]. Inverardi y Tivoli [Inverardi and Tivoli, 2003] proponen dos formas bisimilares de componer un sistema y luego analizan abrazos mortales y recuperación ante fallas junto con políticas de coordinación entre componentes (Orquestación) usando lógica lineal temporal. Bertolino, Inverardi y Muccini [Antonia Bertolino and Muccini, 2003] han propuesto un enfoque para derivar pruebas de conformidad en arquitecturas de software usando STRs y procesos de estado finito. Kramer, Magee y Uchitel [Jeff Kramer and Uchitel, 2003] [Jeff Magee / and Kramer, 1995] presentan un enfoque para el diseño y análisis de sistemas complejos a nivel arquitectural usando una herramienta de análisis de STR. Garlan y sus colaboradores [David Garlan / and Wile, 1997] formalizan un lenguaje para intercambio entre descripciones arquitecturales denominado Acme. Lamsweerde aporta un método para mapear requerimientos a especificaciones arquitecturales de software [van Lamsweerde, 2003]. Otros trabajos se concentran en métodos formales aplicados al análisis de dependencia en arquitecturas de software ([Judith A. Stafford and Caporuscio, 2003] , [Issarny and Zarras, 2003]) y evaluación de desempeño a nivel arquitectural [Simonneta Balsamo and Simeoni, 2003].

Un tópico de investigación activo en la ingeniería de software es el chequeo de corrección a nivel arquitectural [Simonneta Balsamo and Simeoni, 2003, Nima Kaveh and Wolfgang Emmerich, 2003, Antonia Bertolino and Muccini, 2003, Inverardi and Tivoli, 2003]. La verificación de corrección de una arquitectura de software frente a un modelo de referencia es un reto que hoy se enfrenta en el área de conocimiento de arquitecturas de software y es el asunto principal de este escrito. Inicialmente, se hace una revisión sucinta a la sintaxis y la semántica operacional del $cálculo_{\rho_{arq}}$ [Henry Diosa ; Juan F. Díaz and Mauricio Gaona, 2005]; se presentan estas secciones sólo para aquellos lectores que no conocen este formalismo. Luego se hace el empalme de este cálculo con los sistemas de transición rotulados y se propone la manera de aplicar los conceptos de congruencia estructural, bisimilaridad y la teoría de equivalencia de observación propuesta por Milner [Milner, 1999] [Sangiorgi and Walker, 2003] en el nivel de abstracción arquitectural. Finalmente, se precisa el enfoque propuesto para verificar corrección y se referencia lo efectuado para corroborar resultados de esta investigación.

2. FUNDAMENTOS TEÓRICOS

El $Cálculo - \rho_{arq}$ [Henry Diosa ; Juan F. Díaz and Mauricio Gaona, 2005] es un formalismo para especificar arquitecturas de software basadas en componentes, permite modelar aspectos estructurales y dinámicos con la posibilidad de controlar la ejecución arquitectural por medio de guardas que reducen a valores booleanos.

2.1 Sintaxis

Se presentan a continuación las entidades sintácticas del $cálculo - \rho_{arq}$ desde la perspectiva del modelamiento arquitectural de software; tales entidades están compendiadas en la Tabla 1.

- 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 un instrumento poderoso 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 (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 y que solamente interesa desde la perspectiva de ejecución exitosa o no de dicho cuerpo interno que puede ser activado por la obtención de 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 y 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.

[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**
$\mid$	$\perp$	**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: [Joachim Niehren and Martin Muller, 1995];[Smolka, 1994a] ; [Smolka, 1994b] ; [Milner, 1999] y este trabajo.

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

- 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 $cálculo - \rho$ [Joachim Niehren and Martin 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. SEMÁNTICA OPERACIONAL

El $cálculo - \rho_{arq}$ permite modelar el comportamiento de arquitecturas basadas en componentes a través de su semántica operacional. Se desarrolla en esta sección la aplicación de los conceptos de congruencia estructural y sistemas de transición rotulados para el modelado comportamental con propósitos de chequeo de corrección.

3.1 La congruencia estructural

Debido a la presencia de restricciones son aplicables las definiciones propuestas por Niehren y Muller [Joachim Niehren and Martin Muller, 1995].

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.

El axioma de *Replicación de observación* permite hacer vistas sucesivas de la 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 $E^\perp$

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:

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

y,

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

3.2 Reglas de reducción

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 innecesaria 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 $C_{\rho_{arq}}$ 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 los posibles dentro de un grupo, se debe usar un adecuado diseño de las guardas de cláusulas para evitar no-determinismos.

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

  Como se puede observar, una expresión **On Success Of** (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.

[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

[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.

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

Fuente [Joachim Niehren and Martin Muller, 1995] y este trabajo

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

$(A_{\rho_{arq}})$	$\phi \wedge x : \overline{y}/E \wedge x'\overline{z}/F \longrightarrow \phi \wedge x : \overline{y}/E \wedge [\overline{z}/\overline{y}]E^{(int)} \wedge F \quad si\ \phi \models_\Delta x = x', \mathcal{V}(\overline{z}) \cap \mathcal{BV}(E^{(int)}) = \emptyset$
$(C_{\rho_{arq}})$	$\phi_1 \wedge \phi_2 \longrightarrow \psi \quad si\ \phi_1 \dot{\wedge} \phi_2 \models\!\!\models_\Delta \psi$
$(Comb_{\rho_{arq}})$	$\phi \wedge if\ (C_1) \ldots (C_n)\ else\ F\ fi \longrightarrow \begin{cases} E_k, & si\ \phi \models_\Delta \psi_k \\ F, & si\ \phi \models_\Delta \neg\psi_k\ ; \forall k = 1, 2, \ldots, n \end{cases}$ $Con\ \ C_k ::= \exists \overline{x}(\psi_k\ Then\ E_k)\ ;\ k = 1, 2, \ldots, n$
$(Ejec_\tau)$	(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 [Joachim Niehren and Martin Muller, 1995] y este trabajo

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

3.3 Sistemas de Transición Rotulados

La Teoría de Equivalencia de Observación [Milner, 1999][Sangiorgi and Walker, 2003] usa el concepto de Sistemas de Transición Rotulados para representar la evolución comportamental de un sistema con procesos concurrentes interactuantes. Esta teoría es útil para demostrar la equivalencia comportamental entre sistemas conformados por autómatas que interactúan. Las raíces del $cálculo-\rho_{arq}$ en el $cálculo-\pi$ permiten reutilizar esta concepción para viabilizar la verificación de *corrección* entre una especificación de un conjunto de arquitecturas de software con características comunes de comportamiento (modelo de referencia) y una arquitectura de referencia particular.

El uso de variables lógicas en el $cálculo-\rho_{arq}$ [Henry Diosa ; Juan F. Díaz and Mauricio Gaona, 2005] permite expresar restricciones de arquitecturas basadas en componentes en tiempo de ejecución, esto permitiría en lenguajes de programación que incorporan el modelo de computación por restricciones utilizar el repositorio (*store*) de restricciones para controlar el flujo de ejecución arquitectural al resolver guardas, basados en restricciones, que controlan la ejecución de componentes de manera concurrente. Esta incorporación de restricciones lleva a proponer Sistemas de Transición Rotulados Condicionados que agregan el concepto de restricción o condición tal como se usa en las máquinas de estado propuestas por David Harel [Harel, 1987].

La actividad dentro de un sistema es descrita por la relación de reducción[Sangiorgi and Walker, 2003]; es decir, las reglas de la semántica operacional serían las que determinarían, al ser aplicadas, a qué nueva configuración puede evolucionar una arquitectura de software. Un sistema de transición rotulado parte de una unidad conceptual básica: **la acción**. Una *acción* permite fluir desde un estado a otro en un sistema de transición rotulado donde los rótulos corresponden a acciones.

DEFINICIÓN 1. ***Sistema de Transición Rotulado****[Milner, 1999]*

Un sistema de transición rotulado (STR) sobre un conjunto de acciones Act es un par $(\mathcal{Q}, \mathcal{T})$ consistente de :

- *Un conjunto de estados $\mathcal{Q}$*
- *Una relación ternaria $\mathcal{T} \subseteq (\mathcal{Q} \times Act \times \mathcal{Q})$ conocida como una relación de transición*

Si $(q, \alpha, q') \in \mathcal{T}$ se escribe $q \xrightarrow{\alpha} q'$ y se denomina a q la fuente y a q' el destino de la transición. Si $q \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} q_n$ se denomina a q_n un derivado de q bajo la serie de acciones $\alpha_1\alpha_2 \ldots \alpha_n$.

3.4 Sistemas de Transición Rotulados Condicionados (STRC)

DEFINICIÓN 2. *Un Sistema de Transición Rotulado Condicionado(STRC) es un Sistema de Transición Rotulado en el que $\alpha \in Act$ puede ser una restricción o guarda booleano y acciones diferentes a restricciones pueden estar condicionadas por restricciones o guardas booleanos.*

DEFINICIÓN 3. *$cálculo - \rho_{arq}$ y STRC's*

Un Sistema de Transición Rotulado Condicionado puede hacerse corresponder a expresiones del $cálculo - \rho_{arq}$ así:

1. $PREF_{recibir}$ $x :: \overline{y}/E \xrightarrow{x\overline{z}} [\overline{z}/\overline{y}]E$
2. $PREF_{enviar}$ $x\overline{y}/E \xrightarrow{x\overline{y}} E$
3. $ESCOG_{Comb_{\rho arq}}$ (a) $\phi \wedge if\ (C_1) \ldots (C_n)\ else\ F\ fi \xrightarrow{\psi_k} E_k$ (b) $\phi \wedge if\ (C_1) \ldots (C_n)\ else\ F\ fi \xrightarrow{\neg\psi_k} F \quad ; \forall k = 1, 2, \ldots, n$ $Con\ \ C_k ::= \exists \overline{x}(\psi_k\ Then\ E_k)\ ;\ k = 1, 2, \ldots, n$
4. Regla de Transición Observacional: (a) $[\ OSO(E)\ do\ F\ else\ G\ ] \wedge [\overline{z}/\overline{y}]E \xrightarrow{E^{\top}} F$ (b) $[\ OSO(E)\ do\ F\ else\ G\ ] \wedge [\overline{z}/\overline{y}]E \xrightarrow{E^{\perp}} G$
5. $\mathcal{A}_{\rho_{arq}}$: $x :: \overline{y}/E \wedge x\overline{z}/F \xrightarrow{\mathcal{A}_{\rho_{arq}}(x\overline{z})} [\overline{z}/\overline{y}]E \wedge F\ ; con\ \mathcal{V}(\overline{z}) \cap \mathcal{BV}(E) = \emptyset$

Tabla 4: Reglas de transición del $cálculo - \rho_{arq}$
Fuente[Este reporte]

1. *Una expresión arquitectural o un conjunto de expresiones arquitecturales que se reducen de manera concurrente son estados en un STRC. Por consiguiente, una configuración arquitectural de componentes puede modelarse con varias expresiones arquitecturales que evolucionan o reducen concurrentemente a otras configuraciones a través de acciones observables y no observables, que se definirán más adelante.*
2. *La semántica operacional en el cálculo* $- \rho_{arq}$ *determina las transiciones en un STRC. El repositorio (**store**) de restricciones podría resolver transiciones condicionadas. Las transiciones no condicionadas respetan el concepto de composición entre componentes [Henry Diosa ; Juan F. Díaz and Mauricio Gaona, 2005].*
3. *El estado inicial se hace corresponder con el término expresado por medio del cálculo* $- \rho_{arq}$ *para este fin.*
4. *La regla* $Ejec_{\tau}$ *o de éxito/fracaso observacional permitirá modelar las posibles transiciones en cualquiera de los dos casos (éxito o fracaso de ejecución de componentes) al aplicar* $\alpha -$ *conversión.*

3.5 Acciones en un STRC

Por consiguiente, se torna necesario definir las acciones concebibles desde el $cálculo - \rho_{arq}$[7].

[7]NOTA IMPORTANTE: En este tratamiento teórico, con el propósito de saturar menos al lector, se simplificará la expresión que representa puertos de requerimiento de servicio:

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

a:

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

presentada en la propuesta más actual y mejorada del cálculo formal.
Sin embargo, el concepto de alcance local del sitio l_E y el concepto de reemplazo en la parte interna de un componente, tal como se presentó en la última socialización del cálculo, se mantiene. El propósito es meramente de facilidad sintáctica en la escritura de expresiones sin afectar la semántica establecida originalmente.

DEFINICIÓN 4. ***Acciones***

Las acciones para un STRC podrán ser:

$$\alpha ::= x\overline{y} \mid \psi_k \mid \neg\psi_k \mid E^{\top} \mid E^{\perp} \mid \mathcal{A}_{\rho_{arq}(x\overline{y})}$$

En el mismo orden que se presenta en la definición 4, se describen a continuación las posibles acciones:

- Enviar $\overline{y}$ vía el nombre x.
- Desde un repositorio (*store*, en el inglés) de restricciones se deduce la condición o guarda booleano ψ_k.
- Desde un repositorio (*store*, en el inglés) de restricciones no se puede deducir la condición o guarda booleano ψ_k.
- La ejecución de un componente es exitosa, lo cual reduce según la regla $Ejec_{\tau}$ de éxito o fracaso observacional.

- La ejecución de un componente no es exitosa. Lo cual reduce según la regla $Ejec_{\tau}$ de éxito o fracaso observacional.

- Por último, puede existir una reacción (conforme al concepto en [Milner, 1999]) que se desprende de la configuración arquitectural de los componentes, donde dos de ellos pueden generar la aplicación de la regla de reducción $A_{\rho_{arq}}$ propia de la semántica operacional del $cálculo - \rho_{arq}$.

3.5.1 Reglas de transición en un STRC

Las acciones posibles para el $cálculo - \rho_{arq}$ se relacionan directamente con las reglas de reducción de la semántica operacional y son las que se expresan a través de las reglas de transición presentadas en la Tabla 4.

1. $PREF_{recibir}$. Esta regla de transición establece que un componente puede introducir paramétricamente $\overline{z}$ por medio del canal x y ejecutarse posteriormente.

2. $PREF_{enviar}$. Esta regla de transición establece que un componente puede enviar $\overline{y}$ a lo largo de un canal x y continuar su ejecución.

3. $ESCOG_{Comb_{\rho_{arq}}}$. La regla de reducción para la escogencia condicionada combinada interroga el repositorio (*store*) de restricciones para activar uno del conjunto de componentes condicionados (caso (a)) o ante una respuesta con valor lógico falso para todos los componentes condicionados se ejecuta uno alternativo F (caso (b)).

4. **La regla de transición observacional**. La regla de reducción asociada a la ejecución de un componente, permite obtener un valor observacional de *éxito*(caso (a)) o de *fracaso* (caso (b)) frente a la ejecución del mismo. Esta regla busca reducir a un término, que permita a través de los axiomas de congruencia estructural, resolver si se activan conectores de salida del componente en ejecución.

5. $\mathcal{A}_{\rho_{arq}(x\overline{z})}$. Esta regla respeta el concepto de reacción propuesto en el $cálculo - \pi$ [Milner, 1999][Sangiorgi and Walker, 2003] y reflejado en el $cálculo - \rho_{arq}$ con esta reducción. Nótese que no hay control por medio de una restricción de la aplicación; por consiguiente, permite que dos componentes interactúen en ausencia de restricciones si su configuración por defecto les permite reaccionar. Es de resaltar que se especifica paramétricamente cuál acción prefijo permite la interacción entre dos componentes.

   Se podría usar solo la acción prefijo, pues sería válida tanto para el componente que envía como para el que recibe. Esta simplificación podría permitir no tener en cuenta a $\mathcal{A}_{\rho_{arq}(x\overline{z})}$ como posible regla de transición; para indicar que ocurre una reacción por configuración por defecto se mantiene esta propuesta.

3.5.2 Algunos ejemplos

Ejemplo 1. *Una expresión arquitectural que especifica la configuración ilustrada en la Figura 1, suponiendo que se introduce al componente fuente $datos_{in}$ para reemplazar el parámetro de entrada i_F e iniciar la ejecución arquitectural, es:*

$S_2 = \{[datos_{in} \ / \ i_F \ ] \ F \ \wedge \ [\ OSO(F) \ do \ (\ F \ \wedge \ C_{FE} \ \wedge \ E \ ) \ else \ (\ F \ \wedge \ C_{FM} \ \wedge M \ ) \ ] \ \} \ \wedge \ \{ \ OSO(E) \ do \ (\ C_{ET} \ \wedge \ T \ ) \ else \ (\ C_{EM} \wedge M \ ) \ \} \wedge \{OSO(T) \ do \ S_2 \ = \ éxito \ else \ (\ C_{TM} \wedge \ M \ ) \ \}$

Donde cada componente tiene como especificación en el $cálculo - \rho_{arq}$:

$F \stackrel{def}{=} (p_F : z/zs_F) \ \wedge \ (p_{Fe} : w/ws_{Fe})$

$E \stackrel{def}{=} (p_E : x/xs_E) \ \wedge \ (r_E :: y/yl_E) \ \wedge \ (p_{Ee} : v/vs_{Ee}) \ \wedge \ (l_E :: i_E/E)$

$M \stackrel{def}{=} (r_M : y/yl_M) \ \wedge \ (l_M : i_M/M)$

$T \stackrel{def}{=} (r_T :: q/ql_T) \ \wedge \ (l_T : i_T/T) \ \wedge \ (p_{Te} : n/ns_{Te})$

*Una vista gráfica parcial del STRC que representa el comportamiento del sistema S_2 se presenta en la Figura 2. Como se puede observar en este STRC sólo se usan **la regla de transición observacional** y $\mathcal{A}_{\rho_{arq}(x\overline{z})}$ para ir de un estado a otro de acuerdo a lo propuesto en la `DEFINICIÓN 3`.*

*La ejecución de esta configuración arquitectural se inicia desde el componente **fuente**(F) que puede ser exitoso o no, se establecen así dos caminos posibles; cada vez que un componente se ejecuta se puede contar con **éxito** o **fracaso** observacional, tal como se ha propuesto desde la presentación del cálculo$-\rho_{arq}$ [Henry Diosa ; Juan F. Díaz and Mauricio Gaona, 2005]*

Cuando un componente no es exitoso en su ejecución, comunica al componente manejador de errores vía una acción $\mathcal{A}_{\rho_{arq}}$ para que este se encargue de efectuar el tratamiento adecuado del error de acuerdo a lo que entregue el componente no ejecutado exitosamente. Esto se puede observar por el camino de la derecha que considera el caso en que el componente F fracasa en su ejecución inicial.

Ejemplo 2. *La Figura 3 presenta una configuración arquitectural donde un repositorio de restricciones puede controlar la ejecución alternativa de componentes. Este STRC usa reglas de transición del tipo $ESCOG_{Comb_{\rho_{arq}}}$ para hacer fluir la ejecución arquitectural. En esta configuración considérese al componente M como el encargado de manejar los errores que comunica A en el proceso de ejecución y que provee a través del puerto P_{Ae} a la interfaz con M.*

La expresión en el cálculo$-\rho_{arq}$ que representa esta configuración, suponiendo que A es un componente fuente que reemplaza el parámetro de entrada $\overline{i}$ por el dato $\overline{d}$ y que cada componente alternativo se ejecuta exitosamente, es:

$S_{alt} \stackrel{def}{=} [\ \overline{d} \ / \overline{i}]A \wedge OSO(A) \ do \ \{ \ [\ if \ \exists X((X = p_S) \ then \ \{S^{\top} \wedge OSO(S) \ do \ [\ r_{AP_S} \wedge S \ ] \ else \ [Manejar \ error \ de \ S]\}) \ , \ \cdots \ , \ ((X = p_N) \ then \ \{ \ N^{\top} \ \wedge \ OSO(N) \ do \ [\ r_{AP_N} \ \wedge \ N \ ] \ else \ [Manejar \ error \ de \ N]\}) \ else \ Manejar \ error \ ] \ \wedge \ A\} \ else \ [\ r_M p_{Ae} \ \wedge \ M \ ] \ \wedge \ A \ \}$

Como se puede observar, el contexto de ejecución arquitectural determinará el valor de X (esto puede ser a través de un repositorio de restricciones que controla la ejecución arquitectural) y se corresponderá con una regla de transición del tipo $ESCOG_{Comb_{\rho_{arq}}}$ donde la restricción ψ_k, correspondiente en este caso a la condición de valor de X, se

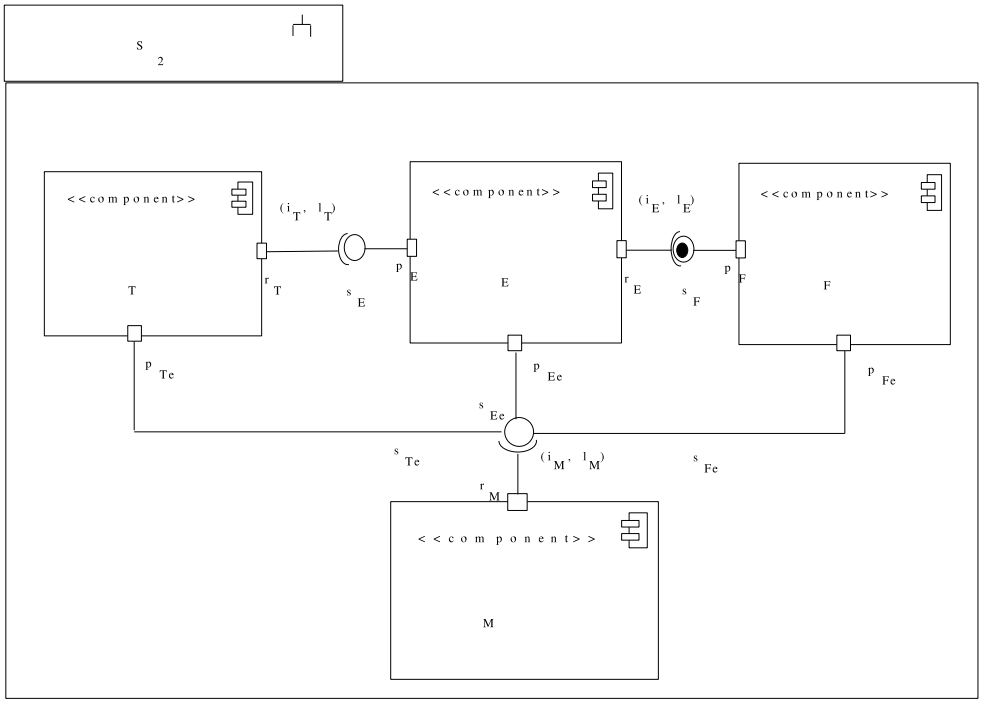

Figura 1: Ensamble complejo de componentes. Momento inicial: captura de datos exitosa

determina y genera diferentes flujos alternativos de ejecución.

La especificación formal de cada componente es:

$A \stackrel{def}{=} PROV_A(p_{Ae}, s_{Ae}) \ \wedge \ REQ_A(r_A, l_A, i_A) \ \equiv \ p_{Ae} : y/ys_{Ae} \ \wedge \ r_A :: y/yl_A \ \wedge \ l_A :: i_A/A$

$S \stackrel{def}{=} PROV_S(p_S, s) \ \equiv \ p_S : x/xs$

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

$M \stackrel{def}{=} REQ_M(r_M, l_M, i_M) \ \equiv \ r_M :: y/yl_M \wedge l_M :: i_M/M$

El STRC que se presenta en la Figura 4 refleja la posible actividad de S_{alt} a partir del cumplimiento de los guardas condicionales que se podrían resolver desde un store de restricciones. Como se puede observar cada ψ_k se representa expresando la condición o guarda booleano entre paréntesis angulares y el $\neg\psi_k$ está representado por el [else].

4. CONGRUENCIA ESTRUCTURAL Y BISIMILARIDAD

Aplicar la equivalencia de observación [Milner, 1999] o teoría de observabilidad [Sangiorgi and Walker, 2003] para verificar *corrección* entre una especificación en el *cálculo* $-\ \rho_{arq}$ y una implementación , en alguna tecnología que permita desarrollo basado en componentes, implica que se deben introducir los conceptos de **congruencia estructural** y bisimilaridad para este tipo de arquitecturas. Esta sección se ocupa de proponer el uso de STRC's para verificar *corrección*; el desarrollo de los conceptos aquí abordados siguen el patrón de desarrollo propuesto por Milner [Milner, 1999] y Walker/Sangiorgi [Sangiorgi and Walker, 2003] para el Cálculo de Sistemas de Comunicación con procesos concurrentes y el *cálculo* $-\ \pi$.

4.1 Contexto arquitectural

DEFINICIÓN 5. *Un contexto arquitectural $\mathcal{C}_{arq}$ es una expresión en el cálculo $-\ \rho_{arq}$ que contiene un espacio donde un componente puede incrustarse, representado por [ ]. Se puede expresar formalmente:*

$\mathcal{C}_{arq} ::= [\,] \mid \exists\overline{x}\,\mathcal{C}_{arq} \mid E \wedge \mathcal{C}_{arq} \mid \mathcal{C}_{arq} \wedge E \mid \phi \wedge if\ (C_1) \ldots (C_n)\ else\ F\ fi$;

$\{$Con $\mathtt{C_k} ::= \exists\overline{\mathtt{x}}(\psi_\mathtt{k}$ Then $\mathtt{E_k})$; $\mathtt{k} = 1, 2, \ldots, \mathtt{n}$ y algún
$\mathtt{C_i} ::= \exists\overline{\mathtt{x}}(\psi_\mathtt{i}$ Then $\mathcal{C}_{\mathtt{arq}})$ con $\mathtt{i} \in [1, \ldots, \mathtt{n}]$ $\}$

$\mathcal{C}_{arq}[F]$ significa que se ha introducido el componente F en el contexto $\mathcal{C}_{arq}$.

$\mathcal{C}_{arq} = [\,]$ es el contexto identidad, en este caso $\mathcal{C}_{arq}[F] = F$.

4.2 Congruencia de expresiones arquitecturales

DEFINICIÓN 6. *Si $\cong_{arq}$ es una relación de equivalencia sobre el conjunto $\mathcal{E}$ de expresiones que representan arquitec-*

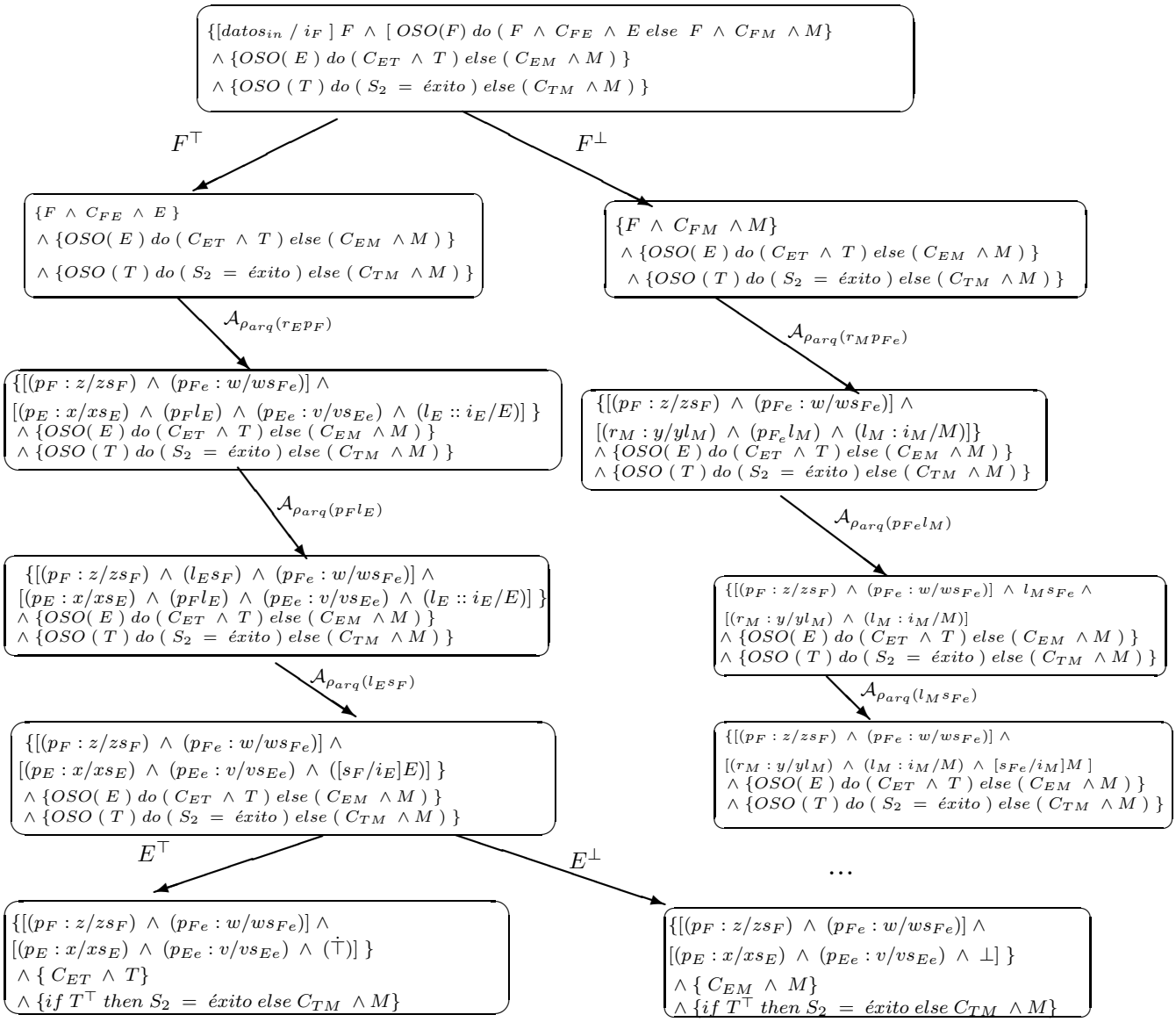

Figura 2: Vista parcial de un STRC

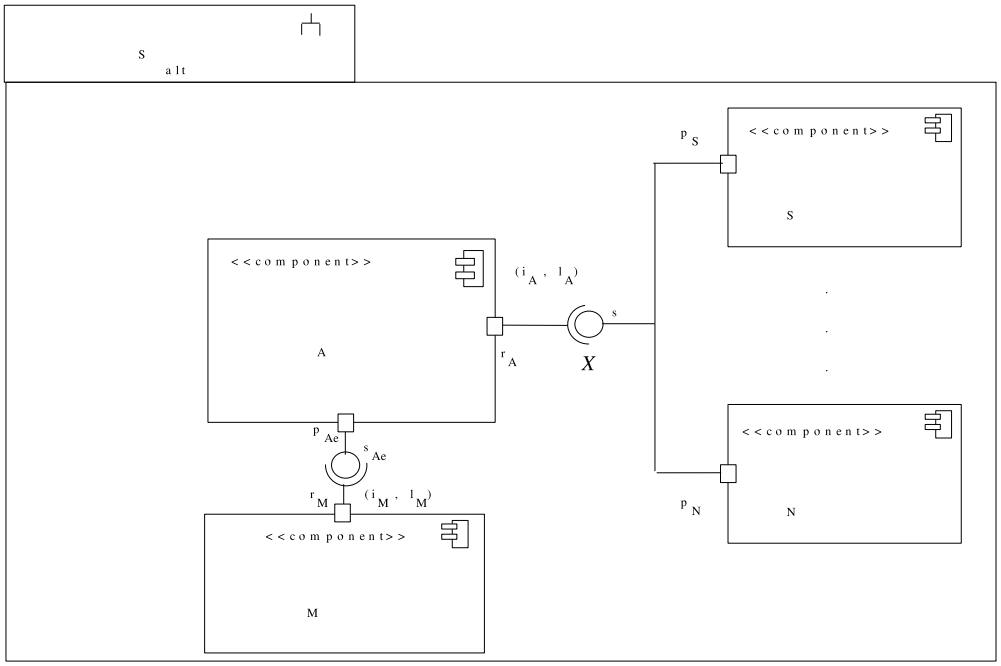

Figura 3: Configuración con componentes alternativos para proveer servicios

turas basadas en componentes, entonces $\cong_{arq}$ *es una congruencia arquitectural. Si* $E \cong_{arq} F$*:*

- $\exists \overline{x}\, E \cong_{arq} \exists \overline{x}\, F$
- $E \wedge G \cong_{arq} F \wedge G$
- $G \wedge E \cong_{arq} G \wedge F$
- $\phi \wedge if\ (C_1) \ldots (C_n)\ else\ M\ fi \cong_{arq} \phi \wedge if\ (L_1) \ldots (L_n)\ else\ M\ fi$

$\{$Con cada $\mathtt{C_k} ::= \exists \overline{\mathtt{x}}(\psi_\mathtt{k}$ Then $\mathtt{E_k})$ correspondiendo a un $\mathtt{L_k} ::= \exists \overline{\mathtt{x}}(\sigma_\mathtt{k}$ Then $\mathtt{F_k})$ donde $\mathtt{E_k} \cong_{\mathtt{arq}} \mathtt{F_k}$ y $\psi_\mathtt{k} \models|_\Delta \sigma_\mathtt{k}$; $\mathtt{k} = 1, 2, \ldots, \mathtt{n}\ \}$

Esta definición de congruencia de expresiones arquitecturales lo que hace es introducir componentes considerados equivalentes en los denominados contextos elementales [Milner, 1999, chap. 4]:

$$\exists \overline{x}\,[\,] \mid E \wedge [\,] \mid [\,] \wedge E \mid \phi \wedge (if\ \psi\ Then\ [\,]\ else\ M\ fi)$$

Como se puede observar, el contexto elemental $\phi \wedge (if\ \psi\ Then\ [\,]\ else\ M\ fi)$ se hace corresponder con la última expresión arquitectural de arriba respecto a que dicha expresión es una combinación para escogencia condicionada de varios contextos elementales de este tipo. Es importante recalcar aquí, si en el conjunto de posibles condiciones ψ_k existen dos o más de ellas equivalentes lógicamente, se presentaría **no-determinismo** en la ejecución de componentes alternativos.

4.3 Congruencia estructural de expresiones arquitecturales

Definición 7. *La congruencia estructural* $\equiv$*, es la congruencia de expresiones arquitecturales sobre* $\mathcal{E}$ *que está regida por:*

1. *La introducción de nombres a través de la interfaz de requerimiento de servicios de un componente. Desde una perspectiva de comportamiento interno del componente se produce una alfa-conversión o reemplazo de nombres ligados localmente por los nombres entrantes.*
2. *El reordenamiento de cláusulas en una expresión de escogencia condicionada de componentes.*
3. $E \wedge \top \equiv E$
4. $\exists \overline{x}\, \top \equiv \top$
5. $E \wedge G \equiv G \wedge E$
6. $E \wedge (F \wedge G) \equiv (E \wedge F) \wedge G$
7. $\exists \overline{xy}\, E \equiv \exists \overline{yx}\, E$
8. $\exists\, x\, (E \wedge F) \equiv (E \wedge \exists\, x\, F)$ *si* $x \notin \mathcal{FV}(E)$

4.4 Simulación fuerte arquitectural

Dados los estados $e \in \mathcal{E}$ y $f \in \mathcal{F}$ en las configuraciones arquitecturales E y F respectivamente.

Definición 8. *Simulación fuerte entre estados* f *simula fuertemente a* e *si existiendo los STRC's* $\{(\mathcal{E}, \mathcal{T}), (\mathcal{F}, \mathcal{T}')\}$ *y una relación binaria* S *aplicada entre ellos, cumplen que para cada relación* $(e\ S\ f)$ *establecida, si* $(e, \alpha_1, e') \in T$ *existe* $(f, \alpha_1, f') \in T'$ *y* $e'\ S\ f'$*.*

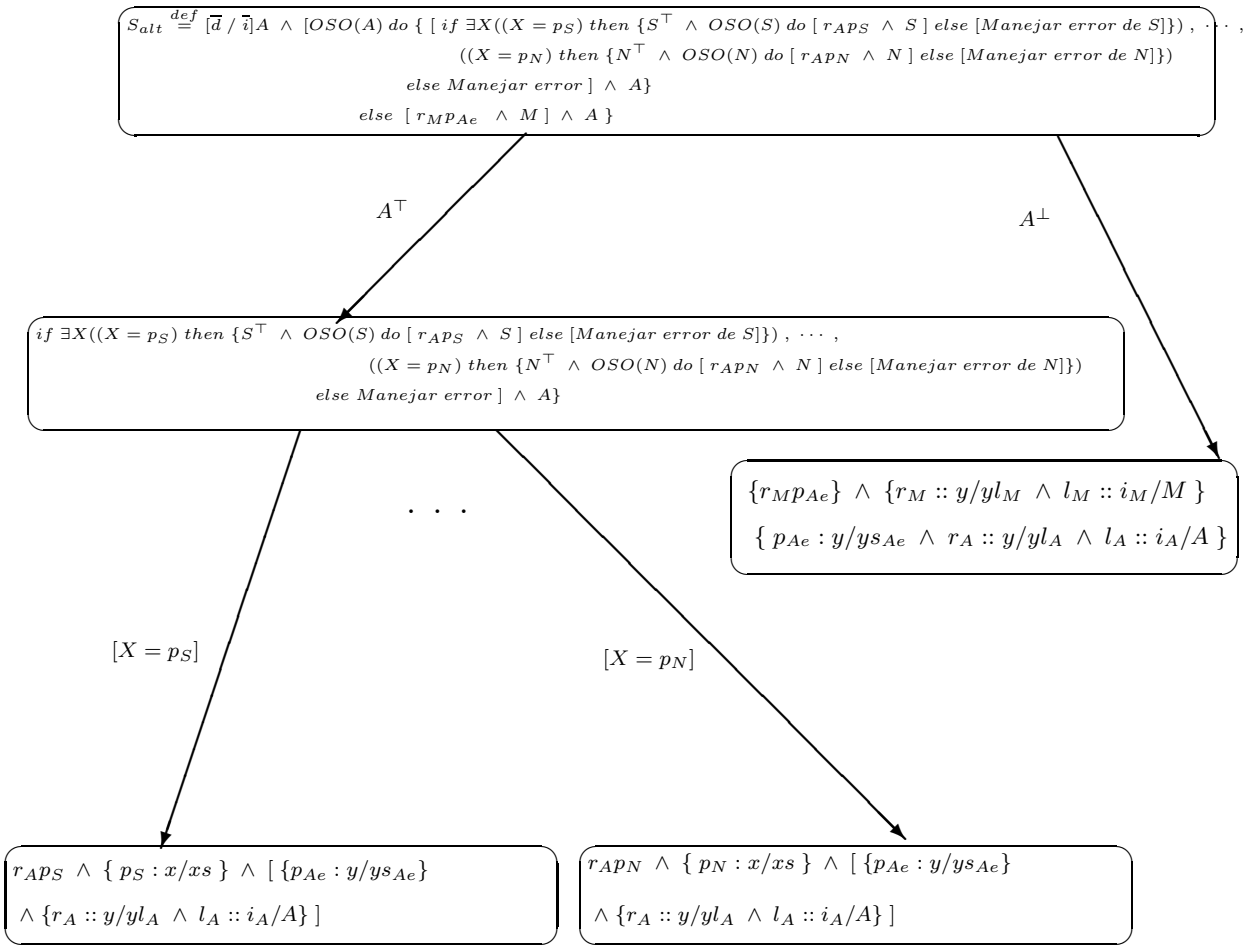

Figura 4: Vista parcial de un STRC con escogencia alternativa de componentes que se ejecutan exitosamente

DEFINICIÓN 9. *Simulación fuerte arquitectural*
Si para todo $e \in \mathcal{E}$ *existe* $f \in \mathcal{F}$ *que cumple la definición [8]* ***la configuración arquitectural*** F ***simula fuertemente a la configuración arquitectural*** E.

4.5 Bisimilaridad fuerte arquitectural

DEFINICIÓN 10. *Bisimulación o equivalencia fuerte entre estados*
Una relación binaria S *como la presentada en [8] es una* ***bisimulación fuerte*** *entre* $(\mathcal{E}, \mathcal{T})$ *y* $(\mathcal{F}, \mathcal{T}')$ *si tanto* S *y su inversa son simulaciones fuertes. Se dice entonces que los estados* e *y* f *son* ***fuertemente bisimilares*** *o* ***fuertemente equivalentes****, escrito como* $e \sim f$.

DEFINICIÓN 11. *Bisimulación o equivalencia fuerte arquitectural*
Si $\forall e \in \mathcal{E}\ \exists f \in \mathcal{F}$ *y* $\forall f \in \mathcal{F}\ \exists e \in \mathcal{E}$ *que cumplen la definición [10],* ***la configuración arquitectural*** F ***es fuertemente bisimilar o fuertemente equivalente a la configuración arquitectural*** E*, escrito como* $E \sim F$.

Es de anotar que la fuerte bisimulación es una relación de equivalencia y que si $E \equiv F$ entonces $E \sim F$.[Milner, 1999, sección 5.2]

5. LA TEORÍA DE EQUIVALENCIA DE OBSERVACIÓN

5.1 El concepto de observación

Debido al enfoque observacional del desarrollo basado en componentes, esta teoría se aplica partiendo de la premisa que sistemas con diferente estructura interna y por lo tanto diferente comportamiento interno pueden considerarse equivalentes.[Milner, 1999][Sangiorgi and Walker, 2003].

Por consiguiente, se denominará $\lambda \in Act$ a toda acción observable como comportamiento externo de una arquitectura. Un sistema configurado por defecto como $\exists\, x\ (\ x\overline{b}\ /\ E \wedge x :: \overline{y}\ /\ F)$ podría reducir de manera no observable externamente a: $\exists\, x\ (\ E \wedge [\overline{b}/\overline{y}]F)$; por consiguiente, aunque E y F se observan mutuamente, para un componente externo a su configuración por defecto no habría observación sino una reacción interna.

5.2 Experimento

Un experimento exp se considera una secuencia $exp = \lambda_1, \ldots, \lambda_n$ de acciones observables sobre una arquitectura basada en componentes y se representa cada acción observable como $\xrightarrow{\lambda_i}$. Los niveles de observabilidad pueden ser controlados desde la definición del conjunto de acciones que permiten transiciones en STRC (esto se aclarará más adelante).

5.3 Relaciones experimento

Las relaciones $\Longrightarrow$ y $\overset{s}{\Longrightarrow}$, con $s = \lambda_1, \ldots, \lambda n$ perteneciendo al conjunto cerradura Act^*, se definen de la siguiente manera:

1. $E \Longrightarrow F$ significa que hay una secuencia de cero o más reacciones internas $E \longrightarrow \ldots \longrightarrow F$.
2. Dado s, tal como se define arriba, entonces $E \overset{s}{\Longrightarrow} F$ significa que $E \Longrightarrow \xrightarrow{\lambda_1} E_1 \ldots \Longrightarrow \xrightarrow{\lambda_n} E_n \Longrightarrow F$. Formalmente $s \overset{def}{=} \Longrightarrow \xrightarrow{\lambda_1} \Longrightarrow \ldots \Longrightarrow \xrightarrow{\lambda_n} \Longrightarrow$

5.3.1 Semántica de las relaciones

1. $\overset{\lambda}{\Longrightarrow}$ es la observación de $\xrightarrow{\lambda}$ acompañada antes y después con algún número de reacciones internas.
2. $\overset{\tau}{\Longrightarrow}$ significa al menos una reacción interna.
3. $\Longrightarrow$ significa cero o más reacciones internas.
4. $\overset{exp}{\Longrightarrow}$ es el desarrollo de un experimento e, tal como se ha definido arriba, es decir, como una secuencia de observaciones $\lambda_1, \ldots, \lambda_n$, pueden existir reacciones internas entrelazadas con las observaciones. Ahora, $\overset{\epsilon}{\Longrightarrow}$ corresponde al experimento vacío, que se puede expresar formalmente como: $\overset{\epsilon}{\Longrightarrow} = \Longrightarrow$. Lo anterior quiere expresar que se pueden dar reacciones internas aún sin observar algo cuando $exp = \epsilon$.

5.4 Simulación débil arquitectural

Dados los estados $e \in \mathcal{E}$ y $f \in \mathcal{F}$ en las configuraciones arquitecturales E y F respectivamente.

DEFINICIÓN 12. *Simulación débil entre estados*
f simula débilmente a e si existiendo los STRC's $\{(\mathcal{E}, \mathcal{T}), (\mathcal{F}, \mathcal{T}')\}$ *y una relación binaria* ($e\ S\ f$)*, se cumple:*

Si $e \overset{exp}{\Longrightarrow} e' \in T$ *existe* $(f \overset{exp}{\Longrightarrow} f') \in T'$ *y* $e'\ S\ f'$.

DEFINICIÓN 13. *Simulación débil arquitectural*
Si para todo $e \in \mathcal{E}$ *existe* $f \in \mathcal{F}$ *que cumple la* `DEFINICIÓN 12` ***la configuración arquitectural*** F ***simula débilmente a la configuración arquitectural*** E.

5.5 Bisimulación o bisimilaridad débil arquitectural

DEFINICIÓN 14. *Bisimulación débil entre estados*
Una relación binaria S *como la presentada en la* `DEFINICIÓN 12` *sobre los STRCs* $\{(\mathcal{E}, \mathcal{T}), (\mathcal{F}, \mathcal{T}')\}$ *se denomina* ***bisimulación débil*** *si tanto* S *como* S^{-1} *son simulaciones débiles.*

DEFINICIÓN 15. *Bisimulación o bisimilaridad débil arquitectural*
Si $\forall e \in \mathcal{E}\ \exists f \in \mathcal{F}$ *y* $\forall f \in \mathcal{F}\ \exists e \in \mathcal{E}$ *que cumplen la* `DEFINICIÓN 14`*, las configuraciones arquitecturales cuyos STRCs se corresponden con* $\{(\mathcal{E}, \mathcal{T}), (\mathcal{F}, \mathcal{T}')\}$ *son* ***débilmente bisimilares, débilmente equivalentes o con observabilidad equivalente****, lo cual se expresará como* $E \approx F$ *si* E *y* F *son las configuraciones arquitecturales correspondientes a los STRCs mencionados.*

6. VERIFICACIÓN DE CORRECCIÓN A NIVEL ARQUITECTURAL

Antes de empalmar la teoría desarrollada aquí con la verificación de corrección a nivel arquitectural, se recordarán algunos conceptos de arquitecturas de software[Len Bass and Kazman, 1998,].

DEFINICIÓN 16. *Modelo de referencia arquitectural de software basado en componentes.*
Es una descomposición estándar de un problema conocido en partes, estereotipadas como componentes, que trabajan cooperativamente para solucionarlo. Esos modelos surgen

del trabajo de campo que genera experiencia. Los modelos de referencia caracterizan dominios maduros y son obtenidos generalmente por medio del análisis de dominio u otra actividad en equipo o grupal.

Un ejemplo de modelo de referencia puede referirse al que se podría obtener en el dominio de problema de los ERPs universitarios. Este modelo podría expresar las subarquitecturas (posibles módulos o subsistemas) con sus componentes internos y su configuración de conexión como unidades computacionales que interactúan para lograr los objetivos propios de dichos sistemas. El modelo de referencia no está amarrado ni describe tecnologías de implementación.

Definición 17. *Arquitectura de referencia basada en componentes*

Es un modelo de referencia traducido a componentes de software(que implementan cooperativamente la funcionalidad definida en el modelo de referencia) y los flujos de servicios entre los componentes. Así como el modelo de referencia divide la funcionalidad, una ***arquitectura de referencia*** *es la traducción de esta funcionalidad a la descomposición de un sistema en componentes; este mapeo no es necesariamente* ***uno-a-uno****. Se puede decir que la arquitectura de referencia es una instancia arquitectural del modelo de referencia.*

Como tales, las arquitecturas de referencia representan una división de funcionalidad de un sistema de software que representa para el caso de servicios web, el flujo de servicios entre los componentes que la constituyen, en un modo de interacción basado en documentos o en un modo interacción de invocación a procesos [**?**, capítulo 6].

Definición 18. *Arquitectura de software*

Es la implementación de una arquitectura de referencia con una tecnología o combinación de tecnologías específicas. Por ejemplo, se podrá tener un modelo de referencia de un ERP universitario con arquitecturas de referencia implementadas sobre tecnologías de servicios web como las propuestas por SUN, Microsoft, Oracle o sobre un ORB conforme CORBA u otras plataformas específicas que soporten desarrollo basado en componentes.

Como se puede observar, el modelo de referencia y la arquitectura de referencia se generan en tiempo de diseño mientras la arquitectura de software se da en tiempo de implementación y ejecución.

Dado que una **arquitectura de referencia** es una instancia arquitectural de un **modelo de referencia** y la primera antecede la implementación de la **arquitectura de software** correspondiente; es de interés establecer si la **arquitectura de referencia** es **correcta** frente a su especificación de más alto nivel que es el **modelo de referencia**.

6.1 Corrección de una arquitectura de referencia frente a un modelo de referencia

Definición 19. *Corrección fuerte arquitectural*

Una arquitectura de referencia es fuertemente correcta respecto a un modelo de referencia arquitectural, si la especificación de las configuraciones arquitecturales expresadas en el $cálculo - \rho_{arq}$ de los dos modelos son fuertemente bisimilares arquitecturalmente.

Definición 20. *Corrección débil*

Una arquitectura de referencia es débilmente correcta respecto a un modelo de referencia arquitectural, si la especificación de las configuraciones arquitecturales expresadas en el $cálculo - \rho_{arq}$ de los dos modelos son débilmente bisimilares arquitecturalmente. Es decir, los comportamientos observables son equivalentes.

7. CORROBORACIÓN DE RESULTADOS

La teoría aquí expuesta se ha corroborado verificando la corrección de una arquitectura de software de un sistema simulado de ascensores frente a un modelo de referencia de sistemas de este tipo. Para un mayor detalle consultar [Diosa, 2008, Cap. 6].

La arquitectura de software fue ejemplificada con componentes de software construídos en el lenguaje de programación Mozart [Roy and Haridi, 2004]. Este hecho abre la posibilida de desarrollar con este lenguaje productos de software dentro del paradigma de desarrollo basado en componentes, en un modelo de distribución abierto que puede estar acompañado metodológicamente de una serie de actividades de diseño, construcción y verificacón arquitectural como las mostradas en la Figura 5.

8. CONCLUSIONES

1. Se ha logrado expresar configuraciones arquitecturales basadas en componentes de software que son traducibles a sistemas de transición rotulados condicionados. Estos sistemas de transición rotulados permiten modelar el comportamiento de dichas configuraciones arquitecturales con componentes de software a los que se pueden aplicar restricciones en tiempo de ejecución.

2. Ajustar a nuestros propósitos la teoría de equivalencia de observación propuesta por Milner y otros en el $\pi - calculus$ [Milner, 1999] [Sangiorgi and Walker, 2003] posibilitó la verificación de corrección a nivel arquitectural como este trabajo de investigación lo ha ilustrado.

3. Se ha determinado que esta técnica de chequeo permite diferenciar niveles de observabilidad en el comportamiento de una arquitectura basada en componentes de software. Estos niveles de observabilidad pueden indicar corrección en los niveles más gruesos o menos granulares y, lo contrario, en niveles más granulares o de observabilidad más detallada.

9. TRABAJO FUTURO

1. Asociar herramientas de chequeo automático de modelos (como *Labelled Transition System Analyzer* [Department of Computing, Imperial College London, 2007]) da soporte al análisis automatizado de aspectos dinámicos de arquitecturas de software.

2. Se ha establecido una primera relación de expresiones arquitecturales con modelos basados en componentes conforme a UML 2.x [Object Management Group, 2007]; se ha asociado una representación gráfica del flujo de la ejecución arquitectural que es similar al flujo comportamental representado por medio de redes de Petri

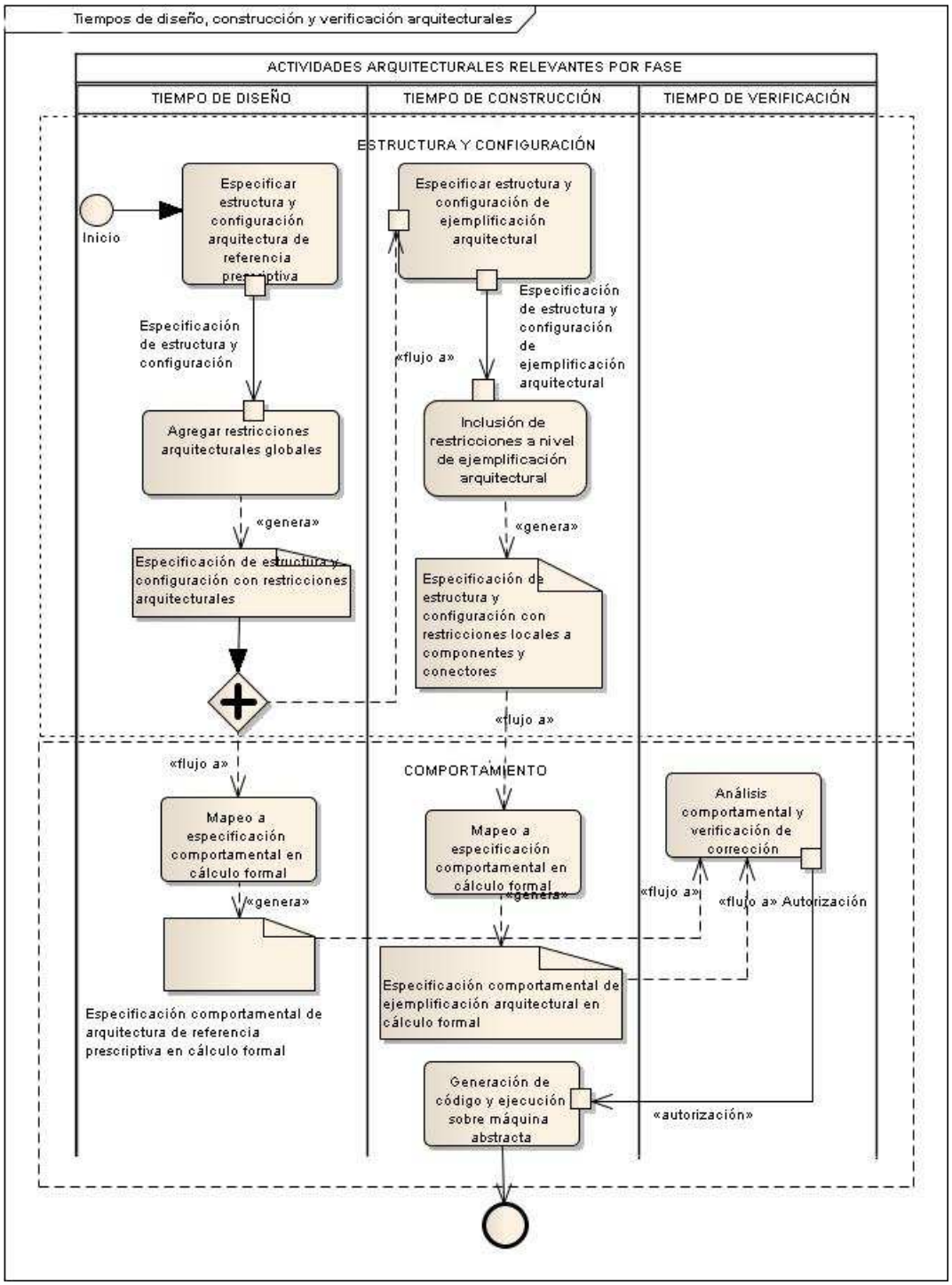

Figura 5: Tiempos de diseño, construcción y verificación arquitectural

[Murata, 1989][Girault and Rüdiger, 2003]. Esta afortunada coincidencia posibilita pensar en la traducción de expresiones arquitecturales en el $cálculo - \rho_{arq}$ a notación de redes de Petri con el objetivo de verificar por esta vía otras propiedades dinámicas. Esta alternativa es explorada actualmente.

3. El $cálculo - \rho_{arq}$ no tiene incorporada la dimensión temporal que permitiría el chequeo de otras propiedades como vivacidad y detección de abrazos mortales. Se trabajará en ampliar la sintaxis y semántica del formalismo para posibilitar el modelado a nivel arquitecural incluyendo tiempo.

10. 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.

11. REFERENCIAS

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.

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

Department of Computing, Imperial College London (2007). Labelled Transition System Analyzer. http://www.doc.ic.ac.uk/ltsa/eclipse/.

Diosa, H. A. (2008). *Especificación de un Modelo de Referencia Arquitectural de Software a Nivel de Configuración, Estructura y Comportamiento*. PhD thesis, Universidad del Valle. Doctorado en Ingeniería con Énfasis en Ciencias de la Computación.

Girault, C. and Rüdiger (2003). *Petri Nets for Systems Engineering.A Guide to Modeling, Verification, and Applications*. Springer.

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming 8*, 21(8):231–274.

Henry Diosa ; Juan F. Díaz and Mauricio Gaona (2005). ρ_{arq} : *Cálculo* para el Modelamiento Formal de Arquitecturas de Software Basadas en Componentes y con Restricciones en Tiempo de Ejecución. Conferencia Latinoamericana de Informática 2005.

Inverardi, P. and Tivoli, M. (2003). Software Architecture for Correct Components Assembly. In Bernardo, M. and Inverardi, P., editors, *Formal Methods for Software Architectures*, Lecture Notes in Computer Science. Advanced Lectures, pages 92–121. Third International School on Formal Methods for the Design of Computer, Communication and Software Systems:Software Architectures, Springer.

Issarny, V. and Zarras, A. (2003). Software Architecture and Dependability. In Bernardo, M. and Inverardi, P., editors, *Formal Methods for Software Architectures*, Lecture Notes in Computer Science. Advanced Lectures, pages 259–285. Third International School on Formal Methods for the Design of Computer, Communication and Software Systems:Software Architectures, Springer.

Jeff Kramer J. M. and Uchitel, S. (2003). Software Architecture Modeling '&' Analysis: A Rigorous Approach. In Bernardo, M. and Inverardi, P., editors, *Formal Methods for Software Architectures*, Lecture Notes in Computer Science. Advanced Lectures, pages 44–51. Third International School on Formal Methods for the Design of Computer, Communication and Software Systems:Software Architectures, Springer.

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

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

Judith A. Stafford A. L. W. and Caporuscio, M. (2003). The Application of Dependence Analysis to Software Architecture Descriptions. In Bernardo, M. and Inverardi, P., editors, *Formal Methods for Software Architectures*, Lecture Notes in Computer Science. Advanced Lectures, pages 52–62. Third International School on Formal Methods for the Design of Computer, Communication and Software Systems:Software Architectures, Springer.

Len Bass P. C. and Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley.

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.

Nima Kaveh and Wolfgang Emmerich (2003). Validating Distributed Object and Component Designs. In Bernardo, M. and Inverardi, P., editors, *Formal Methods for Software Architectures*, Lecture Notes in Computer Science. Advanced Lectures, pages 63–91. Third International School on Formal Methods for the Design of Computer, Communication and Software Systems:Software Architectures, Springer.

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

Pahl, C. (2001). A Pi-Calculus based Framework for the Composition and Replacement of Components. OOPSLA'2001-Workshop on Specification and Verification of Component-Based Systems.

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.

Simonneta Balsamo M. B. and Simeoni, M. (2003). Performance Evaluation at the Software Architecture Level. In Bernardo, M. and Inverardi, P., editors, *Formal Methods for Software Architectures*, Lecture Notes in Computer Science. Advanced Lectures, pages 207–258. Third International School on Formal Methods for the Design of Computer, Communication and Software Systems:Software Architectures, Springer.

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.

van Lamsweerde, A. (2003). From System Goals to Software Architecture. In Bernardo, M. and Inverardi, P., editors, *Formal Methods for Software Architectures*, Lecture Notes in Computer Science. Advanced Lectures, pages 25–43. Third International School on Formal Methods for the Design of Computer, Communication and Software Systems:Software Architectures, Springer.