Especificación del comportamiento de líneas de productos mediante modelos de funcionalidades y máquinas de estados con variabilidades

Luna, Carlos
González, Ariel

Documento de Trabajo No. 6
Facultad de Ingeniería
Universidad ORT Uruguay
Diciembre, 2007
ISSN 1688-3217

Documento de Trabajo

UNIVERSIDAD ORT
Uruguay

ISSN 1688-3217

Especificación del comportamiento de líneas de productos mediante modelos de funcionalidades y máquinas de estados con variabilidades

Carlos Luna
Ariel González

Documento de Trabajo Nº 6
Facultad de Ingeniería
Universidad ORT Uruguay

Diciembre, 2007

Especificación del Comportamiento de Líneas de Productos mediante Modelos de Funcionalidades y Máquinas de Estados con Variabilidades [§]

Carlos Luna
Universidad ORT, Facultad de Ingeniería,
Montevideo, Uruguay, CP 11100
luna@ort.edu.uy

y

Ariel Gonzalez
Universidad Nac. de Río Cuarto, Dep. de Computación,
Río Cuarto, Argentina, CP 5800
gonzalezg@exa.unrc.edu.ar

Resumen

El estudio de la variabilidad en el desarrollo del software viene creciendo de manera significativa en los últimos años. Las áreas de investigación abarcan desde la personalización del software a las líneas de productos. La manera más común de representar la variabilidad en líneas de productos de software es mediante modelos de funcionalidades o características. Sin embargo, la relación entre estos modelos y los modelos de diseño, generalmente basados en UML, no es sencilla. Este trabajo presenta una extensión de las máquinas de estados de UML con el uso de variabilidades en sus componentes esenciales para especificar el comportamiento de líneas de productos. Esto se logra junto con el uso de modelos de funcionalidades para describir los componentes comunes y los variantes, de forma tal que a partir de distintas configuraciones de un modelo se pueden generar máquinas de estados concretas para los diferentes productos de una línea.

Palabras clave: Máquinas de Estados, Modelos de Funcionalidades, Líneas de Productos (Familias de Sistemas), UML.

[§] Este trabajo es parcialmente financiado por el proyecto PDT Uruguay 54/106.

1 Introducción

La envergadura de los sistemas de software y hardware a nivel industrial ha generado la necesidad de basar los desarrollos en el modelado previo de los sistemas. Este modelado permite, entre otras cosas, verificar los sistemas antes de su construcción y guiar el desarrollo de éstos mediante técnicas tales como la generación automática de código.

El desarrollo dirigido por modelos (Model-Driven Development, MDD) [MCF03, Sel03] es una metodología de la ingeniería de software que idealmente eleva el desarrollo de software a un mayor nivel de abstracción. Los lenguajes de modelado y las herramientas de transformación de modelos y de generación de código permiten disminuir la brecha existente entre el dominio de los problemas y el de las soluciones, proporcionando un enfoque con el potencial de incrementar considerablemente tanto la productividad del desarrollo industrial de software como la calidad del resultado de estos desarrollos.

Las notaciones visuales o gráficas ganan día a día significación para la comunicación entre personas, así como también para la interacción hombre-computadora. En particular, para el desarrollo de software basado en modelos, el lenguaje UML (Unified Modeling Language) [OMG04] provee una notación gráfica y se ha convertido en el estándar para el modelado de diferentes aspectos de sistemas de software tanto en el ambiente académico como en desarrollos industriales.

UML sigue el paradigma de orientación a objetos y permite la descripción de aspectos tanto estáticos como dinámicos de sistemas de software. Más que un lenguaje es un conjunto de lenguajes, en su mayoría notaciones gráficas, soportado además por un número importante de herramientas desarrolladas por varias compañías comerciales. Actualmente está disponible la versión 2.0, especialmente adaptada para el soporte de MDD.

La complejidad de los emprendimientos en los dominios de uso intensivo de software demanda una disciplina en el nivel adecuado de abstracción. La administración de configuraciones de productos que varían en aspectos más o menos periféricos ha dado lugar al concepto de Línea de Productos. Una Línea de Productos (Products Line, PL), también llamada Familia de Sistemas es un conjunto de sistemas que comparten funcionalidades y satisfacen, en general, las necesidades de un segmento particular del mercado [CN02, Gom04]. Aunque dependiendo de la literatura, las nociones de PLs y, Familias de Sistemas o Productos pueden no ser totalmente equivalentes, usaremos estos términos de manera indistinta a lo largo de este trabajo, según la definición y referencias previas.

Desarrollar una familia de sistemas de software en lugar de un conjunto de sistemas por separado tiene importantes ventajas. Mediante la creación de un núcleo que englobe las funcionalidades comunes se facilita la construcción de los productos deseados. Los diferentes productos de la línea se obtienen incorporando funcionalidades distintivas (variabilidades) al núcleo. Por ejemplo, actualmente observamos en el mercado un número importante de distintos tipos de teléfonos móviles que comparten un núcleo de funcionalidades básicas y difieren en otras más especificas: disponibilidad de cámara digital, acceso a internet, capacidad de reproducir sonido mp3, entre otras.

La disciplina de PLs es una clara exponente del principio de reuso. Es posible aplicar estos conceptos desde los requerimientos hasta el código, con las ventajas que esto implica. El desarrollo y modelado de las partes comunes y variantes deben ser soportados por métodos y notaciones adecuados.

Los medios que UML pone a disposición para la especificación del comportamiento de sistemas son: acciones, actividades, máquinas de estados, interacciones y casos de uso. En particular, las máquinas de estados y las interacciones están especialmente confeccionadas para la fase de diseño de software. Las Máquinas de Estados (StateCharts, SCs), introducidas originalmente por Harel [Har87], son utilizadas para especificar el comportamiento de las instancias de una clase (intra-component behaviour) y constituyen por tanto un mecanismo adecuado para detallar el comportamiento de ciertos problemas mediante una representación gráfica. Estos diagramas son compactos, expresivos y proveen la capacidad de modelar no sólo sistemas simples sino también complejos sistemas reactivos. La representación gráfica de los problemas produce especificaciones comprensibles, fáciles de mantener y de extender. En la versión 2.0 de UML, los SCs no ofrecen operadores y/o sublenguajes para la especificación de familias de sistemas.

En este trabajo proponemos una extensión de los SCs para especificar PLs. Utilizamos Modelos de Funcionalidades (Features Models, FMs) para describir las funcionalidades (también llamadas características) comunes y las variantes de una familia [CE00], e incorporamos variabilidades en los componentes esenciales de los SCs para que a partir de distintas configuraciones de un FM se puedan generar SCs concretos para los diferentes productos de una PL.

Si bien existen extensiones de algunos modelos de UML para la especificación de variabilidades, por ejemplo [CGW06, Gac01, GS02, Gom04, LLR06, GBS01, ZHJ04], no se habían definido al inicio de esta investigación, al menos para nuestro conocimiento, mecanismos para la especificación de variabilidades en

los SCs, que juegan un rol central en la fase de diseño de software. Dada la relevancia de UML, confiamos en que los resultados de este trabajo puedan ser aplicados en problemas reales de la industria del software.

La organización del resto de este artículo es como sigue. La sección 2 introduce los SCs y la sección 3 los FMs. En la sección 4 presentamos una extensión de los SCs con elementos variables, los cuales junto con el uso de los FMs permiten especificar PLs. Luego, en la sección 5 detallamos el mecanismo para la obtención de productos de una PL a partir de distintas configuraciones de un FM. En la sección 6 analizamos trabajos relacionados y finalmente, incluimos las conclusiones y los trabajos futuros en la sección 7. A lo largo del artículo desarrollamos parcialmente un caso de estudio basado en tecnología de telefonía móvil.

Una versión preliminar de este trabajo fue presentada en CLEI'2007 [SLG07].

2 Máquinas de Estados

Las *Máquinas de Estados* (Statecharts, SCs) constituyen una notación para describir el comportamiento de sistemas. Fueron introducidas por Harel [Har87] e incorporadas a las diferentes versiones de UML con algunas variaciones. En esta sección presentamos los principales conceptos y definiciones de los SCs basados en [vdB02], aunque existen formalizaciones alternativas que podrían considerarse, como [JEJ04, LMM99], entre otras.

Los SCs son una generalización de los autómatas finitos. Estos consisten esencialmente de estados y transiciones entre ellos. La principal característica de los SCs es que sus estados pueden refinarse, definiendo así una jerarquía de estados. La descomposición de un estado puede ser secuencial o paralela. En la primera un estado se descompone en un autómata (estado Or), mientras que en la segunda se descompone en dos o más autómatas que se ejecutan concurrentemente (estado And).

En la figura 1 el estado Or s_0 es el estado de más alta jerarquía, que se descompone en 4 estados: s_1, s_2, s_3 y s_8. El estado s_3 a su vez se descompone paralelamente en los estados s_4 y s_7 (la descomposición paralela se indica con línea punteada).

Una *configuración de un SC* representa el estado del sistema en un instante dado [vdB02]. Para describirla basta dar el conjunto de estados del SC en los cuales se encuentra el sistema. En el SC del ejemplo, las posibles configuraciones del sistema son $\{s_1\}$, $\{s_2\}$, $\{s_8\}$, $\{s_3, s_5, s_9\}$, $\{s_3, s_6, s_9\}$, $\{s_3, s_5, s_{10}\}$, $\{s_3, s_6, s_{10}\}$.

Más formalmente, siguiendo [vdB02], denominamos *S, TR, Π y A (Π ⊆ A)* al conjunto de los estados, transiciones, eventos y acciones respectivamente de un SC. Asimismo, definimos a un estado $s \in S$ o bien como un término básico de la forma $s = [n]$, como un término Or de la forma $s = [n, (s_1,...,s_k), l, T]$, o como un término And de la forma $s = [n, (s_1,...,s_k)]$, donde $nombre(s) =_{def} n$ es el nombre del estado. Para estados compuestos, $sub\_est(s) =_{\text{def}} (s_1,...,s_k)$ es la secuencia de estados componentes de s, $inicial(s) =_{def} s_1$ es el estado inicial de s, $T \subseteq TR$ es el conjunto de transiciones internas de s y l es el subestado activo de s. En la notación gráfica se etiqueta cada estado con su nombre, siendo las otras componentes deducibles del contexto.

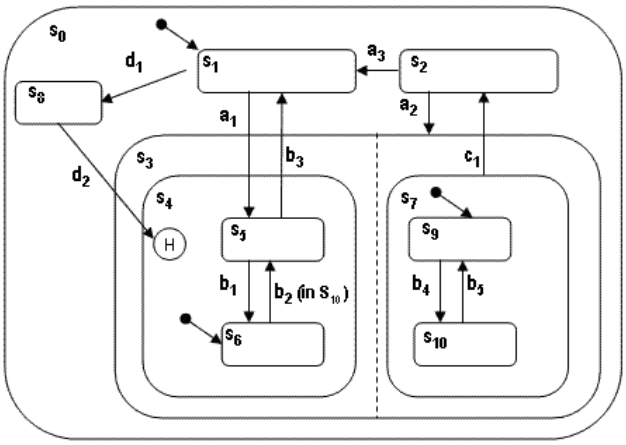


Figura 1. Representación gráfica de un SC.

Las *transiciones* son dirigidas. Una transición se representa como una tupla $t = (t', s_o, e, c, \alpha, s_d, ht)$, donde $nombre(t) =_{\text{df}} t'$ es el nombre de la transición, $origen(t) =_{\text{df}} s_o$ es el estado origen, $ev(t) =_{\text{df}} e$ es el

evento que la "dispara", $cond(t) =_{df} c$ la condición de disparo, $acc(t) =_{df} \alpha$ es la secuencia de acciones que se realizan al efectuarse la transición, $dest(t) =_{df} s_d$ es el estado destino e $hist(t) =_{df} ht$ es el tipo de historia del estado destino (este concepto se explica más adelante).

La notación gráfica utilizada en las transiciones es *t: e,c /α,* siendo las otras componentes de la transición deducibles del dibujo. Para que se pueda realizar una transición *t: e,c /α* con estado origen s_o y estado destino s_d es necesario que el sistema se encuentre en una configuración a la cual pertenezca s_o, ocurra el evento *e* y la condición *c* se cumpla. La realización de la transición produce la ejecución de las acciones de *α* y el cambio de la configuración. Los SCs de UML 2.0 hacen una clasificación de los tipos de eventos internos o externos que pueden ocurrir: de señal (evento asíncrono), de llamada (solicitud de una operación), de cambio (when(expr)) y temporal (alter(time)). En el ejemplo de la figura 1 sólo mostramos los eventos asociados a cada transición y en el resto de este artículo no haremos distinciones entre los distintos tipos de eventos, ya que en la propuesta serán tratados por igual.

Cuando una transición tiene como destino un estado compuesto debe aclarase cuál de los componentes de este estado es el destino de la misma. En el caso de un estado compuesto en forma paralela deben existir transiciones a cada uno de los componentes del mismo (*fork*). Para ello, la componente *ht* de una transición *t* es un elemento del conjunto {*ninguna, profunda, superficial*}, que indica a dónde llegarán las transiciones que tienen como destino un estado. El tipo de historia *ninguna* refiere al estado por defecto o inicial del estado compuesto. En el gráfico, este tipo de historia se indica representando la transición como una flecha al super-estado, y el estado por defecto se marca con una flecha hacia él, sin origen (s_6 y s_9, en el ejemplo). Los otros dos tipos de historia indican que la transición tiene como destino el último estado visitado de un estado compuesto (aunque al inicio es el estado por defecto): *profunda* indica que la transición entrará en el subestado más interno posible, y *superficial* al del primer nivel. Para los tipos de historia *profunda* y *superficial* se utilizan en el gráfico las letras H* y H como destino de la transición, respectivamente. Finalmente, cuando una transición tiene como origen un estado compuesto paralelamente y como destino a uno de nivel superior, la ejecución de esa transición lleva al sistema al estado destino, sin importar el estado en el que se encuentran las otras componentes del estado (*join*). En el ejemplo, estando el sistema en la configuración $\{s_3, s_5, s_9\}$ la transición b_3 lleva al sistema a $\{s_1\}$.

3 Modelos de Funcionalidades

Los *Modelos de Funcionalidades* o *Características* (Feature Models, FMs) se utilizan para describir propiedades o funcionalidades (features) relativas a un dominio. Estas funcionalidades pueden ser obligatorias, opcionales o incluso componentes dentro de un sistema de alternativas, excluyentes o no. Una funcionalidad modela un elemento distintivo de un producto u objeto, y dependiendo del contexto puede referir a un requerimiento, a un componente en un sistema o en una arquitectura, a un fragmento de código, entre otros. Los FMs permiten describir las similitudes y diferencias entre los distintos productos de una PL, y establecer relaciones entre las funcionalidades comunes y las variantes de la línea.

Existen múltiples notaciones para describir FMs, entre otras [CE00, KLD02, GBS01]. En este trabajo usaremos la propuesta por Czarnecki [CE00], por ser simple, clara, expresiva y admitir extensiones, incluso para modelar características no funcionales de un dominio. La figura 2 describe un posible FM en el dominio de la tecnología de teléfonos móviles (TM, de aquí en más) usando la notación de Czarnecki. Por razones de espacio y claridad sólo exhibimos un subconjunto de todas las posibles funcionalidades de un TM.

Como ilustra la figura 2, un FM puede ser descrito gráficamente a través de una estructura arborescente, donde los nodos representan las funcionalidades y los arcos las relaciones entre éstas. Existen cuatro tipos de relaciones fundamentales entre funcionalidades: *obligatoria*, *opcional*, *alternativa* y *disyuntiva*. Las dos primeras relaciones vinculan una funcionalidad padre con una funcionalidad hija, mientras que las dos últimas relacionan una funcionalidad padre con una o más funcionalidades hijas. La relación *obligatoria* indica que si la funcionalidad padre está presente, entonces la funcionalidad hija también lo está siempre en los productos de la PL. Por ejemplo, cada TM tiene un display y un directorio telefónico. Una relación *opcional* indica que la funcionalidad hija puede o no estar presente en un producto cuando su funcionalidad padre es incluida. Por ejemplo, hay TMs con y sin cámara digital. La relación *alternativa* determina que sólo una funcionalidad del conjunto de funcionalidades hijas debe estar presente cuando la funcionalidad padre lo está. Por ejemplo, si un TM dispone de conexión inalámbrica, ésta debería ser por infrarrojo o bluetooh, pero no por ambas. Finalmente, en una relación *disyuntiva* al menos una funcionalidad del conjunto de funcionalidades hijas debe estar presente en un producto si su funcionalidad padre es incluida. Por ejemplo, un TM puede tener búsqueda en el directorio telefónico directa, por strings o ambas. En el árbol cada tipo de relación tiene una notación propia, tal cual se muestra en la figura 2.

Una *configuración de un FM* (*ConfFM*, en adelante) es una instancia de la estructura arborescente que describe al modelo, que respeta la semántica de las relaciones que lo constituyen. Esto es, un FM permite identificar las funcionalidades comunes y las variantes entre los productos de una PL, mientras que una configuración de un FM caracteriza las funcionalidades de un producto específico de la PL. Por ejemplo, según la figura 2 un TM básico podría corresponder a una configuración que posee sólo display y directorio telefónico, y en este último caso tanto con búsqueda directa como por strings. Podrían obtenerse TMs más completos a partir de configuraciones que incluyan distintas características opcionales, como por ejemplo incorporar una cámara digital de 1Mpixel.

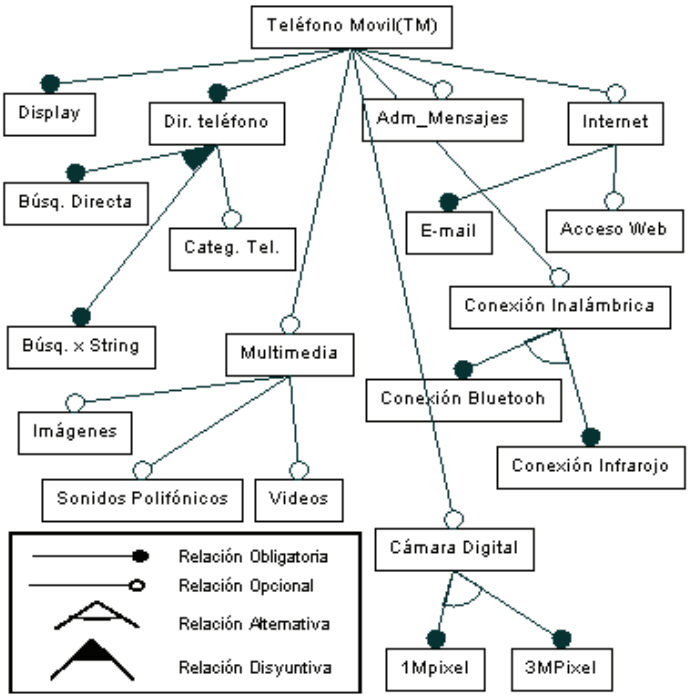


Figura 2. FM de un Teléfono Móvil.

Las funcionalidades que están presentes en toda configuración de un FM, es decir en todo producto de la PL que caracteriza el FM, constituyen el *núcleo del modelo*. La relación *obligatoria* permite modelar el núcleo de un sistema. Por ejemplo, sólo TM, Display y Dir. teléfono constituyen el núcleo del FM de la figura 2. Más formalmente,

Definición 1. Un ***FM*** queda definido por una estructura arborescente descripta por la tupla *(Funcs, f_0, Obl, Opc, Alt, Disy)*, donde *Funcs* es el conjunto de las funcionalidades de un dominio (los nodos del árbol), $f_0 \in Funcs$ es la funcionalidad raíz del árbol de funcionalidades y, *Obl*, *Opc*, *Alt*, *Disy* $\subseteq Funcs \times (\wp(Funcs) - \{\varnothing\})$ son las relaciones obligatoria, opcional, alternativa y disyuntiva del modelo, respectivamente, que representan las aristas del árbol. Si $(f,sf) \in Obl \cup Opc$ entonces #sf = 1.

Ejemplo 1. Según el ejemplo de la figura 2:

- *Funcs* = {TM, Display, Dir. Teléfono, Adm_Mensajes, Internet, Búsq. Directa, Búsq. × String, Categ. Tel., E-mail, Acceso Web, Multimedia, Conexión Inalámbrica, Imágenes, Sonidos Polifónicos, Videos, Conexión Bluetooh, Conexión Infrarrojo, Cámara Digital, 1Mpixel, 3MPixel}.
- f_0 = TM.
- *Obl* = {(TM,{Display}), (TM,{Dir. Telefono}), (Internet,{E-mail})}.
- *Opc* = {(TM,{Multimedia}), (TM,{Cámara Digital}), (TM,{Conexión Inalámbrica}) (TM,{Adm_mensajes}), (TM,{Internet}), (Multimedia,{Imágenes}), (Multimedia,{Sonidos Polifónicos}), (Multimedia,{Videos}), (Internet,{Acceso Web}), (Dir. Teléfono,{Categ. Tel.}).

- *Alt* = {(Conexión Inalámbrica,{Conexión Bluetooh, Conexión Infrarrojo}), (Cámara Digital, {1Mpixel, 3MPixel})}.
- *Disy* = {(Dir. Teléfono, {Búsq. Directa, Búsq. × String})}.

Definición 2. Una ***configuración de un FM*** (*Funcs, f_0, Obl, Opc, Alt, Disy)* es un árbol *(F, R)* donde *F* es el conjuntos de nodos y *R* el conjuntos de las aristas; $F \subseteq Funcs$ y $R \subseteq \{(f,sf) \in F \times (\wp(F)-\{\emptyset\}) / \exists sf' \in \wp(Funcs): sf \subseteq sf' \wedge (f,sf') \in Obl \cup Opc \cup Alt \cup Disy\}$. El árbol *(F, R)* cumple las siguientes condiciones:

1. $f_0 \in F$.
2. Para cada $(f,sf) \in Obl$: si $f \in F$ entonces $(f,sf) \in R$.
3. Si $(f,sf) \in Alt$ y $f \in F$ entonces $\exists!\ sf' \in \wp(F)$: $sf' \subseteq sf \wedge (f,sf') \in R \wedge \#sf'=1$.
4. Si $(f,sf) \in Dis$ y $f \in F$ entonces $\exists!\ sf' \in \wp(F)$: $sf' \subseteq sf \wedge (f,sf') \in R \wedge \#sf' \geq 1$.

Notar que las relaciones opcionales son las únicas que, naturalmente, no poseen restricciones en el contexto de una configuración de un FM.

Ejemplo 2. A continuación ilustramos dos configuraciones del FM de la figura 2, previamente referidas:

Configuración 1:
- F_1 = { TM, Display, Dir. Telefono, Búsq. Directa, Búsq. × String }.
- R_1 = { (TM,{Display}), (TM,{Dir. Telefono}), (Dir. Telefono, {Búsq. Directa, Búsq. × String}) }.

Configuración 2:
- F_2 = { TM, Display, Dir. Telefono, Búsq. Directa, Búsq. × String, Cámara Digital, 1Mpixel }.
- R_2 = { (TM,{Display}), (TM,{Dir. Telefono}), (Dir. Telefono, {Búsq. Directa, Búsq. × String}), (TM,{Cámara Digital}), (Cámara Digital, {1Mpixel}) }.

Definición 3. El **núcleo *N* de un FM** (*Funcs, f_0, Obl, Opc, Alt, Disy)* es el conjunto de funcionalidades definido inductivamente por las siguientes reglas:

1. $f_0 \in N$.
2. Si $f_1 \in N$ y $(f_1, \{f_2\}) \in Obl$ entonces $f_2 \in N$.

4 Máquinas de Estados con Variabilidades

Para modelar el comportamiento de una PL especificada a través de un FM definimos una extensión de las máquinas de estados (SCs) que admite variabilidades en sus componentes, logrando precisar el comportamiento de la línea en forma general. Para ello, en esta sección, extendemos los SCs con *elementos opcionales o variantes* y luego establecemos la relación que vincula a estos elementos con las funcionalidades de un FM. Llamaremos StateCharts* (SCs*) a las máquinas de estados extendidas.

4.1 Representación Gráfica de un SC*

La representación de los elementos opcionales que extienden el núcleo del sistema en un SC* puede observarse en la figura 3. Tanto los estados como las transiciones opcionales son destacados gráficamente con líneas de puntos.

Figura 3. Estado y Transición Opcionales.

4.2 Sintaxis Abstracta de un SC*

Siguiendo la formalización de los SCs dada en la sección 2, denominamos *S*, TR*, Π*, A** al conjunto de los estados, transiciones, eventos y acciones de un SC* respectivamente. Ahora los términos que definen un estado tienen un componente adicional $s_{op} \in \{opcional, no\_opcional\}$ que llamaremos *tipoEstado(s)* y que indica si el estado *s* es opcional o no. De igual modo, a las transiciones se les agrega la componente $t_{op} \in \{opcional, no\_opcional\}$ denotada *tipoTrans(t)*. Definimos además los siguientes conjuntos de elementos opcionales de un SC*: $SOp \subseteq S^*$, $TOp \subseteq TR^*$ y $ElemVar = SOp \cup TOp$.

Referenciaremos a los estados directamente por su nombre en caso de que éste sea único en todo un SC*, de lo contrario utilizaremos el punto como separador entre nombres de estados y subestados. El nombre de una transición se forma con el nombre del evento que la dispara seguido por los nombres de los estados origen y destino, respectivamente.

4.3 Caso de Estudio: TMs

Basándonos en el ejemplo del TM, el SC* de la figura 4 especifica el comportamiento de un subconjunto de las funcionalidades del FM descrito en la sección 3[1]. Consideramos aquí un conjunto de teléfonos móviles que comparten algunas características, como la capacidad de reproducir sonidos monofónicos y vibración, agendar contactos en el directorio del teléfono, entre otras. Opcionalmente podríamos incorporar al núcleo de funcionalidades del modelo la capacidad de efectuar llamadas mediante marcado rápido[2], manipular mensajes de texto, administrar contenido multimedia, y la combinación de éstas como mensajes con contenido multimedia. Entendemos por contenido multimedia a imágenes, sonidos polifónicos y videos.

El TM considerado consta de teclas para: dígitos (TDigito), despazamientos laterales (TDer y TIzq) y verticales (TSubir y TBajar), mensajes (TMens) y, para llamar (TLlamar) y finalizar llamadas (TColgar). Para mayor claridad, el evento que se genera por la selección de cualquiera de dichas teclas se representa en el diagrama por el nombre de la misma.

Por cuestiones de espacio, y considerando que no son influyentes para el desarrollo del resto de este artículo, el comportamiento de los estados Adm.Multimedia (alta, baja y modifición de elementos multimedia en el teléfono), Llamada Entrante y Escribir Mens., entre otros, no han sido detallados. Asimismo, algunas transiciones han sido obviadas o no se desarrollan sus especificaciones debido a comportamientos similares en otras secciones del diagrama.

Las características elegidas para el ejemplo buscan abarcar, por un lado, los distintos tipos de elementos de un SC que pueden considerarse opcionales y, por el otro, reflejar los diferentes grados de vinculación entre las funcionalidades del TM. Por ejemplo, el tipo de sonido polifónico es una funcionalidad opcional que está involucrada en diferentes áreas del comportamiento del TM. En particular, al seleccionar el estilo de timbre en las llamadas entrantes, la existencia de sonidos polifónicos influye en la navegación. Por otro lado, la funcionalidad Adm_Mensajes (ver FM de la figura 5) es bastante compleja y abarca estados tanto de tipo OR como de tipo AND. Sobre este último caso se puede ver que el evento TMens desde el estado Pantalla Principal conlleva a un estado u otro dependiendo de la condición [in EstadoMensaje...], donde EstadoMensaje es un estado paralelo a Main.

[1] Los estados representados con círculos, sin contenido interior, corresponden a pseudo-estados de elección, según UML 2.0 [OMG04].
[2] El marcado rápido permite llamar al número asociado a un dígito cuando dicho dígito se mantiene presionado por el lapso de 2 segundos.

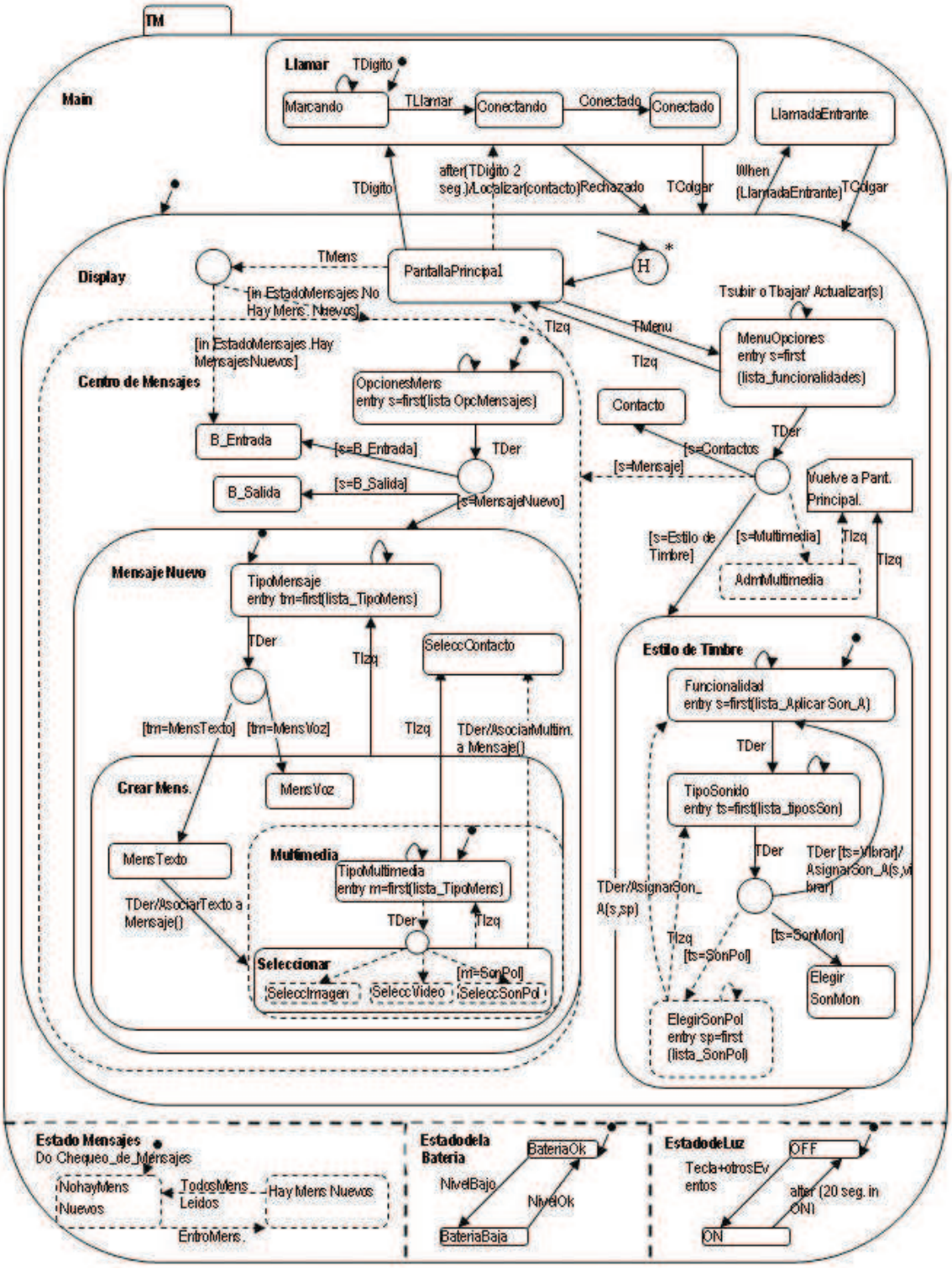


Figura 4. SC* de un TM.

A fin de exhibir un ejemplo completo en el desarrollo de este artículo formulamos en la figura 5 un FM que se relaciona con las funcionalidades involucradas en el TM de la figura 4.

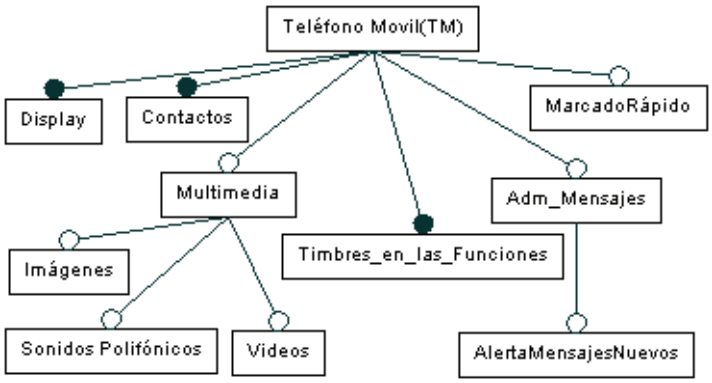


Figura 5. FM del SC* de la figura 4.

Ejemplo 3. A fin de ilustrar las definiciones de la sección 4.2, en el ejemplo de la figura 4 los estados MensajeNuevo, TipoMensaje y CrearMens se representan respectivamente de la siguiente manera: [Mensaje Nuevo, (Tipo Mensaje, CrearMens, SeleccContacto.), 1, $T_{MensajeNuevo}$, no_opcional], [Tipo Mensaje, no_opcional] y [CrearMens., (MensTexto, Multimedia, MensVoz), 1, $T_{CrearMens.}$, no_opcional]. Asimismo, las siguientes tuplas, entre otras, pertenecen al conjunto de transiciones $T_{MensajeNuevo}$: (TDerTipoMensaje-MensTexto, TipoMensaje, TDer, [tm=MensTexto], <>, CrearMens.MensTexto, ninguna, no_opcional), (TIzqCrearMens-TipoMensaje, CrearMens, TIzq, true, <>, TipoMensaje, ninguna, opcional), (TDerTipoMensaje-MensVoz, TipoMensaje, TDer, [tm=MensVoz], <>, CrearMens.MensVoz, ninguna, no_opcional), (TDer, TipoMultimedia, TDer, true, <>, SeleccContacto, ninguna, no_opcional). Si tenemos en cuenta sólo el estado MensajeNuevo, los conjuntos *S*, TR*, SOp, TOp* y *ElemVa*r incluyen los siguientes elementos: *S** = {MensajeNuevo, TipoMensaje, CrearMens, MensVoz, MensTexto, Multimedia, Multimedia.TipoMultimedia, Seleccionar, SeleccImagen, SeleccVideo, SeleccSonPol, SeleccContacto}, *SOp* = {Multimedia, SeleccImagen, SeleccVideo, SeleccSonPol}, *TR** = $T_{MensajeNuevo} \cup T_{TCrearMens} \cup T_{TMultimedia}$, *TOp* = {TDerTipoMultimedia-SeleccImagen, TDerTipoMultimedia-SeleccVideo, TderSeleccionar-SeleccContacto, ...} y ElemVar = *SOp* ∪ *TOp*.

Ejemplo 4. El FM de la figura 5 queda definido por la tupla (*Funcs, f_0, Obl, Opc, Alt, Disy)*, donde:

- *Funcs* = {TM, Display, Contactos, Adm_Mensajes, Multimedia, Imágenes, Sonidos Polifónicos, Videos, MarcadoRápido, Timbres_en_las_Funciones, AlertaMensajesNuevos }.
- f_0 = TM.
- *Obl* = {(TM,{Display}), (TM,{Contactos}), (TM,{Timbres_en_las_Funciones})}.
- *Opc* = {(TM, {Multimedia}), (TM, {Adm_mensajes}), (TM, {MarcadoRápido}), (Multimedia, {Imágenes}), (Multimedia, {Sonidos Polifónicos}), (Multimedia, {Videos}), (Adm_Mensajes, {AlertaMensajesNuevos})}.
- *Alt* =∅ .
- *Disy* = ∅.

4.4 Relación entre un FM y un SC*

Un FM y un SC* son complementarios, dado que ambos modelan diferentes aspectos de un sistema, pero a la vez no son independientes, ya que los elementos del SC* modelan comportamientos de funcionalidades presentes en el FM. En general, una funcionalidad es descripta por más de un elemento de un SC*, razón por la cual debemos introducir una relación que vincule los elementos de un FM con los de un SC* asociado.

Definimos entonces una función *Imp* que representa la asociación entre los elementos variables de un SC* y las funcionalidades de un FM vinculado al SC*. De esta manera establecemos qué elementos variantes del SC* implementan las características del sistema descriptas en el FM. Dado un FM *(Funcs, Obl, Opc, Alt, Disy)* y un SC* (*S*, TR*, IT*, A*)*, la función *Imp* tiene tipo *Imp*: *Funcs* → *P (ElemVar)*, donde *ElemVar* es el conjunto *SOp*∪*TOp*, *SOp* ⊆ *S** y *TOp* ⊆ *TR**. Los conjuntos de elementos variantes en *Imp* no necesariamente deben ser excluyentes, dado que un estado o una transición pueden ser parte de la implementación de una o más funcionalidades.

Teniendo en cuenta que las funcionalidades obligatorias del FM siempre están presentes en todos los productos de la línea, no es necesario definir qué elementos sintácticos del SC* las implementan. En cambio, si es necesario hacerlo para aquellas funcionalidades que pueden no estar en el FM configurado. *Imp* será entonces una función *parcial* definida sobre los elementos del FM que no pertenecen al *núcleo* del mismo. Por lo tanto, el comportamiento de una PL queda definido por un FM, un SC* y una función de implementación que los vincula.

Ejemplo 5. Considerando el FM de la figura 5 y el SC* del caso de estudio de la figura 4, definimos los elementos del SC* que implementan las funcionalidades del TM como sigue:

Imp(SonidosPolifónicos) = {SeleccSonPol, TDerTipoMultimedia-SeleccSonPol, ElelgirSonPol, TDerTipoSonido-ElegirSonPol, TIzqElegirSonPol-TipoSonido, TDerElegirSonPol-Funcionalidad ...},
Imp(Multimedia) = {Multimedia, AdmMultimedia, TDerMultimedia.Seleccionar-SeleccContacto} ∪ *Imp*(Imágenes) ∪ *Imp*(SonidosPolifónicos) ∪ *Imp*(Videos),
Imp(Adm_Mensajes) = { CentrodeMensajes, TMens-PantallaPrinicipal-B_Entrada, TMens-PantallaPrinicipal-CentrodeMensajes, TIzq-CentrodeMensajes-PantallaPrinicipal, TDer-MenuOpciones-CentrodeMensajes} ∪ *Imp*(AlertaMensajesNuevos),
Imp(AlertaMensajesNuevos) = { EstadoMensajes, TMens-PantallaPrincipal-B_Entrada}.

Las definiciones de las funcionalidades Videos e Imágenes son similares a la de SonidosPolifónicos. Es en particular interesante resaltar la implementación de la funcionalidad MarcadoRápido. Esta funcionalidad es implementada por la transición que va desde el estado PantallaPrincipal al estado Llamar, es decir, *Imp*(MarcadoRápido) = {afterTDigito2seg-PantallaPrincipal-Llamar}, donde TDigito2seg-PantallaPrincipal-Llamar es (TDigito2seg, PantallaPrincipal, after(Tdigito 2 seg), true, LocalizarContacto(), Llamar, ninguna, opcional). Seguramente en el detalle del comportamiento del estado Contactos aparezcan más elementos variantes relacionados con esta característica que son omitidos en nuestro caso de estudio por razones de espacio.

5 Instanciación de Máquinas de Estados con Variabilidades

Una configuración de un FM define un producto o sistema concreto, dado un conjunto de características seleccionadas. A partir de una configuración de un FM y del SC* correspondiente al FM (vinculados a través de una función de implementación), definimos una *función de instanciación* que retorna un SC que especifica el comportamiento del producto definido, $Inst: SC^* \times ConfFM \rightarrow SC$.

Utilizamos la función *Imp* definida en la sección anterior para eliminar del SC* todos aquellos estados y transiciones que implementan funcionalidades que no están presentes en la configuración del FM. La eliminación directa de estados y transiciones del SC* no es trivial, ya que la supresión de componentes de un SC* en forma no controlada puede tornar al resultado inconsistente (podrían quedar, por ejemplo, estados inalcanzables o transiciones sin destino). Se define entonces un mecanismo de control y (re)construcción de un SC a partir de un SC* a fin de obtener un producto concreto.

Dado un FM y una configuración de éste, llamaremos *FNS* al conjunto de las *funcionalidades* del modelo *no seleccionadas* por la configuración. Más formalmente, para el FM $fm = (Funcs, f_0, Obl, Opc, Alt, Disy)$ y una configuración $conf_{fm} = (F, R)$ del mismo, $FNS = Funcs - F$. Definimos además el conjunto de los *componentes no seleccionados* (*CNS*), por la configuración $conf_{fm}$, del SC* *sc**, que no formarán parte del SC resultante, como:

$CNS\ (conf_{fm}, sc^*) = \{ x \in ElemVar \mid \exists f_i \in FNS: x \in Imp(f_i) \wedge \neg\exists f_{i'} \in F: f_{i'} \neq f_i \wedge x \in Imp(f_{i'}) \}$, con *ElemVar* e *Imp* definidos para *sc** según detalla la sección 4.4. Esto es, los estados y las transiciones que no implementan funcionalidades seleccionadas por una configuración serán excluidos del comportamiento del SC resultante.

El proceso de instanciación puede resumirse en el siguiente algoritmo:

```
SC Inst (sc*: SC*, cfm: ConfFM) {
   return InstRec (sc*, CNS(cfm, sc*));
}
```

Donde:

```
SC InstRec (sc*: SC*, cns: P(SOp ∪ TOp)) {
   if cns = ∅  return ConvOpcional_NoOpcional(sc*);
   else {
        select x ∈ cns;
        switch (TipoElementoVar(x)) {
            case estado_simple:
                cns = cns – ({x} ∪ {t:TOp| t es transición de entrada o salida
                opcional a x});
                /*  Se  eliminan  de  cns  el  elemento  variante  x  y  las
                transiciones opciones de entrada y salida */
                sc* = (Eliminar x del sc*; Reconstruir el sc*);
                /* Caso 1.1, 2, 3 y 4 */
                return InstRec(sc*, cns);
            case estado_or:
                cns = cns – ({x} ∪ {t:Top | t es transición de entrada o
                salida opcional a x} ∪ SubComponentes(x, sc*))
                /* Se eliminan de cns el estado variante x, las transiciones
                opciones de entrada y salida de x, y los posibles subestados y
                transiciones internos de x */
                sc* = (Eliminar x del sc*; Reconstruir el sc*);
                /* Casos 1.2, 2, 3 y 4 */
                return InstRec(sc*, cns);
            case estado_and:
                cns = cns – ({x} ∪ {t:Top | t es transición de entrada o
                salida opcional a x} ∪ SubComponentes(x, sc*);
                /* Se eliminan de cns el elemento variante x, las transiciones
                opciones de entrada y salida, y los posibles subestados y
                transiciones internos de x */
                sc* = (Eliminar x del sc*; Reconstruir sc*);
                /* Casos 1.3, 2, 3 y 4 */
                return InstRec(sc*, cns);
            case transición:
                cns = cns – ({x} ∪ CompontesInalcanzables(x, sc*));
                /* Se eliminan de cns el elemento variante x y los componentes
                inalcanzables de sc* por la eliminación de x */
                sc* = (Eliminar x del sc*; Reconstruir el sc*);
                /* Ver Caso 5 */
                return InstRec(sc*, cns);
          }
}
```

La función *ConvOpcional_NoOpcional(sc*)* convierte los elementos *opcionales* de *sc** en *no_opcionales* (esto es, en elementos del SC correspondiente); la función *TipoElementoVar(x)* retorna el tipo de elemento variante de *x (estado_simple, estado_or, estado_and, transición)*; la función *SubComponentes(x, sc*)* retorna el conjunto de los subestados y las transiciones internas relacionadas al estado *x* de *sc**; y, la función *ComponentesInalcanzables(x, sc*)* retorna el conjunto de los estados y las transiciones inalcanzables de *sc** luego de la eliminación del estado *x*.

Las reglas de reconstrucción se describen en la sección 5.1. Al finalizar el algoritmo, los elementos opcionales del SC* que aún permanecen pertenecen a la especificación de las funcionalidades del producto

concreto, razón por la cual son transformados a estados y transiciones del SC resultante a través de la función *ConvOpcional_NoOpcional*.

5.1 Reglas de Reconstrucción

Para definir los métodos de reconstrucción considerados en el algoritmo previo, nos basamos en los casos sobre las cuales éste se aplica.

Caso 1. Eliminación de un Estado

Caso 1.1 Eliminación de un Estado Simple

Si un estado simple *E* desaparece, también desaparecen sus transiciones de entrada y salida opcionales, mientras que las obligatorias se componen usando el siguiente método de reconstrucción.

Método de reconstrucción

Sea $E \in SOp$ el estado a eliminar, $A_1,...,A_n$ estados predecesores de *E* (esto es, estados desde los cuales hay transiciones no opcionales hacia *E*: $t_{AE\_1, ..., } t_{AE\_n}$), y S_1 ,..., S_m estados sucesores de *E* (esto es, estados hacia los cuales llegan transiciones no opcionales desde *E*: $t_{ES\_1, ..., } t_{ES\_m}$), según muestra la figura 6. Al desaparecer el estado variante *E* todas las transiciones de entrada y de salida que vinculan a *E* se eliminan, a la vez que se generan nuevas transiciones por la composición de las transiciones no opcionales de entrada ($t_{AE\_1, ..., } t_{AE\_n}$) con las de salida ($t_{ES\_1, ..., } t_{ES\_m}$).

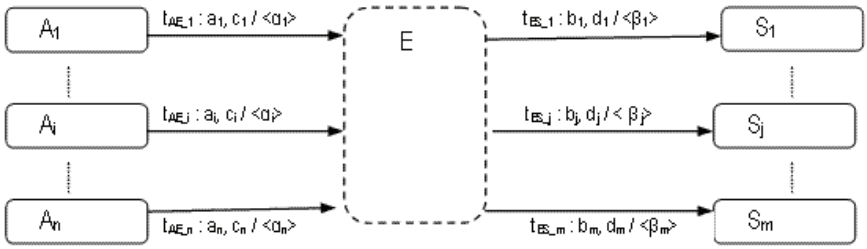


Figura 6. SC* generalizado del caso 1.

La composición de dos transiciones $t_1 = (t_1, so_1, e_1, c_1, \alpha_1, sd_1, ht_1, no\_opcional)$ y $t_2 = (t_2, so_2, e_2, c_2, \alpha_2, sd_2, ht_2, no\_opcional)$ se define como una nueva transición $t_1t_2 = (t_{12}, so_1, e_1::e_2, c_1 \wedge c_2, \alpha_1::a_2, sd_2, ht_2, no\_opcional)$, donde *::* es la composición secuencial de eventos y acciones, y $\wedge$ la conjunción de condiciones. Ambas operaciones deben ser asociativas para que el algoritmo de instanciación resulte determinístico.

En general, el resultado de eliminar el estado *E* se define como sigue: Sean $t_{E1},..., t_{En} \in TR^*$, tales que $dest(t_{Ei}) = E$, con $1 \leq i \leq n$ y $tipoTrans(t_{Ei}) = no\_opcional$, y sean $t_{1E}, ..., t_{mE} \in TR^*$, tales que $origen(t_{jE}) = E$, con $1 \leq j \leq m$ y $tipoTrans(t_{iE}) = no\_opcional$. Por cada transición t_{Ei} se crean *m* transiciones $t_{ij} = t_{Ei}\ t_{jE}$. En la figura 7 se muestra el resultado de la aplicación de esta regla al SC* de la figura 6.

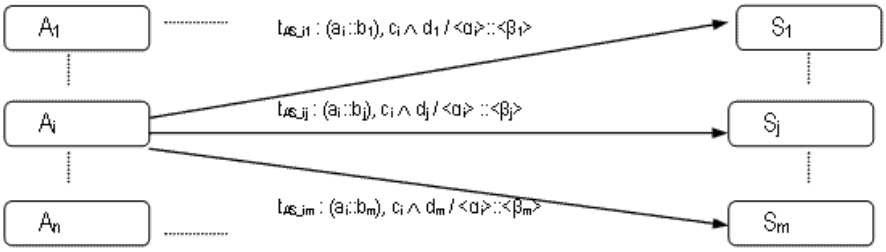


Figura 7. SC* luego de la eliminación del estado opcional de la figura 6.

Observación: Alternativamente podría considerarse que las transiciones remanentes luego de la supresión de un estado opcional sean especificadas como un parámetro más del modelo, siendo en este caso el método de reconstrucción detallado para este caso la opción por defecto. Si bien la alternativa descripta aumentaría en cierta medida la complejidad de los diseños, permitiría flexibilizar el comportamiento de las opcionalidades sobre los mismos.

Caso 1.2 Eliminación de un estado Or

En caso de eliminación de un estado variante Or *E* debemos tener en cuenta que las transiciones de entrada y salida al mismo son una abreviatura de transiciones de entrada y salida a sus estados internos. Por lo tanto, se podría aplicar el caso 1.1 a cada uno de los subestados componentes de *E*, recursivamente. Finalmente, se eliminaría el propio estado *E*.

Si bien esta propuesta de solución es válida, teniendo en cuenta que cualquier SC con estados jerárquicos tiene un equivalente plano (sin estados compuestos), produce un gran número de transiciones entre los estados antecesores y sucesores a *E*. Como consecuencia, este encare abandona la practicidad del concepto de jerarquía de los SCs. Podemos ver en la figura 8 el resultado para un pequeño ejemplo. Tener presente que una transición de entrada directamente al estado *E* es una transición con destino el estado por defecto de *E*. De la misma manera, una transición de salida de *E* es una transición de salida desde todos los subestados de *E*.

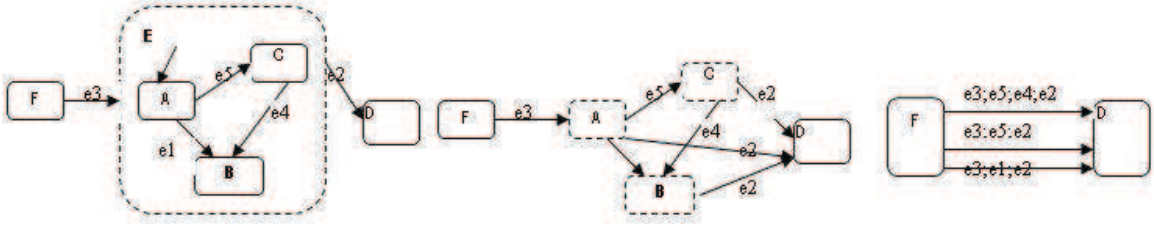


Figura 8. Una posibilidad para la eliminación de un estado variante Or.

Método de reconstrucción

A partir del análisis sobre las transiciones de entrada y salida de *E* para el ejemplo previo es que proponemos una solución para el caso de eliminación de un estado variante Or. La propuesta consiste en aplicar el mismo método de composición de transiciones del caso 1.1 sobre *E*, teniendo en cuenta ciertas condiciones y afectaciones al SC*. A continuación se definen las condiciones que deben cumplir las transiciones de entrada y salida, y como es alterado previamente el SC* antes de aplicar la composición de transiciones según el caso 1.1.

Sea $T_{e\_s}(E) = \{ (t_e, t_s) \in TR^* \times TR^* \mid dest(t_e) \in \text{subestados}(E) \wedge origen(t_s) \in \text{subestados}(E) \}$ el conjunto de todos los pares de transiciones de entrada y salida a *E,* donde *TR** es el conjunto de transiciones del SC* y subestados(*E*) es el conjunto de subestados del estado *E*.

Decimos que cada transición de entrada a *E* se compone con una de salida si el estado origen de la transición de salida es alcanzable desde el estado destino de la transición de entrada. Definimos *Alc(E, A)* como el conjunto de subestados de *E* alcanzables a partir del subestado *A*.

En la figura 9 podemos ver un caso donde tras la eliminación de *E* es posible aplicar la composición de las transiciones *e3* con *e6*, debido a que el estado $B \in Alc(E, A)$. En tanto que las transiciones *e7* y *e2* no se deberían componer ya que $C \notin Alc(E, B)$.

Más formalmente, definimos $TComp_{e\_s}(E) = \{ (t_e, t_s) \in T_{e\_s}(E) \mid origen(t_s) \in Alc(E, dest(t_e)) \}$, como el conjunto de pares de transiciones que deben componerse mediante el caso 1.1, previa modificación de dichas transiciones como se indica a continuación. Para toda transición de entrada $t_e \in Dominio(TComp_{e\_s}(E))$ su estado destino es ahora *E*, es decir, $dest(t_e)=E$ (más precisamente el término que define *E*). Asimismo, para toda transición de salida $t_s \in Rango(TComp_{e\_s}(E))$, $origen(t_e)=E$. Donde, *Dominio* y *Rango* representan el dominio y rango, respectivamente, de una relación binaria.

Luego procedemos a la eliminación de:

- todos los subcomponentes (subestados y transiciones involucradas) de *E*,
- las transiciones opcionales de entrada y salida, y
- las transiciones de entrada a *E* no opcionales que no pertenecen al conjunto $Dominio(TComp_{e\_s}(E))$ y las transiciones de salida de *E* no opcionales que no pertenecen al conjunto $Rango(TComp_{e\_s}(E))$.

Finalmente componemos sólo los pares de transiciones pertenecientes a $TComp_{e\_s}(E)$ según el caso 1.1.

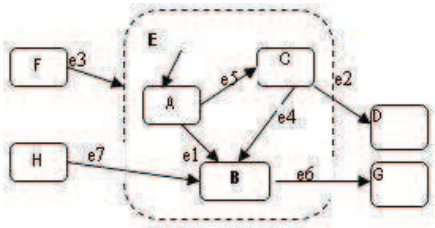
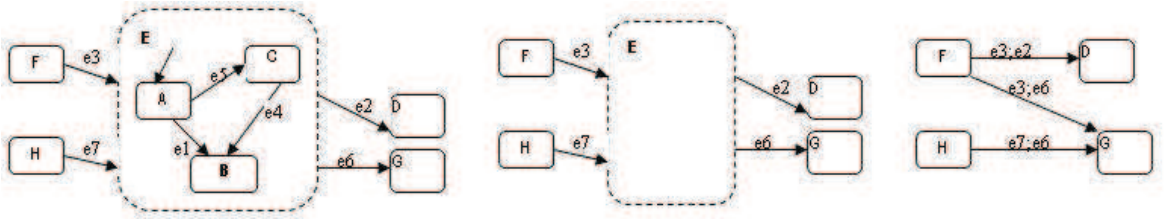


Figura 9. Eliminación de un estado variante *Or*.

Ejemplo 6. Considerando el SC* de la figura 9, los conjuntos $T_{e\_s}(E)$ y $TComp_{e\_s}(E)$ se definen como sigue. $T_{e\_s}(E) = \{(e3,e2), (e3, e6), (e7,e2), (e7,e6)\}$ y $TComp_{e\_s}(E) = \{(e3,e2), (e3, e6), (e7,e6)\}$. Notar que *(e7,e2)* no forma parte de $TComp_{e\_s}(E)$ debido a que $origen(e2) \notin Alc(E, dest(e7))$. Las modificaciones de las transiciones *e3* y *e7,* como transiciones de entrada, y de *e2* y *e6*, como transiciones de salida, se observan en el SC* de la figura 10(a). Posteriormente se eliminan todos los componentes internos, transiciones de entrada y salida opcionales, y las transiciones de entrada y salida que no pertenecen a $Dominio(TComp_{e\_s}(E))$ y $Rango(TComp_{e\_s}(E))$ respectivamente, ver figura 10(b). Finalmente aplicamos la composición de transiciones como en el caso 1.1 pero sólo a los pares de transiciones de $TComp_{e\_s}(E)$. El resultado final de la eliminación del estado *E* se observa en la figura 10(c).



(a) (b) (c)

Figura 10. Supresión de *E* en el SC* de la figura 9.

Caso 1.3 Eliminación de un estado And.

Un estado And *E* es una composición parelela de estados, sus subestados, que pueden ser estados simples, estados Or o incluso nuevos estados And. La eliminación de estados variantes And implica la eliminación de todos sus subestados paralelos.

Método de reconstrucción

Nuevamente, haciendo análisis sobre las posibles transiciones de entrada y salida al estado a eliminar *E*, aplicamos el método de composición de transiciones del caso 1.1, siempre que se satisfagan ciertas condiciones.

Una transición de entrada directa a *E* es una transición de entrada a cada uno de sus subestados, y más precisamente a sus estados iniciales. Una transición de salida directa desde *E* es una transición de salida desde cada uno de sus subestados. La aplicación del método en este escenario es directa. Sin embargo, la solución se torna algo más compleja cuando existen transiciones de entrada a subestados internos de los estados de *E* y/o transiciones de salida desde alguno de éstos (transiciones internivel).

Existen dos posibles relaciones de dependencia o sincronización entre estados paralelos. Una de ellas refiere a la ocurrencia de un evento que produce el disparo de dos o más transiciones pertenecientes a cada uno de los subestados paralelos. La segunda corresponde a la utilización de condiciones de tipo "*in E*" (ver caso 3 más adelante). Este último tipo de sincronización impide que el concepto de alcanzabilidad en estados paralelos sea definido de manera independiente en cada uno de los estados ortogonales.

El concepto de alcanzabilidad descrito anteriormente es extendido y aplicado nuevamente para definir cuando una transición de entrada se compone con una de salida. Un estado And es un producto ortogonal de los subestados que lo componen [Har87]. En el ejemplo de la figura 11 el estado And *E* es el producto de los estados *L* y *H*. Sea entonces *E* un estado And con *n* estados ortogonales. Definimos $Alc(E, (E_1, E_2, ..., E_n))$ como el conjunto de n-tuplas de estados alcanzables a partir de $(E_1, E_2, ..., E_n)$. De esta manera, manteniendo la definición de $T_{e\_s}(E)$ del caso anterior y redefiniendo $TComp_{e\_s}(E)$ es posible resolver el método de eliminación y composición de transiciones de forma análoga al caso anterior. La idea es que cada transición de entrada a *E* se compone con una de salida si existe una n-tupla alcanzable que contiene al estado origen de la transición de salida, a partir de una n-tupla inicial que contiene al estado destino de la transición.

En la figura 11 vemos que el par de estados $(C,J) \in Alc(Y, (A,D))$, mientras que $(C,J) \notin Alc(Y, (B,D))$, razón por la cual no es posible concebir el disparo de la transición *e8* con condición de disparo "*in C*". En consecuencia, vemos que es posible componer la transición de entrada *e3* con las transiciones de salida *e8* y *e10*, la transición de entrada *e7* con la transición de salida *e10*, pero no es posible componer *e7* con *e8*.

Más formalmente, definimos $TComp_{e\_s}(E) \subseteq T_{e\_s}(E)$ como el conjunto de pares de transiciones que deben componerse como consecuencia de la eliminación del estado *E*.

$$TComp_{e\_s}(E) = \{ (t_e, t_s) \in T_{e\_s}(E) \mid (\exists\ n\text{-}tupla\_inicial,\ n\text{-}tupla\_final \in (S^* \times S^* \times ... \times S^*)_n \mid n\text{-}tupla\_final \in Alc(E,\ n\text{-}tupla\_inicial) \wedge (\exists i,j\ 1 \le i,j \le n \mid n\text{-}tupla\_final[i] = origen(t_s) \wedge n\text{-}tupla\_inical[j] = dest(t_e) \wedge cond(t_s)\ ))\ \}$$, con *n* la cantidad de estados ortogonales en *E*.

A partir del conjunto $TComp_{e\_s}(E)$ se aplica el caso 1.1, previa modificación de las transiciones involucradas, como en el caso anterior. Esto es, para toda transición de entrada $t_e \in Dominio(TComp_{e\_s}(E))$ su estado destino es ahora *E* ($dest(t_e)=E$) y para toda transición de salida $t_s \in Rango(TComp_{e\_s}(E))$, $origen(t_e)=E$.

Luego procedemos a la eliminación de:

- todos los subcomponentes (subestados y transiciones involucradas) de *E*,
- las transiciones opcionales de entrada y salida, y
- las transiciones de entrada a *E* no opcionales que no pertenecen a ninguna n-tupla del conjunto $Dominio(TComp_{e\_s}(E))$ y las transiciones de salida de *E* no opcionales que no pertenecen a ninguna n-tupla del conjunto $Rango(TComp_{e\_s}(E))$.

Finalmente componemos sólo los pares de transición pertenecientes a $TComp_{e\_s}(E)$ según el caso 1.1.

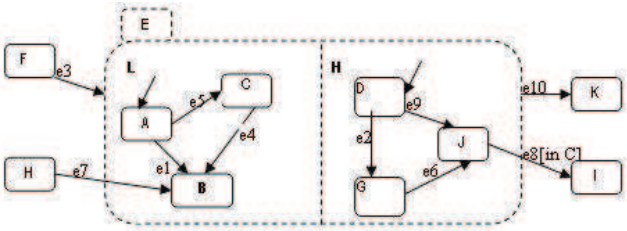
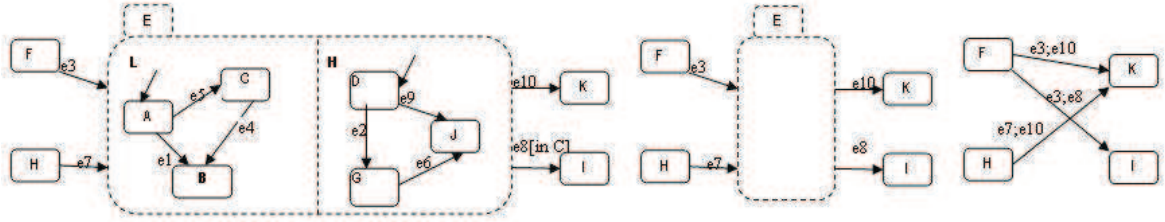


Figura 11. Eliminación de un estado variante *And.*

Ejemplo 7. Considerando el SC* de la figura 11, $T_{e\_s}(E)$ = *{(e3,e8), (e3, e10), (e7,e8), (e7,e10)}* y $TComp_{e\_s}(E)$ = *{(e3,e8), (e3, e10), (e7,e10)}*. Observar que *(e7,e8)* no forma parte de $TComp_{e\_s}(E)$ debido a que a partir del disparo de la transición *e7* no es posible alcanzar una n-tupla en donde la condición de disparo de *e8* se satisfaga. Las modificaciones de las transiciones *e3* y *e7,* como transiciones de entrada, y *e8* y *e10*, como transiciones de salida, se plasman en el SC* de la figura 12(a). Posteriormente se eliminan todos los componentes internos, transiciones de entrada y salida opcionales, y las transiciones de entrada y salida que no pertenecen a *Dominio*($TComp_{e\_s}(E)$) y *Rango*($TComp_{e\_s}(E)$) respectivamente, ver figura 12(b). Finalmente aplicamos la composición de transiciones como en el caso 1.1 pero sólo a los pares de transiciones en $TComp_{e\_s}(E)$, ver figura 12(c). Notar que en la figura 12(b) la condición [in C] de la transición *e8* es eliminada (ver caso 3 "*Eliminación de subestados en una descomposición paralela*", más adelante).



(a) (b) (c)

Figura 12. Supresión de *E* en el SC* de la figura 11.

NOTA: En los casos descritos anteriormente, si el estado que se elimina (o cualquiera de sus subestados) es un estado inicial, debe considerarse adicionalmente el caso 2 subsiguiente. Si se elimina un estado *E* que aparece en una condición "*in E*", es necesario aplicar también el caso 3 y finalmente, ante la eliminación de todos los subestados de un superestado debe contemplarse asimismo el caso 4. Los casos 2-4 se describen a continuación.

Caso 2. Eliminación de un estado inicial

Si se elimina un estado inicial de un estado *E*. Distinguimos dos casos:

Caso 2.1. Si el sucesor del estado inicial de *E* pertenece a los subestados de *E* y es único, pasa a ser el nuevo subestado inicial.

Método de reconstrucción

Sea $s = [E, (s_1,...,s_k), l, T]$ el estado compuesto E de tipo Or, donde $inicial(s) = s_1$. Si existe un único $t \in T$ tal que $origen(t) = inicial(s)$, entonces $inicial(s') = dest(t)$ y $sub\_est(s') = swap(sub\_est(s'), 1, índice(dest(t)))$, donde *s*' es el estado compuesto *E* luego de la eliminación del estado inicial, y la función *swap(sec, i, j)* intercambia en la secuencia *sec* los elementos de las posiciones *i* y *j*, a fin de preservar el estado inicial como el primero de la lista de los subestados de *E*. La eliminación del estado inicial debe efectuarse como lo indica el caso 1.

Caso 2.2. Si existe más de un sucesor del estado inicial de *E* que es subestado de éste, el nuevo estado inicial pasa a ser el segundo estado de la lista de los subestados de *E*.

Método de reconstrucción

Sea $s = [E, (s_1,...,s_k), l, T]$ el estado compuesto E de tipo OR, donde $inicial(s) = s_1$. Si existen $t_1, t_2 \in T$ tales que $t_1 \neq t_2$ y $origen(t_1) = origen(t_2) = inicial(s)$, entonces $inicial(s') = s_2$.

Caso 3. Eliminación de subestados en una descomposición paralela

Es posible establecer como condición de disparo de una transición restricciones de tipo "*in E*", que indican que el disparo se realizará siempre que el estado E pertenezca a la configuración actual de la máquina de estados. Estas restricciones sólo se aplican a estados paralelos, como el de la figura 1, donde se crea una relación de dependencia entre los estados compuestos s_4 y s_7. La condición "*in* s_{10}" causa que la transición b_2 se dispare si la configuración de la máquina es únicamente $\{s_3, s_6, s_{10}\}$. El SC* es afectado si una configuración del FM produce la eliminación del estado s_{10}, ya que deja una inconsistencia en la condición de b_2.

Método de reconstrucción

Las condiciones de las transiciones en una descomposición paralela de tipo "*in E*" se eliminan cuando el estado E es suprimido por alguna configuración del FM.

Caso 4. Eliminación de todos los subestados

Si desaparecen todos los subestados de un superestado E entonces E también desaparece, aplicándose el caso 1. Los subestados pueden desaparecer por estar involucrados en implementaciones de distintas funcionalidades.

Caso 5. Eliminación de una transición

Si una transición t desaparece no produce alteraciones en el SC*, excepto cuando algún estado (o eventualmente toda una subárea) del SC* queda inalcanzable a partir del estado inicial del sistema. En este último caso debería eliminarse el área inalcanzable, aplicando en cada situación (para cada componente) los casos previamente considerados.

5.2 Instanciaciones del caso de estudio: TMs

El FM de la figura 5 puede configurarse para caracterizar a distintos TMs, según las especificaciones del caso de estudio de la sección 4.3. Por ejemplo, podríamos configurar un TM sin la posibilidad de manipular sonidos polifónicos, sin el manejo de mensajes cortos, sin el chequeo automático de mensajes nuevos o inclusive, sin combinaciones de éstas funcionalidades, entre otras configuraciones. Para ilustrar lo expuesto a lo largo del trabajo exhibiremos 2 configuraciones del TM de la figura 5 y procederemos a obtener los SCs correspondientes –los TMs deseados– a través del algoritmo de instanciación presentado en la sección 5.

5.2.1 TM sin sonidos pólifonicos ni alerta de mensajes nuevos

Un TM sin incluir soporte para manipular sonidos polifónicos y sin la capacidad de alertar cuando ingresan mensajes nuevos al buzón de entrada queda definido por la configuración $conf_{fm} = (F, R)$ del FM de la figura 5, donde:

- F = {TM, Display, Contactos, Adm_Mensajes, Multimedia, Imágenes, Videos, MarcadoRápido, Timbres_en_las_Funciones}.
- R = {(TM, {Multimedia}), (TM, {Adm_mensajes}), (TM, {MarcadoRápido}), (TM, {Display}), (TM, {Contactos}), (TM, {Timbres_en_las_Funciones}), (Multimedia, {Imágenes}), (Multimedia, {Videos})}.

Teniendo en cuenta la configuración anterior y la función *Imp* descripta en el ejemplo 5 de la sección 4.4, los conjuntos *FNS* y $CNS(conf_{fm}, sc^*)$ quedan definidos como:

- *FNS* = {Sonidos Polifónicos, AlertaMensajesNuevos}.

- *CNS*($conf_{fm}$, *sc**) = {*SeleccSonPol, TDerTipoMultimedia-SeleccSonPol, ElelgirSonPol, TDerTipoSonido-ElegirSonPol, TIzqElegirSonPol-TipoSonido, TDerElegirSonPol-Funcionalidad, EstadoMensajes, TMens-PantallaPrincipal-B_Entrada*}. Llamaremos simplemente *CNS* al conjunto anterior en el resto de esta subsección.

Mostramos a continuación la aplicación de la función *Inst* al SC* de la figura 4 y a la configuración $conf_{fm}$ previamente definida, teniendo en cuenta los conjuntos *FNS* y *CNS* anteriores.

1. $CNS \neq \varnothing$

   `select` *SeleccSonPol* $\in$ *CNS*

   TipoElementoVar(*SeleccSonPol*) = *estado_simple*

   (por razones de espacio y simplicidad algunos estados no han sido explotados/refinados, razón por la cual los asuminos para nuestro ejemplo como estados simples)

   CNS = *CNS* − ({*SeleccSonPol*} ∪ {*TDerTipoMultimedia-SeleccSonPol*})

   Resultado de eliminar y reconstruir: ver figura 13 (sólo se dibuja la parte del SC* afectada)

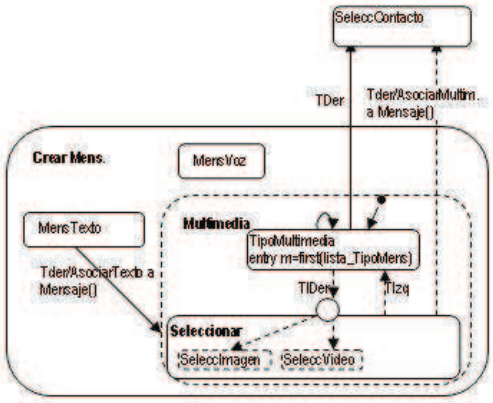


Figura 13. SC* de la figura 4 al procesar *imp*(*SeleccSonPol*).

2. *CNS* = {*ElegirSonPol, TDerTipoSonido-ElegirSonPol, TIzqElegirSonPol-TipoSonido, TDerElegirSonPol-Funcionalidad, EstadoMensajes, TMens-PantallaPrincipal-B_Entrada*}

   `select` *ElegirSonPol* $\in$ *CNS*

   TipoElementoVar(*ElegirSonPol*) = *estado_simple*

   CNS = *CNS* − ({*ElegirSonPol*} ∪ {*TDerTipoSonido-ElegirSonPol, TIzqElegirSonPol-TipoSonido, TDerElegirSonPol-Funcionalidad*})

   Resultado de eliminar y reconstruir: ver figura 14 (sólo se dibuja la parte del SC* afectada).

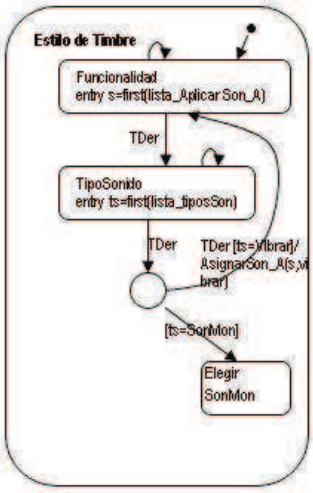


Figura 14. SC* de la figura 4 al procesar *imp*(*ElegirSonPol*).

3. *CNS* = {*EstadoMensajes, TMens-PantallaPrincipal-B_Entrada*}

select *EstadoMensajes* ∈ *CNS*

TipoElementoVar(*EstadoMensajes*) = *estado_or* (*EstadoMensajes* es un estado Or, parte de un estado And)

CNS = *CNS* − {*ElegirSonPol*}

Resultado de eliminar y reconstruir: se limita simplemente a eliminar el estado dado, ya que no hay transiciones de entrada y salida de manera directa a EstadoMensajes. Sin embargo, este caso produce una alteración en las condiciones de las transiciones TMens-PantallaPrincipal-B_Entrada y TMens-PantallaPrincipal-OpcionesMens, pues los subestados NohayMensNuevos y HayMensNuevos forman parte de las condiciones de disparo de éstas. Según el caso 3, las condiciones son eliminadas de las transiciones.

4. *CNS* = {*TMens-PantallaPrincipal-B_Entrada*}

select *TMens-PantallaPrincipal-B_Entrada* ∈ *CNS*

TipoElementoVar(*TMens-PantallaPrincipal-B_Entrada*) = *transición*

CNS = *CNS* − {*TMens-PantallaPrincipal-B_Entrada*}

Resultado de eliminar y reconstruir: La transición se elimina según el caso 5. El SC*, teniendo el paso anterior y el actual, se muestra en la figura 15. Por razones de espacio no visualizamos la eliminación del estado EstadoMensajes, aunque si su afectación sobre la transición TMens-PantallaPrincipal-OpcionesMens.

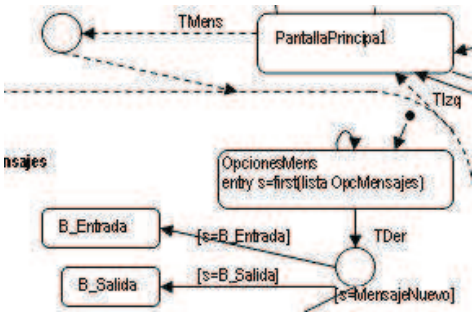


Figura 15. SC* de la figura 4 al procesar *imp*(*TMens-PantallaPrincipal-B_Entrada*).

5. $CNS = \varnothing$

Finalmente, se convierten los componentes opcionales remanentes en elementos del SC resultante, según se observa en la figura 16.

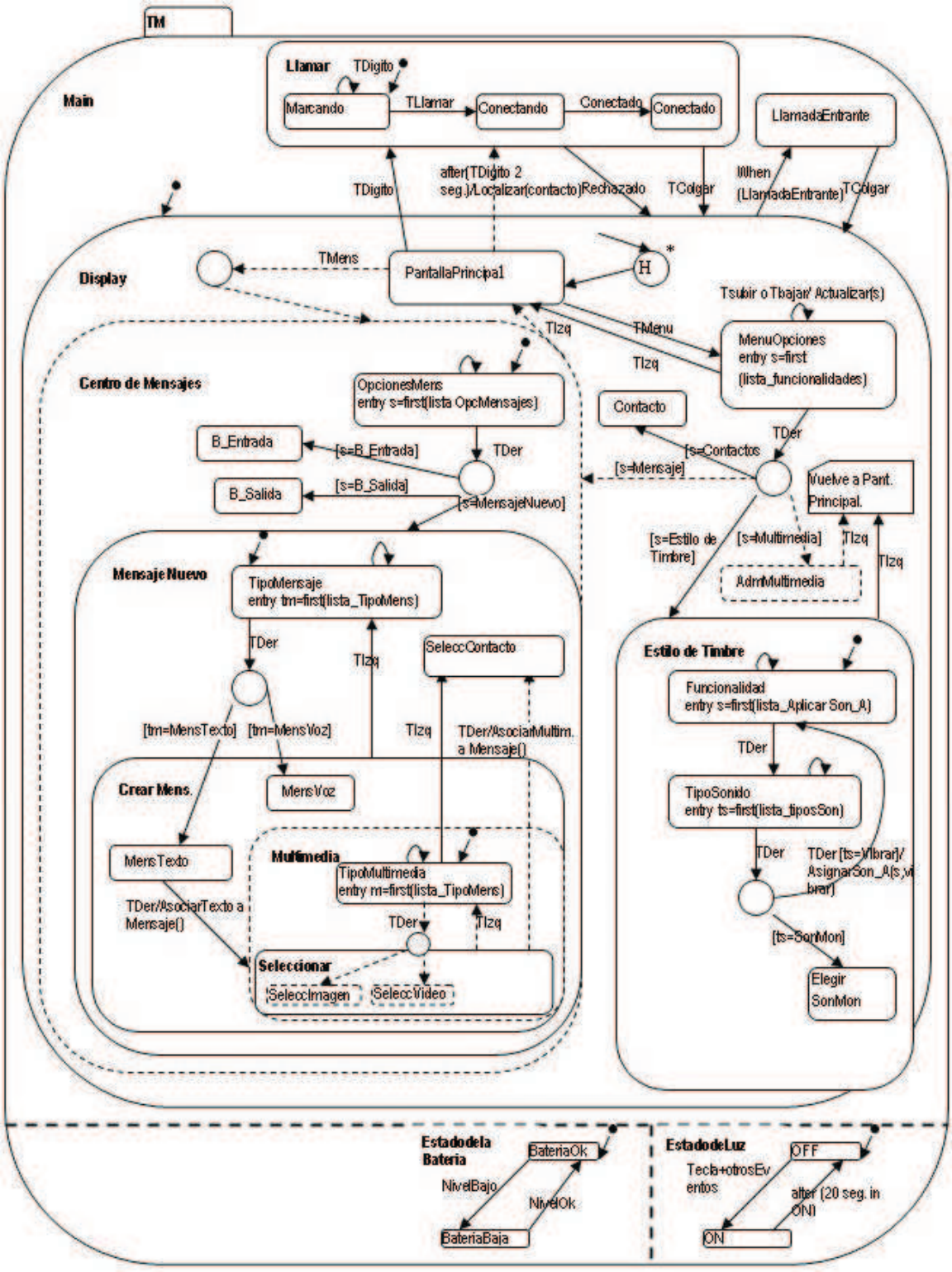


Figura 16. SC para un TM sin sonidos pólifónicos ni alerta de mensajes nuevos.

5.2.2 TM sin multimedia ni marcado rápido

Un TM sin soporte para operar contenido multimedial y que no posee funcionalidades de marcado rápido queda definido por la configuración $conf_{fm} = (F, R)$ del FM de la figura 5, donde:

- F = {TM, Display, Contactos, Adm_Mensajes, Timbres_en_las_Funciones, AlertaMensajesNuevos}.
- R = {(TM, {Display}), (TM, {Contactos}), (TM, {Timbres_en_las_Funciones}), (TM, {Adm_mensajes}), (Adm_mensajes, {AlertaMensajesNuevos })}.

Teniendo en cuenta la configuración anterior y la función *Imp* descripta en el ejemplo 5 de la sección 4.4, los conjuntos *FNS* y $CNS(conf_{fm}, sc^*)$ quedan definidos en este caso como:

- *FNS* = {Multimedia, MarcadoRápido}
- $CNS(conf_{fm}, sc^*)$ = {*Multimedia, AdmMultimedia, TDerMultimedia.Seleccionar-SeleccContacto, SeleccSonPol, SeleccImagen, SeleccVideo, TDerTipoMultimedia-SeleccSonPol, TDerTipoMultimedia-SeleccImagen, TDerTipoMultimedia-SeleccVideo, ElelgirSonPol, TDerTipoSonido-ElegirSonPol, TIzqElegirSonPol-TipoSonido, TDerElegirSonPol-Funcionalidad, afterTDigito2seg-PantallaPrincipal-Llamar*}. Llamaremos nuevamente *CNS* al conjunto anterior en el resto de esta subsección.

Mostramos a continuación la aplicación de la función *Inst* al SC* de la figura 4 y a la configuración $conf_{fm}$ previamente definida, teniendo en cuenta los conjuntos *FNS* y *CNS* anteriores.

1. $CNS \neq \varnothing$

   `select` *Multimedia* $\in$ *CNS*

   TipoElementoVar(*Multimedia*) = *estado_or*

   CNS = *CNS* - {*Multimedia, TDerMultimedia.Seleccionar-SeleccContacto, SeleccSonPol, SeleccImagen, SeleccVideo, TDerTipoMultimedia-SeleccSonPol, TDerTipoMultimedia-SeleccImagen, TDerTipoMultimedia-SeleccVideo*}

   Luego de la eliminación del estado elegido mediante el caso 1.2, el conjunto de componentes no seleccionados es actualizado dando de baja no sólo al estado en cuestión sino también a la transición opcional de salida TDerMultimedia.Seleccionar-SeleccContacto y los subestados que forman parte de *CNS*. Notar que la reconstrucción correspondiente al caso 1.2 produce la composición de la transición de entrada TDerMensTexto-Multimedia con la transición de salida TIzqTipoMultimedia-SeleccContacto, debido a que es el único par de transiciones de entrada-salida y además satisface la condición *origen(TIzqTipoMultimedia-SeleccContacto)* $\in$ *Alc(Multimedia, dest(TDerMensTexto-Multimedia))*.

   Resultado de Eliminar y reconstruir (se exhibe sólo el área afectada)*: ver figura 17.*

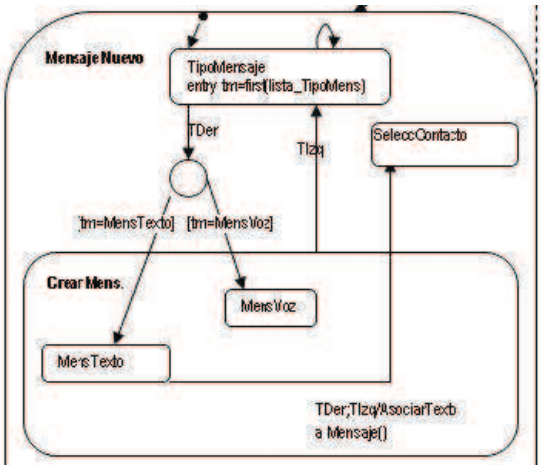


Figura 17. SC* de la figura 4 al procesar *imp*(*Multimedia*).

2. CNS = {AdmMultimedia, ElelgirSonPol, TDerTipoSonido-ElegirSonPol, TIzqElegirSonPol-TipoSonido, TDerElegirSonPol-Funcionalidad, afterTDigito2seg-PantallaPrincipal-Llamar}

Selecct afterTDigito2seg-PantallaPrincipal-Llamar ∈ CNS

TipoElementoVar(afterTDigito2seg-PantallaPrincipal-Llamar) = transición

CNS = CNS - { afterTDigito2seg-PantallaPrincipal-Llamar}

La eliminación y reconstrucción simplemente produce la baja de la transición *afterTDigito2seg-PantallaPrincipal-Llamar*, en consecuencia el TM carecerá de la funcionalidad *Marcado Rápido.* Recordar que por razones de simplicidad hemos omitido ciertas áreas del SC* que podrián relacionarse en particular con la funcionalidad *Marcado Rápido*, por ejemplo la asociación del dígito con el contacto.

3. CNS = {AdmMultimedia, ElelgirSonPol, TDerTipoSonido-ElegirSonPol, TIzqElegirSonPol-TipoSonido, TDerElegirSonPol-Funcionalidad}

Selecct ElelgirSonPol ∈ CNS

TipoElementoVar(ElelgirSonPol) = estado_simple

CNS = CNS – ({ElegirSonPol} ∪ {TDerTipoSonido-ElegirSonPol, TIzqElegirSonPol-TipoSonido, TDerElegirSonPol-Funcionalidad })

Resultado de eliminar y reconstruir: ver figura 14.

4. CNS = {AdmMultimedia}

Selecct AdmMultimedia ∈ CNS

TipoElementoVar(AdmMultimedia) = estado_simple (lo asumimos como estado simple debido a que no ha sido detallado por razones de espacio y claridad)

CNS = CNS – ({AdmMultimedia} ∪ {TDerMenuOpciones- AdmMultimedia, TizqAdmMultimedia-PantallaPrincipal })

El resultado de eliminar y reconstruir podemos visualizarlo, por simplicidad, directamente sobre el SC obtenido de la figura 18.

5. CNS = ∅

Finalmente se convierten los componentes opcionales remanentes en elementos del SC resultado, según se detalla en la figura 18.

Observación: Es interesante destacar que el SC obtenido es independiente del orden en que se eligen los componentes a eliminar. Si en lugar de elegir el estado Multimedia seleccionáramos primero el estado SeleccSonPol, el SC* obtenido primero sería en este caso el de la figura 13. Luego, al elegir el estado Multimedia, el SC* correspondería al de la figura 17. En cualquier caso, el SC resultado es el de la figura 18.

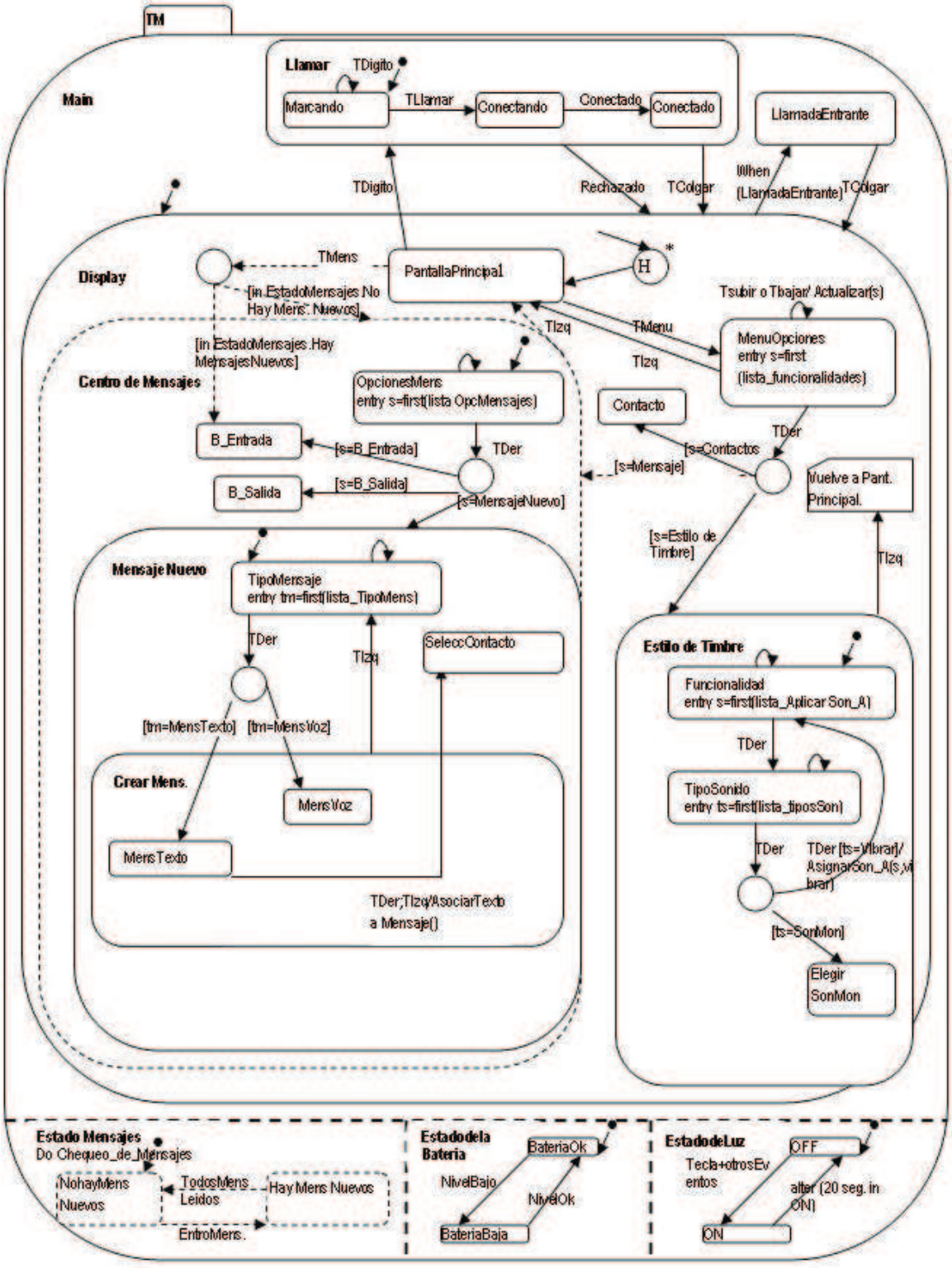


Figura 18. SC para un TM sin multimedia ni marcado rápido.

5.3 Consideraciones de Diseño

Es importante que el diseñador tenga en cuenta que a partir de un SC* pueden obtenerse SCs concretos muy variados, al considerar todas las posibilidades de instanciación factibles. Por ejemplo, la eliminación de transiciones puede producir SCs que reflejen un "mal diseño", con estados que son alcanzados en el SC pero de los cuales no es posible salir. Otra situación errónea ocurre cuando una funcionalidad no seleccionada por

una configuración es implementada por estados variantes de un SC* (a ser eliminados por la instanciación) que contienen subcomponentes (subestados o transiciones) que implementan funcionalidades seleccionadas por la configuarción. Según el algoritmo de instanciación definido, estos subcomponentes no serán incluidos en el SC resultante ya que refieren a un comportamiento dependiente de otro más general no deseado según la configuración establecida. Algo similar ocurre si una configuración de un FM selecciona una transición opcional que vincula a un estado opcional no seleccionado. En dicho caso, la transición no será parte del comportamiento del SC resultado de la instanciación, según se desprende del caso 1 de la sección 5.1.

Otro aspecto que el diseñador debe contemplar, para no producir modelos no deseados, es que los conjuntos de elementos variantes que implementan las funcionalidades de un FM no necesariamente deben ser disjuntos. En consecuencia, es posible que no se eliminen estados o transiciones de un SC* vinculados a funcionalidades no seleccionadas por una configuración del FM, si dichos componentes implementan comportamientos necesarios del producto o sistema deseado.

Respecto a las transiciones de un SC*, queremos destacar algunas consideraciones específicas. Es una decisión de diseño elegir entre transiciones opcionales o no opcionales vinculadas a un estado variante. En el primer caso, al no ser elegido el estado por una configuración, no se conservará el comportamiento de las transiciones en el SC resultante. No obstante, lo contrario ocurrirá si hay transiciones de entrada y salida no opcionales, por efecto de la generación de nuevas transiciones compuestas, según se describió en el caso 1 de la sección 5.1.

Por otra parte, debe evitarse la definición de elementos opcionales del SC* que no formen parte de ninguna implementación de funcionalidad del FM correspondiente. Si bien este último caso no influye en un mal diseño, ya que según el algoritmo presentado éstos pasarían a ser elementos del SC resultante, tampoco cumplen el rol de elementos opcionales del SC*.

El responsable de definir la función *Imp* y configurar el FM deberá atender las consideraciones mencionadas y otras que puedan producir productos o sistemas no deseados como consecuencia, en general, de una subespecificación de la PL.

6 Trabajos Relacionados

Existen múltiples trabajos que proponen la incorporación de variabilidades en sistemas de software y en particular en PLs. Entre ellos el diseñado por Jacobson [JGJ97], cuyas debilidades han sido analizadas por varios autores, por ejemplo en [LGL07]. Las soluciones más recientes pasan por modificar o extender los modelos utilizados. Distintos autores han propuesto representar explícitamente los puntos de variación añadiendo anotaciones o cambiando las bases de los diagramas de casos de uso. Por ejemplo, Von der Maßen [vdML02] plantea utilizar una notación gráfica con nuevas relaciones. John y Muthig [JM02] sugieren la aplicación de plantillas de casos de uso, aunque no distinguen entre variantes opcionales, alternativas u obligatorias. En cambio Halman y Pohl [HP03] defienden la modificación de los modelos de casos de uso para representar de forma gráfica los puntos de variación. En [LGL07] los autores proponen utilizar el mecanismo de combinación de paquetes (“package merge”) de UML 2.0, basados en [ZDD06], como herramienta de representación y configuración de la variabilidad de una línea de productos y reservar los mecanismos clásicos de modelado para expresar las variantes válidas en tiempo de ejecución de cada aplicación concreta.

En cuanto a los modelos estructurales, o bien se utilizan directamente los mecanismos de UML como en el caso mencionado de Jacobson, o bien se anotan de forma explícita mediante estereotipos. Los trabajos de Gomaa [Gom04] y Clauß [Cla01] son ejemplos de este último enfoque.

Czarnecki propone en [CA05] y [CP06] anotar los diagramas de UML con condiciones de presencia expresadas por ejemplo mediante códigos de colores, de manera que cada característica opcional se vea reflejada en una o en varias partes de un diagrama. En principio esta técnica de representación podría aplicarse a diferentes diagramas de UML, aunque en el trabajo referido se analizan solamente diagramas de clases. Una ventaja de este encare es que no se limita de forma artificial la representación de una variante a un único elemento e incluso el código de colores ayuda visualizar las implicaciones de elegir cierta variante. Sin embargo esta ayuda visual es muy poco escalable sobre el número de variantes y requiere introducir nuevos elementos auxiliares en el meta-modelo de UML.

Nuestra solución se diferencia del resto en que se centra en un modelo de especificación del comportamiento de una PL, como lo son los SCs, con un sustento formal claramente definido. Asimismo, para definir PLs y caracterizar a los distintos productos de las mismas usamos FMs, que admiten una definición formal y nos permiten configurar las características funcionales (aunque podría extenderse a no funcionales) de una línea. La variabilidad es introducida en los SCs distinguiendo tanto estados como

transiciones opcionales y no opcionales. Una PL se especifica con un SC*, un FM y una relación formal de vinculación (implementación) entre ambos modelos, que es usada para permitirnos describir el comportamiento de cada producto de la línea, a través de un algoritmo basado en casos/reglas. Un enfoque alternativo a éste está siendo desarrollado por integrantes del mismo proyecto de investigación en el cual se enmarca este trabajo. En [SV08] los autores, en un marco formal, definen funciones que asocian SCs (no componentes de SCs, como en nuestro caso) a funcionalidades de un FM y analizan formas de combinación entre distintos SCs que especifican posibles variantes de una PL. Mientras que bajo nuestro encare el comportamiento de un producto dentro de una PL se obtiene básicamente por un proceso de selección, en [SV08] el enfoque se orienta a un proceso de combinación de SCs.

7 Conclusiones y Trabajos Futuros

MDD es un enfoque con el potencial de hacer más eficiente el desarrollo y más confiables los resultados del mismo, ya que, entre otras cosas, habilita la verificación de los sistemas en etapas tempranas del desarrollo, ofreciendo mayor control. Gran parte de las técnicas de MDD utilizan UML, lenguaje incorporado como estándar de facto a nivel académico e industrial, que permite la descripción de múltiples aspectos de un sistema. En particular, los SCs de UML constituyen un mecanismo para especificar el comportamiento de sistemas mediante una representación gráfica. Estos diagramas son compactos, expresivos y proveen la capacidad de modelar no sólo sistemas simples sino también complejos sistemas reactivos.

En este trabajo presentamos una extensión de los SCs de UML con el uso de variabilidades en sus componentes esenciales para especificar PLs. Usamos FMs para describir las funcionalidades comunes y las variantes, de forma tal que a partir de distintas configuraciones de un FM se pueden generar SCs para los diferentes productos de la línea, a partir de un algoritmo basado en reglas. En el artículo desarrollamos ejemplos parciales de un caso de estudio referido a tecnología de telefonía móvil, cuya versión completa no se incluye en este artículo por razones de espacio.

Si bien existen extensiones de algunos modelos de UML para la especificación de variabilidades (entre otros trabajos, [CGW06, Gac01, GS02, Gom04, LLR06, GBS01, ZHJ04]), no se habían definido mecanismos al inicio de esta investigación, al menos para nuestro conocimiento, para la especificación de variabilidades en los SCs, que juegan un rol central en la fase de diseño de software. En la más reciente versión de UML, los SCs no ofrecen operadores y/o sublenguajes para la especificación de familias de sistemas. Dada la relevancia de UML confiamos en que los resultados de este trabajo puedan ser aplicados en problemas reales de la industria del software. Se trata pues de una contribución muy oportuna y en el nivel de abstracción adecuado para un auténtico problema actual. Es importante señalar que dentro del proyecto de investigación en el cual se enmarca este trabajo se está desarrollando en paralelo un enfoque alternativo al presentado, según se describió en la sección 6.

Como trabajos futuros estamos interesados en una extensión de los SCs que nos permita abarcar íntegramente a las máquinas de estados de UML 2.0 y analizar variabilidades en todos sus componentes, no sólo en los considerados en este artículo. Asimismo, nos proponemos dotar de una semántica formal la extensión propuesta. Esta semántica es un paso previo imprescindible para la generación automática de código y la validación de sistemas complejos de software. Finalmente, pretendemos comparar formalmente los dos encares abordados por integrantes del proyecto de investigación marco de este trabajo, buscando complementar y enriquecer ambas líneas.

Agradecimientos

A la Dra. María Victoria Cengarle (Lehrstuhl IV Software & Systems Engineering, Institut für Informatik, Technische Universität München) por el planteo original de las ideas en las que se basa este trabajo y a la Dra. Nora Szasz (Universidad ORT, Uruguay), responsable del proyecto en el cual se enmarca esta investigación.

Referencias

[Bos00] J. Bosch. “Design & Use of Software Architectures. Adopting and Evolving a Product-Line Approach”. Addison-Wesley. 2000. Chung, L., Nixon, B., Yu, E. and Mylopoulos, J. “Non-Functional Requirements in Software Engineering” Kluwer Academic Publishers 2000.

[CA05] K Czarnecki, M. Antkiewicz, Mapping Features to models: a template approach based on superimposed variants, In proc. International Conference on Generative Programming and Component Engineering (GPCE’05), LNCS 3676, Springer, pp. 422-437.

[CE00] K. Czarnecki and U.W. Eisenecker. Generative Programming: Methods, Techniques, and Applications. Addison–Wesley, 2000.

[CGW06] M. V. Cengarle, P. Graubmann, and S. Wagner. Semantics of UML 2.0 Interactions with Variabilities. Electronic Notes in Theoretical Computer Science -- International Workshop on Formal Aspects of Component Software (FACS'05, Proceedings), 160:141-155, 2006.

[Cla01] M. Clauß. Generic modeling using Uml extensions for variability. In Workshop on Domain Specific Visual Languages at OOPSLA 2001, 2001.

[CN02] P. Clements, L. Northrop. “Software Product Lines: Practices and Patterns”. SEI Series in Software Engineering, Addison-Wesley. 2002.

[CP06] Czarnecki, K., and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In Proceedings of ACM SIGSOFT/SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE'06), ACM Press, 2006.

[Gac01] C. Gacek. Implementing product line variabilities. Proc. of the 2001 Symposium on Software Reusability: putting software reuse in context, pages: 109 – 117, Canada, 2001.

[GBLM04] B. González-Baixauli, J.C.S.P. Leite, and J. Mylopoulos. “Visual Variability Analysis with Goal Models”. Proc. of the RE’2004. Sept. 2004. Kyoto, Japan. IEEE Computer Society, 2004. pp: 198-207.

[GBS01] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA’01), IEEE Computer Society, pages 45–54, 2001.

[Gom04] H. Gomaa. Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures. The Addison-Wesley Object Technology Series, 2004.

[GS02] H. Gomaa and M. Shin. Multiple-view meta-modeling of software product lines. In Proc. of the Eighth IEEE International Conference on Engineering of Complex Computer Systems, pages 238- 246, 2002.

[Har87] D. Harel: Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8: 231-274,1987.

[HP03] G. Halmans, K. Pohl. “Communicating the Variability of a Software-Product Family to Customers”. Journal of Software and Systems Modeling 2, 1 2003, 15--36.

[JEJ04] Y. Jin, R. Esser, and J. Janneck. A method for describing the syntax and semantics of UML statecharts. Software and Systems Modeling, 3(2):150-163, 2004.

[JGJ97] I. Jacobson, M. Griss and P. Jonsson . “Software Reuse. Architecture, Process and Organization for Business Success”. ACM Press. Addison Wesley Longman. 1997.

[JM02] John, I., Muthig, D.: Tailoring Use Cases for Product Line Modeling. Proceedings of the International Workshop on Requirements Engineering for Product Lines 2002 (REPL’02). Technical Report: ALR-2002-033, AVAYA labs, 2002.

[KLD02] K. Kang, J. Lee, and P. Donohoe. Feature–Oriented Product Line Engineering. IEEE Software, 19(4):58–65, 2002.

[Lau06] S. Lau. “Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates”, MASc Thesis, ECE Department, University of Waterloo, Canada, 2006.

[LGL07] M. Laguna, B. González-Baixauli, O. López. Gestión de la Variabilidad en Líneas de Productos. In Proc of CLEI´07, Costa Rica, Octubre de 2007.

[LGLG03] M. Laguna, B. González, O. López, F. García, “Introducing Systematic Reuse in Mainstream Software Process”, IEEE Proocedings of EUROMICRO'2003, Antalya, Turkey, 2003, pp: 351-358.

[LLR06] J. Liu, R. Lutz, and H. Rajan, The Role of Aspects in Modeling Product Line Variabilities. In GPCE 2006 Workshop on Aspect-Oriented Product Line Engineering (AOPLE), 2006.

[LMM99] D. Latella, I. Majzik, and M. Massink: Towards a formal operational semantics of UML statechart diagrams. In Formal Methods for Open Object-based Distributed Systems Chapman & Hall,1999.

[MCF03] S. Mellor, A. Clark, and T. Futagami. Model-driven development. IEEE Software, 20(5):14-18, 2003.

[OMG04] Object Management Group: OMG Unified Modeling Language Specification Version 2.0, 2004. Available at http://www.uml.org.

[Sel03] B. Selic. The pragmatics of model-driven development. IEEE Software, 20(5):19-25, 2003.

[SLG07] N. Szasz, C. Luna y Ariel Gonzalez. Hacia una Formalización de Líneas de Productos mediante Máquinas de Estados con Variabilidades. In Proc of CLEI´07, Costa Rica, Octubre de 2007.

[SV08] N. Szasz and P. Vilanova. Statecharts and Variabilities. To appear on proc. Second International Workshop on Variability Modelling of Software-intensive Systems, Essen, Germany, January 2008.

[vdB02] M. von der Beeck. A structured operational semantics for UML-statecharts. Software and System Modeling, 1(2):130-141, 2002.

[vdML02] T. von der Maßen, H. Lichter. Modeling Variability by UML Use Case Diagrams. Proceedings of the International Workshop on Requirements Engineering for Product Lines 2002 (REPL’02). Technical Report: ALR-2002-033, AVAYA labs, 2002.

[ZDD06] A. Zito, Z. Diskin, and J. Dingel. Package merge in UML 2: Practice vs. theory? In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, MoDELS/UML 2006, October 2006.

[ZHJ04] T. Ziadi, L. Hélouët, Jean-Marc. Jézéquel. Behaviors Generation From Product Lines Requirements. In Proc. of UML2004 Workshop on Software Architecture Description & UML, 2004.