Verificación de Propiedades Temporales en PPML

Germán Regis y Nazareno Aguirre
Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto
Ruta 36 Km. 601, Río Cuarto (5800)
Córdoba, Argentina
{gregis,naguirre}@dc.exa.unrc.edu.ar

Abstract

Product Process Modeling Languaje(PPML) is a formal language for the specification of Business Processes, it has a formal semantics based on timed transition systems. The language has artifacts that make it suitable for the formal specification of Business Processes with temporal restrictions, concurrency, etc. . Nevertheless, there is no support tool for the language. Particulary, the language lacks tools for the verification of temporal properties associated to specifications. In this paper we propose, first, a codification of the PPML semantics into timed automatas through a translation from PPML to the language UPPAAL. Second, we use this translation, that was automated in a prototype, in order to verify CTL (branching time) propierties of the PPML specifications, using the UPPAAL asociated tool.

Keywords: Formal Methods, Model Checking, Timed Automatas, Business Processes

Resumen

Product Process Modeling Languaje(PPML) es un lenguaje formal para modelar Procesos de Negocios que posee una semántica basada en sistemas de transición de estados temporizados. El lenguaje posee elementos que lo hacen apropiado para la especificación formal de procesos de negocios con restricciones temporales, concurrencia, etc. Sin embargo, no existe actualmente ninguna herramienta de soporte al lenguaje; en particular, el lenguaje carece de herramientas de verificación de propiedades temporales asociadas a las especificaciones. En este trabajo proponemos, en primer lugar, una codificación de la semántica de PPML en autómatas temporizados, a través de una traducción de PPML al lenguaje UPPAAL. En segundo lugar, aprovechamos esta traducción, que ha sido automatizada en un prototipo, para realizar verificación de propiedades CTL (*branching time*) de especificaciones PPML, utilizando la herramienta asociada a UPPAAL.

Palabras Clave: Métodos Formales, Model Checking, Autómatas Temporizados, Procesos de Negocios

1 Introducción

El constante esfuerzo de distintas organizaciones por el perfeccionamiento de sus procesos en busca de eficiencia y control, ha impulsado el actual auge de diversos métodos y lenguajes para el modelado de procesos de negocios. Si bien existen diversos lenguajes como *Business Process Execution Language*(BPEL) entre otros, en su gran mayoría están orientados a servicios y en general no proveen una semántica formal que nos permitan realizar análisis formales. PPML [9] es un lenguaje para el modelado de procesos de negocios con una semántica formal basada en sistemas de transición de estados temporizados [8]. Sus modelos se construyen mediante la especificación y composición de *procesos* y su interacción con los *productos* que manipulan. PPML ofrece una variedad de elementos, en particular la asignación de cotas temporales a los procesos, que lo hacen adecuado para la especificación formal de procesos en los cuales existan restricciones temporales, sincronización entre componentes, concurrencia en las tareas, etc.. Pese a las ventajas mencionadas, aún no existe ninguna herramienta que asista la creación de modelos y que permita obtener información de propiedades que estos cumplan.

En los últimos años, se han conseguido significativos avances en el desarrollo de técnicas y herramientas para la verificación de requerimientos, una de las más conocidas es *Model checking*[4] cuya combinación con el uso de lenguajes formales nos permiten automatizar el procesos de verificación. Entre las herramientas más utilizadas de *Model checking* para sistemas de tiempo real, se pueden mencionar Kronos [6] y Uppaal [1]. Uppaal está basada en la teoría de autómatas temporizados y provee un subconjunto de Computational Tree Logic(CTL)[10, 11, 3] como lenguaje de especificación de cosultas.

En el presente trabajo presentamos la utilización de Uppaal para la verificación de propiedades temporales de modelos especificados en PPML. Para ello se propone una traducción de los modelos al lenguaje que utiliza Uppaal y así poder consultar la validés de dichas propiedades. Como caso de estudio mostraremos una versión simplificada de una línea de producción de una compañía de MotherBoards.

Primeramente se daremos una descripción del lenguaje PPML, una introducción a Uppaal y luego mostraremos la traducción propuesta junto con las propiedades verificadas. Para finalizar mencionaremos el trabajo presente y futuras extensiones del mismo.

2 Descripción de PPML

PPML es un leguaje formal para modelar Procesos de Negocios, está basado en el método descripto en [12]. El Método consta tres tipos de construcciones básicas: *Productos*, *Procesos* y *Puertas*. Los *Productos* son entidades representadas por un conjunto de atributos. Los *Procesos* son entidades que pueden manipular, transformar productos. Éstos son caraterizados por su comportamiento en el tiempo teniendo como resticción, sólo un producto de entrada y uno de salida. Esta restricción revela la necesidad de poder empaquetar y desmpaquetar productos mediante *Puertas*.

2.1 Productos

Los *Productos* capturan las características relevantes del modelo empírico, por ejemplo altura, peso, color, etc. Estas características pueden ser directamente observables o calculables a partir de otras mediante funciones, además son determinadas en base a una escala apropiada.

Definimos Productos como: **atómicos**, **compuestos** o **estructurados**.

- Un **Producto Atómico** P es una 5-upla $< codigo_P$, $nombre_P$, $tiempo_P$, $atributos_P, < \Sigma_P, A_P >>$ donde: $codigo_P$ es una constnate del conjunto $Codigo$, distingue unívocamente cada producto, $nombre_P$ es una constante del conjunto $Nombre$, $tiempo_P$ es una constnate del conjunto $Time$ (*time stamp* del producto), $atributos_P$ es un conjunto de constantes de Σ_P, representando las caratecrísticas que queremos modelar del referente empírico. Los atributos pueden ser *directos* o *derivados*, éstos últimos obtienen su valor a partir de otros por medio de *axiomas* A_P. $< \Sigma_P, A_P >$ es la teoría de presentación, el *proto-producto* que consta de todas las constantes definidas. La igualdad de productos queda determinada por al igualdad de la especificación del *proto-producto*.

- Un **Producto Compuesto** P es : o bien, un par $P =< codigo_P, \otimes(P_{c1}, .., P_{cn}) >$ donde $\otimes(P_{c1}, .., P_{cn})$ es una inyección de los componentes P_{ci} en el producto cartesiano (usado para sincronizar y empaquetar varios productos como entrada de un proceso), o bien, un par $P = codigo_P, \iota(\mathcal{P})$ donde $\mathcal{P}$ es un conjunto finito de productos y $\iota(\mathcal{P}) \in \mathcal{P}$ (utilizado en situaciones donde el producto de entrada es elegido en base a una condición). Los productos compuestos no tienen nombres y tiempos ya que son una construcción del método y no representan referentes empíricos.

- Un **Producto Estructurado** P puede ser: **Especializado o refinado**, se extiende uno definido agregando nuevas constantes, funciones y relaciones. **Agregado**, permite combinar varios productos como constituyentes de un nuevo producto. Los atributos de los productos agregados pueden ser atributos de sus constituyentes o nuevos atributos y su proto-producto es una especialización de la unión disjunta de los proto-produtos que lo conforman.

Como ejemplo, supongamos que queremos modelar una versión simplificada de una línea de producción de una compañía de MothoerBoards; en la cual tenemos un producto *MotherBoard* con un slot para un procesador y un slot para un banco de memoria y como producto terminado la versión ensamblada y testeada de los tres componentes. Por cuestiones de espacio sólo se mostrará la especificación de algunas partes del modelo.

```
Product MotherBoard <
   <MProccessor: Proccessor;  MMemory:Memory> ,
   weight ,
   HasProccessor ,
   Proccessor_Socket ,
   HasMemory >
```

Figura 1: Producto estructurado *MotherBoard*

Podemos observar que las precedentes definiciones representan los que en Orientación a Objetos llamamos *clases*. Así llamaremos *instancias* a los referentes empíricos que se corresponden con dichas clases; estas *instancias* de productos son las tendrán lugar en la ejecución de los procesos. Una **instancia** de un producto $P =< c, n, t, att, T >$ es un una interpretación en el sentido lógico de T.

En el mundo real existen situaciones en las que un producto va evolucionando en la medida en que es procesado, es decir, existen situaciones en las que se desconocen ciertos atributos del mismo; o en caso de ser un producto estructurado, situaciones en las cuales aún no se le ha asociado alguno de sus componentes. Para ello por cada dominio necesitamos definir valores que se correspondan a estas situalciones (*valores nulos*).

2.2 Procesos

Un proceso modela un referente empírico que transforma un producto de entrada en uno de salida. Al igual que los productos, los procesos pueden ser **atómicos** (sin estructura interna o subprocesos) o **estructurados**. Los procesos modelan transformaciones, para ello, por cada proceso existe una *máquina virtal* que interpreta sus comandos básicos. Esta máquina virtual es un objeto con métodos, por ejemplo, asignaciónes a variables. Para especificar un proceso, definimos las transformaciones éste realiza usando acciones básicas o combinaciones de estas por medio de estructuras de control. La formalización del concepto de *máquina virtual* [7] consiste en un *framework lógico* sobre sistemas de transición de estados llamado RETOOL [2]. La especificación es un par que contiene una signatura y una conjunto de sentencias que describen su comportameinto.

La definición de un proceso es una *transacción*, un segmento de computación de la máquina virtual subyacente. Consta de la *condición inicial* q (estados que satisfacen las condiciones iniciales), la *condición final* p (estados que satisfacen la condición final de la transacción), el *Invariante* I (sentencia que debe evaluarse verdadera en todos los estados de la traza) y las *cotas inferior y superior* l, u que especifican el tiempo mínimo y máximo que puede demorar en completarse. Formalmente $(q, I)_l\Delta^u p$, y su semántica: $\sigma, T, i \vDash (q, I)_l\Delta^u p$ si y sólo si para algún j, k tal que $i + 1 \leq j < k \leq i + u$, tenemos para cada $i \leq m \leq j$ que $\sigma, T, m \vDash q$; $\sigma, T, k \vDash p$, y para cada $j \leq n \leq k$ tenemos $\sigma, T, n \vDash I$.

Un **Proceso Atómico** p es un par $< proc, VM >$, donde $VM =< \theta, \Theta >$ y $proc =< codigo$, $nombre$, $propietario$, $P_I, P_O, (q, I)_l\Delta^u p >$. $codigo$, $nombre$, $propietario$ son variables de los correspondientes conjuntos y son fijos, es decir, no cambian durante trazas de cómputo. P_I, P_O son los productos de entrada y salida, y $(q, I)_l\Delta^u p$ es la especificación del comportamiento del proceso.

Como ejemplo se muestra el proceso que toma una MotherBoard sin ensamblar y un Procesador y devuelve la MotherBoard con el procesador ensamblado. El producto de entrada será un paquete con ambos productos de entrada. Como condición inicial especificamos la compatibilidad de los socket de conexión y que la MotherBoard no tenga asemblado el Procesador. El invariante requiere que no puede ser asemblado simultaneamente la memoria, y como producto de salida retorna la MotherBoard con el Procesador ensamblados. Además especificamos que el proceso tomará como mínimo 5 unidades de tiempo y 10 como máximo.

```
Process assem_1 <
  input: ⊗(<Mother_in:Motherboard;Proc_in:Proccessor>)
  output: MotherBoard
  invariant: Mother_in.HasMemory = false
  requires: Mother_in.Proccessor_Socket = Proc_in.Proccessor_Socket
  ∧ Mother_in.HasProccessor=false
  ensure: output=<<Mother_in,Proc_in,item> , true, false>
  l_time: 5 , u_time:10 >
```

Figura 2: Proceso que ensambla un Procesador en una Motherboard

2.3 Puertas

Usaremos el concepto de **puertas** para modelar construcciones que necesitamos en el lenguaje, por ejemplo, cuando un proceso necesita varios productos o para distribuir varios productos como entrada a procesos o para organizar *sincronizar* el control del sistema cuando cierta condición debe ser cumplidad para poder continuar. Exiten tres tipos básicos de puertas:

- **Multiplexor** es una 4-upla $< codigo,\ \mathcal{P},\ P,\ F >$ donde *codigo* es una variable estática que la identifica, $\mathcal{P}$ es el conjunto de productos de entrada, P es el producto de salida y F es la función que define cada P_i explícitamente en terminos de $\mathcal{P}$.

- **Demultiplexor** , definición simétrica a la anterior, en este caso los productos de salida son proyecciones sobre producto de entrada.

- **Semáforo** es una 3-upla $< codigo, P, S, >$ donde P es el producto de entrada/salida y S es la condición que debe satisfacerse para continuar. Gráficamente:

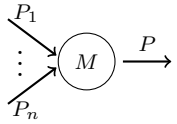


Figura 3: Multiplexer

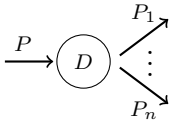


Figura 4: Demultiplexer

Figura 5: Semaphore

2.4 Frameworks

En el proceso de modelar sistemas, al igual que en programación, necesitamos *abstracción* y *encapsulamiento* como herramientas. PPML provee estructuras de procesos **framework process** a través de las diferentes formas de combinar procesos:

- **Composición Sequencial**: compone dos procesos p_1, p_2 (denotado $p_1; p_2$) en uno nuevo p de la siguiente manera: el *codigo* y *nombre* de p tienen un nuevo y único *codigo* y *nombre* respectivamente, el *propietario* será la entidad responsable de p. La entrada de p será la de p_1 y su salida la de p_2. La función de transición τ es $(q_p, I_p)_{l_p}\Delta^{u_p}p_p$ donde: l_p es la *cota inferior* del primer proceso, u_p es la suma de las *cotas superiores*, q_p es q_1, p_p es p_2 y I_p es $q_1\mathcal{U}_{>l_{p_1}}(I_{p_1}\mathcal{U}_{\leq u_{p_1}-l_{p_1}}(p_{p_1} \rightarrow true\mathcal{U}(q_{p_2}\mathcal{U}_{>l_{p_2}}(I_{p_2}\mathcal{U}_{\leq u_{p_2}-l_{p_2}}p_{p_2}))))$

  El *invariante* requiere que la condición inicial del primer proceso se mantenga almenos hasta l_{p_1}, a partir de ese punto el invariante de p_1 debe ser cierto durante $u_{p_1} - lp_1$ unidades de tiempo, cuando la condición final p_{p_1} se torna verdadera. Eventualemente la condicion inicial q_{p_2} será verdadera almentos hasta l_{p_2}, en ese momento comezará la validéz de la invarte I_{p_2} durante un tiempo no superior a $u_{p_2} - l_{p_2}$ justo en el punto en el cual se cumple la condición final p_{p_2}. Cabe destacar que deben concordar las especificaciones de los produtos de salida de p_1 y entrada p_2. El tiempo entre el estado final de p_1 y el inicial de p_2 no es tenido en cuenta en el modelo, en caso de querer reflejar cierto tiempo que manifieste el referente empírico, debeíamos componer otro proceso intermedio que lo refleje. Denotamos la composición secuencial como $p_1; p_2$.

- **Composición Condicional (Semáforos)**: Denotado por $p_1;_s p_2$ es el proceso $p =< codigo, nombre, propietario, P_i, P_o, \tau >$ con una definición análoga a la anterior, y cuyo invariante está dado por:
  $q_1\mathcal{U}_{>l_{p_1}}(I_{p_1}\mathcal{U}_{\leq u_{p_1}-l_{p_1}}(p_{p_1} \rightarrow true\mathcal{U}((q_{p_2} \wedge S_s)\mathcal{U}_{>l_{p_2}}(I_{p_2}\mathcal{U}_{\leq u_{p_2}-l_{p_2}}p_{p_2}))))$.

- **Composición Paralela**: Denotada por $[p_1; p_2](d_I, m_O)$ es: *codigo*, *nombre* y *propietario* análogos a las definiciones anteriores; la entrada de p es la del demultiplexor d_I, la salida

es la del multiplexor m_O y la función τ de p es $(q_p, I_p)_{l_p}\Delta^{u_p}p_p$: l_p es *máximo* (l_{p_1}, l_{p_1}), u_p es *máximo* (l_{p_1}, l_{p_1}), q_p es $q_1 \wedge q_2$,
p_p es $p_1 \wedge p_2$, y I_p es $(q_1 \wedge q_2) \rightarrow (\tau_1[(p_1\mathcal{U}(p_1 \wedge p_2))/p_1] \wedge (\tau_2[(p_2\mathcal{U}(p_1 \wedge p_2))/p_2]))$ (donde $\tau[(p\mathcal{U}q)/r]$ significa el reemplazo de la condición final r en τ por la condición $(p\mathcal{U}q)$).

Un *framework process* p es un Proceso **atómico** definido a través de un conjunto de procesos $\{p_1, ..., p_n\}$ que lo forman en términos de las distintas composiciones definidas anteriomente y lo denotamos con el par $< p, fw.exp >$, donde p es la difinición clásica de proceso y $fw.exp$ es la especificación de los procesos que lo forman en términos de composiciones entre ellos.

3 Uppaal

Uppaal es una herramienta para la verificación de sistemas de tiempo real. Está basada en la teoría de autómatas temporizados y provee un subconjunto de CTL (Computational Tree Logic) como lenguaje de especificación de cosultas. Un modelo de un sistema en Uppaal es un conjunto de instancias de esquemas que se comunican por medio de distintos tipos de canales. La especificación consta de tres partes: *Declaraciones Globales*, *Esquemas* (Automata Templates) con sus *Declaraciones Locales* y la *Especificación del Sistema*.

3.1 Declaraciones

Las declaraciones globales, o locales a un esquema, contienen definiciones de variables, arreglos, registros y tipos (estilo lenguaje C). Existen cuatro tipos predefinidos: int (enteros), bool (lógicos), clock (relojes), and chan (canales); éstos ultimos puden ser *canales urgent* o *broadcast*. También pueden definirse constantes anteponiendo el prefijo *const*. Uppaal provee un lenguaje rico para la declaración de funciones que pueden ser invocadas dentro de los los esquemas, podemos especificar parámetros, sentencias condicionales, sentencias iterativas como *While* y *For*.

3.2 Esquemas

Los Esquemas son definidos en forma de *Autómatas Temporizados Extendidos*; pudiendo en ellos leer y modificar relojes, variables, invocar a funciones definidas, etc. El autómata de un esquema cosiste en *Estados* y *Aristas*, también pueden contener declaraciones locales y *parámetros* por *valor* o *referencia*.

Los **Estados** pueden estar *etiquetados* (nombres que los referencian). También podemos especificar *invariantes* en los estados, esto es, una condición que siempre debe cumplirse. Las expresiones permitidas como invariantes son restringuidas, sólo pueden ser una conjunción de condiciones simples sobre relojes o expresiones lógicas que no contengan relojes. No se permiten condiciones que involucren cotas inferiores sobre relojes. Existen tres modificadores para los estados de esquemas: *Inicial*, cada esquema debe tener exactamente un estado inicial. *Urgentes*, detienen el paso del tiempo mientras un proceso se encuetre en uno de ellos. *Comprometidos*, al igual que los urgentes detienen el paso del tiempo, además obliga que la próxima transición debe tomar alguna arista que comienze en un estado comprometido habilitado.

Los estados están conectados por medio de **Aristas**, en ellas se pueden definir: *selectores*, *guardas*, *sincronizaciones* y *modificaciones*. Los selectores retornan aleatoreamente un valor de un rango especificado. Una arista esta habilitada si y sólo si la condición de su guarda se cumple,

las condiciones de las guardas tienen las mismas restricciones que las de los invariantes. Los procesos pueden sincronizarse compartiendo un canal, en el cual uno escribe (!) y el otro escucha (?), sólo se permite un canal por arista. Cuando las aristas son ejecutadas, las expresiones de modificaciones son evaluadas y su efecto cambia el estado del sistema.

Cuando dos procesos son sincronizados, ambas aristas de los procesos son ejecutadas. Las modificaciones de ambos se realizan se realiza en orden, primero evaluando las del proceso que escribe y luego las del que escucha. Los canales Broadcast perminten sincronizacioes de 1-a-muchos, esto es, un proceso escribe en un canal y todos aquellos con aristas habilitadas en escucha de dicho canal serán invocados.

3.3 Especificación del Sistema

La Especificación de un modelo de sistema consiste de uno o mas procesos concurrentes (instancias de esquemas), variables y canales de comunicación. Las variables, canales y funciones definidas aquí no son visibles para los esquemas.

3.4 Propiedades Temporales

Uppaal soporta como lenguaje de consultas, lógica temporal CTL [ref] con ciertas restricciones, sólo pude haber un cuantificador de caminos. Así las consultas que podemos realizar son de la forma:

- $E \diamond q$: se evalua como verdadera si y sólo si existe una secuencia de transiciones $s_0 \rightarrow \cdots \rightarrow s_n$ donde s_0 es el estado inicial y en s_n satisface q.
- $A[]q$: es verdadera si y sólo si todo estado alcanzable satisface q.
- $E[]q$: es verdadera si y sólo si existe una secuencia de transiciones $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_i \rightarrow \cdots$ donde todo s_i satisface q.
- $A \diamond q$: es verdadera si y sólo si toda secuencia de transiciones posible alcanza un estado que satisface q.

Donde q es un expresión lógica bien formada, en ella podemos hacer referencia a variables o estados de los procesos (instancias de esquemas), por ejemplo $A[]$ *MotherBoard.End imply MotherBoard.hasProcessor*, especifica que para cualquier secuencia de ejecución siempre es cierto que si una instancia del esquema *MotherBoard* está en el estado *End* entonces su variable *hasProcessor* debe ser verdadera.

4 De PPML a Uppaal

Con el objetivo de probar propiedades temporales de modelos especificados en PPML, se implementó una traducción *automática* de modelos PPML al lenguaje Uppaal, así luego de traducir e invocar al *solver* de Uppal podemos verificar dichas propiedades. La idea general de traducción consiste en: Por cada clase de *producto* generar un esquema que describe sus estados dentro del sistema. Por cada *proceso* se construye un esquema que describe su comportamiento. La semántica de las puertas se incluyen dentro de los esquemas de productos y procesos. La comunicación entre productos y procesos se realiza mediante variables compartidas (globales) y

mensajes de sincronización. Esta forma de traducir refleja la independencia de cada proceso y nos permite incrementar la cantidad de instancias de productos dentro del sistema de manera automática. Con el fin de ejemplificar la traducción se seguirá el siguiente caso de estudio:

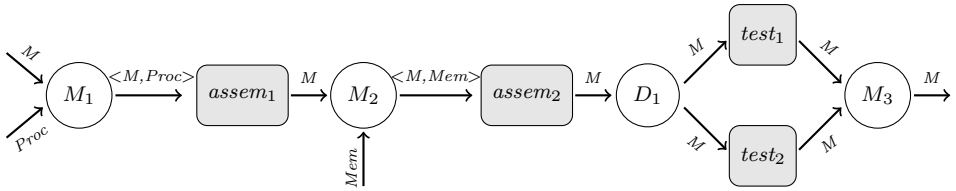


Figura 6: Diagrama PPML $assem_1;assem_2;[test_1;test_2](D_1,M_3)$

El proceso de ensamblado comienza con un proceso $assem_1$ que recibe una MotherBoard y un Procesador, los ensambla en un intervalo de entre 5 y 10 unidades de tiempo, seguidamente el proceso $assem_2$ recibe el producto de salida del proceso anterior, una Memoria y los ensambla con la misma restricción temporal. Luego se testean el Procesador y la Memoria de la MotherBoard en paralelo por los procesos $test_1$ y $test_2$ respectivamente cuyos tiempos se estiman entre 3 y 5 unidades.

4.1 Traducción de Procesos

Por cada proceso en PPML se genera:

- En las declaraciones globales del sistema, canales de mensajes para la comunicación con los producto que procesa, esto es, un canal y una variable (del tipo de código correspondiente) por cada producto que el proceso manipula. Si bien en PPML, los procesos sólo tienen un producto de entrada, éste puede ser un producto compuesto (empaquetado), esto se refleja en la traducción como la sincronización a través de los canales mencionados. También se genera un canal (*broadcast*) por el cual, el proceso avisa a los productos intervinientes que están siendo procesados.
- Un esquema con una declaración de una variable reloj (tiempo que demora el proceso en completarse), un estado inicial, los estados correspondientes a: la puerta previa (en caso de ser una puerta multimplexora, las posibles combinaciones de sincronizaciones de los productos que el proceso manipula), un estado para representar la condición "listo para comenzar" o pre-ejecución del mismo y un estado para su pos-ejecución. Éstos dos últimos estados contienen como invariante, la traducción del Invariante del proceso PPML. Las aristas que unen el estado inicial y los correspondientes a la sincronización de productos, están etiquetados con la espera por el correspondiente canal de comunicación. La arista que conecta los estados de pre y pos ejecución del proceso tiene como guarda la restricción temporal sobre la variable reloj del proceso. Finalmente un arco que reestablece las condiciones para un próximo proceso conectando el estado final con el inical.
- Dentro de la declaración del sistema, se generan una instacia por cada proceso.

Globales

```
//input sincronizations channels
urgent chan cassem2TMotherBoard;
urgent chan cassem2TMemory;
//procesing comunication channel
broadcast chan cAssem2;
//ids of the inputs products
n_motherboards0 assem2_id_motherboard;
n_memories0 assem2_id_memory;
```

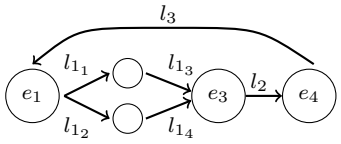
Locales al esquema

```
clock x;
```

Sistema

```
Assem20 = Assem2();
```

Estados y Aristas del esquema



e1: Estado Inicial (M_2)
$\mathbf{l}_{1_1}$: Sincr.: cassem2TMotherBoard?
$\mathbf{l}_{1_2}$: Sincr.: cassem2TMemory?
$\mathbf{l}_{1_3}$: Sincr.: cassem2TMotherBoard?
$\mathbf{l}_{1_4}$: Sincr.: cassem2TMemory?
e2: Pre-assem2
l2: Guarda: x >=3 && x<=5
Sincr.: cassem2!
e3: Pos-assem2
l3: Modif.: x =0;
assem2_id_memory = 0 ;
assem2_id_motherboard = 0;

Figura 7: Traducción del Proceso `assem2`

4.2 Traducción de Productos

Por cada producto en PPML se genera:

- Una definición de tipo subrango de entero 0..n en las declaraciones globales, que indica *n-1* instancias del producto. Estos valores serán los *códigos* o identificadores unívocos de cada producto dentro del sistema, el identificador 0 se reserva para el producto *nulo*, en caso de productos estructurados incompletos.
- Un esquema que contiene: Como **declaración local**: una variable de tipo reloj (tiempo del sistema). Una variable por cada atributo. En caso de ser estructurado, una variable (identificador) por cada uno de sus componentes. Una función por cada proceso que lo manipula, en la cual se reflejan los cambios que realiza el proceso en los atributos del producto. Un **parámetro** constante del tipo correspondiente al los identificadores de dicho producto, que es utilizado para la creación de las instancias dentro del sistema. **Estados**: un estado incial, un estado por cada proceso o puerta que lo manipula y un estado final. **Arcos**: Para simplificar la traducción y sin pérdida de generalidad asumimos que a cada proceso le precede una puerta (en caso de no existir asumimos una puerta de tipo semáforo con la condición *verdadero*). De esta manera tenemos dos tipos de arcos, los que van de un estado correspondiente a una puerta al de un proceso y viceversa. Los primeros (e_{puerta},$e_{proc.}$) están etiquetados con un mensaje de escritura por el canal correspondiente (listo para ser tratado) y la asignación de su identificador en la variable correspondiente al proceso en caso de que su valor sea 0 (no hay otro producto esperando para ser tratado). Para los arcos ($e_{proc.}$,e_{puerta}), un mensaje de espera por el canal correspondiente al proceso, la condición de que el identificador del producto que el proceso va a manipular sea el suyo y la invocación a la función que refleja los cambios producidos por el proceso en sus atributos. En caso de productos que sean copiados por demultiplexores, por ejemplo *MotherBoard*, estas bifurcaciones de procesamiento paralelo se reflejarán en el esquema como todos los posibles casos (*interleaving*) en que el producto puede ser procesado.

- En la declaración del sistema, se delaran las instancias de de los esquemas.

Globales

```
//amount of instances 2
 typedef int[1,2] n_memories;
//amount of instances 2 + null
 typedef int[0,2] n_memories0;
```

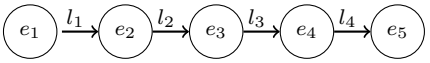
Locales al esquema

```
clock x;
//attributes
int size;
bool tested;
bool assembled;
//Processes Updates
void Assem2() assembled = true;
void Test2() tested = true;
```

Sistema

```
Memories(const n_memories id) =
Memory(id);
```

Estados y Aristas del esquema



e1: Estado Inical (M_2)
l1: Guarda: assem2_id_memory = 0;
Sincr.: cassem2TMemory!
Modif.: assem2_id_memory = id
e2: $assem_2$
l2: Guarda: assem2_id_memory == id
Sincr.: cassem2?
Modif.: Assem2()
e3: D_1
l3: Guarda: test2_id_memory = 0;
Sincr.: ctest2TMemory!
Modif.: test2_id_memory = id
e4: $test_2$
l4: Guarda: test2_id_memory == id
Sincr.: ctest2?
Modif.: Test2()
e5: Estado Final (M_3)

Figura 8: Traducción del Producto Memory

4.3 Declaración del sistema y verificación de propiedades

Finalmente se genera la declaración del sistema con todas las instancias de los esquemas de procesos y productos. Con el modelo generado podemos invocar al verificador con las propiedades que deseamos verificar.
Como parte del caso de estudio se verificaron las siguientes propiedades:

```
1) A[] forall (i:n_motherboards) forall (j:n_motherboards)
    i != j imply ((MotherBoards(i).End and MotherBoards(j).End)
      imply (MotherBoards(i).id_Memory != MotherBoards(j).id_Memory
      and MotherBoards(i).id_Processor != MotherBoards(j).id_Processor))
```

Toda MotherBoard ensamblada no puede compartir su memoria ni su procesador con otra.

```
2) A[] forall(i:n_motherboards) (MotherBoards(i).End imply
   (MotherBoards(i).id_Memory!=0 and MotherBoards(i).id_Processor!=0
   and Processors(MotherBoards(i).id_Processor).tested == true and
   Memories(MotherBoards(i).id_Memory).tested == true)
```

Toda MotherBoard que llege al final del proceso es ensamblada con una memoria y un procesador testeados.

```
3) E<> exists(i:n_motherboards)(MotherBoards(i).End and
    MotherBoards(i).x<=13)
```

Los testeos puden realizarse en paralelo reduciendo el tiempo de producción (13 es la suma de los tiempos mínimos de los procesos del modelo).

Las propiedades se verificaron en una PC con un procesador Intel Pentium 4 de 3 ghz y 2GByte memoria DDR con sistema operativo Linux, los detalles de los tiempos son:

	2 c/prod.	**3** c/prod.	**3** Moth., **4** Proc. y Mem.	**3** Moth., **5** Proc. y Mem.
1)	0,18 s	15,83 s	22,15 s	$> 3\ h$
2)	0,02 s	0,97 s	6,13 s	$> 3\ h$
3)	0,001 s	0,01 s	4,51 s	826,23 s

Nota: Los experiemtos se realizaron con la cantidad Procesadores y Memorias suficientes para la cantidad de MotherBoards, es decir, *almenos* un Procesador y una Memoria por MotherBoard; caso contrario propiedades como 2 no podían verificarse, tal como se esperaba. Con 4 MotherBoards al igual que en los casos de 3 MotherBoards, 5 Procesadores y 5 Memorias, pasadas las 3 horas de ejecutada la verificación y habiendo duplicado el espacio de memoria virtual con respecto al físico, se cancelaron las verificaciones.

5 Conclusiones y Trabajos Futuros

Hemos mostrado una traducción de modelos del lenguaje PPML al de Uppaal con el fin de poder verificar propiedades temporales de los mismos. Actualemnte se esta desarrollando una herramienta para asistir en la creación de modelos en PPML y, como extensión de la misma, la traducción automática propuesta. Creemos que el lenguaje PPML es de gran utilidad para el modelado de diversos procesos en los cuales se necesite un análisis formal. Por medio de este trabajo se puede contar con una herramienta automática que facilita la construccion de modelos de negocios y verificar en ellos propiedades de *safety*, *invariancia*, etc.
Debido al problema de la explosión de estados, se está estudiando la posibilidad de aplicar técnicas de abstracción [5] para poder explorar modelos con dominios de productos y procesos más grandes.

Bibliografía

[1] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. *UPPAAL- a tool suite for the automatic verification of real-time systems*. In Proceedings of Hybrid Systems III. LNCS 1066.pages 232-243. Spriger Verlag. 1996.

[2] S. Carvalho, J. Fiadeiro, E. Haeusler. *A Formal Approach to Real-Time Object Oriented Software*. In: Proceedings of the Workshop on Real-Time Programming. IFAP/IFIP, 1997, pp91-96, Lyon, France.

[3] E. M. Clarke and E. A. Emerson and A. P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Programming Languages and Systems, 1986, vol. 8, 244–263.

[4] E. Clarke, O. Grumberg y D. Peled, *Model Checking*, MIT Press, 2000.

[5] S. Das, D. L. Dill, S. Park. *Experience with predicate abstraction*. In 11th International Conference on Computer-Aided Verification, pages 160–172. Springer-Verlag, July 1999.

[6] C. Daws, A. Olivero, S. Tripakis, S. Yovine. *The tool KRONOS*. In Hybrid Systems III, Springer LNCS 1066, pp. 208–219, 1996.

[7] J. Fiadeiro, T. Maibaum, *Temporal Theories as Modularisation Units for Concurrent System Specification*. Formal Aspects of Computing. 1992, 4(3):239-272

[8] T. A. Henzinger, Z. Manna, A. Pnueli,*Timed Transition Systems*,1996.

[9] T. S. E. Maibaum, *An Overview of The Mensurae Language: Specifying Business Processes*, Rigorous Object-Oriented Methods, BCS, 2000.

[10] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems - Specification* -, Springer, 1991.

[11] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems - Safety* -, Springer, 1995.

[12] M. Myers, A. Kaposi, *A First Systems Book: Technology and Management*, ISBN 978-1860944321, Imperial College Press, 2 edition, 2004.