Verificación de Propiedades Temporales de Diseños CommUnity: Un Caso de Estudio

Nazareno Aguirre y Germán Regis
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

Resumen

En este trabajo, exploramos la utilización de una lógica temporal lineal de primer orden para la verificación de propiedades de ejecuciones de programas abstractos, escritos en un lenguaje de diseño de programas llamado CommUnity. CommUnity es un lenguaje cuya semántica y mecanismos de composición están basados en construcciones de teoría de categorías. Desarrollamos un caso de estudio, en el cual hacemos uso de una interpretación de programas CommUnity como sistemas de transición de estados, y experimentamos con el uso del demostrador de teoremas STeP para realizar verificaciones.

Finalmente, discutimos algunas posibles formas de realizar las actividades de verificación de manera modular, aprovechando los distintos tipos de morfismos de programas disponibles en CommUnity.

1 Introducción

Debido al tamaño y complejidad de los sistemas de software modernos, la noción de arquitectura de software [7], entendida como la estructura de un sistema en términos de las componentes que lo conforman y de qué manera éstas se comunican, ha ganado considerable importancia en relación con el análisis y diseño de sistemas. Es claro que contar con una modularización adecuada de un sistema lleva a mejoras importantes, no necesariamente en términos de performance, sino en la facilidad para introducir cambios o extensiones, o para depurar, validar y documentar el sistema.

Durante las etapas tempranas de muchos procesos de desarrollo, y en particular en el ámbito de los denominados *sistemas críticos*, se hace necesario contar con mecanismos de especificación, análisis y estudio de distintos aspectos arquitectónicos de sistemas. En arquitecturas de software, se sugiere modelar sistemas a un alto nivel de abstracción, en término de componentes, las cuales representan las unidades computacionales de los sistemas, interconectadas por medio de conectores, que representan los mecanismos de interacción entre componentes. Una gran variedad de lenguajes, denominados *architecture description languages* [12], han sido propuestos especialmente para la especificación de arquitecturas de software. Algunos de éstos son Acme [6], Darwin [9], C2 [11], Rapide [8], CommUnity [13], y otros. Algunos de

estos lengaujes cuentan con herramientas de software, que permiten realizar distintos tipos de análisis, tales como simulación, chequeo de restricciones de integridad, verificación de consistencia, etc.

Estamos particularmente interesados en uno de estos lenguajes, el lenguaje CommUnity [5]. CommUnity cuenta con una semántica formal, que permite analizar rigurosamente propiedades de los sistemas especificados. La semántica de CommUnity está basada en construcciones de teoría de categorías. Más aún, los mecanismos de composición de componentes en CommUnity se definen en términos de morfismos (en el sentido de teoría de categorías), para relaciones de refinamiento, uso o extensión.

Existen algunas herramientas asociadas a CommUnity, para el análisis de especificaciones de arquitecturas de software. Sin embargo, la herramienta disponible actualmente, denominada CommUnity Workbench, no permite verificar propiedades temporales de sistemas especificados en CommUnity. En este trabajo, estudiamos la utilización de la lógica de Manna-Pnueli [10] para la especificación de propiedades temporales de programas CommUnity, y la utilización del demostrador de teoremas STeP [1] para la verificación asistida de las mismas. Presentamos un caso de estudio, basado en un modelo simple de un procesador, que incluye el planificador de atención de procesos. Mostramos su especificación en CommUnity, su interpretación como sistema de transición de estados en STeP, y la caracterización de propiedades del mismo en la lógica temporal de Manna-Pnueli. Demostramos estas propiedades combinando *model checking* [2] y deducción semi-automática en el ambiente STeP, y finalmente discutimos cómo pueden modularizarse las actividades de verificación, mediante el aprovechamiento de los morfismos entre componentes disponibles en CommUnity. Advertimos que, si bien los morfismos de refinamiento no permiten realizar refinamiento de datos, se puede aprovechar este tipo de refinamiento directamente mediante el uso de STeP. Esto permite demostrar propiedades usando técnicas de interpretación abstracta [3].

2 CommUnity

En esta sección, introducimos al lector en el lenguaje de diseño de programas CommUnity. CommUnity fue originalmente introducido en [5], y existen actualmente versiones más poderosas, que abarcan, por ejemplo, la especificación de arquitecturas de software dinámicas [13]. Introduciremos el lenguaje mediante un ejemplo. El ejemplo consiste de la especificación de un procesador, incluyendo un planificador de atención de procesos. Las unidades de cómputo en CommUnity se representan mediante *diseños*. Un diseño es básicamente una componente, con variables (o campos, o atributos) y acciones (o métodos o rutinas). Las acciones se consideran atómicas, y no se ejecutan en paralelo. Considere el programa que se muestra en la Figura 1. Este programa abstracto consta de cuatro variables, una de entrada (`PSec`), una de salida (`Curr`), y dos locales (`Buffer` y `CurrId`). La variable `PSec` representa el número de segundos de procesador que un nuevo proceso, a ser incorporado a la cola de procesos en espera, utilizará. La variable `Curr` contiene un par de enteros, donde el primero representa el identificador del proceso y el segundo representa la cantidad de segundos que restan por ejecutarse del mismo. La variable `Buffer` representa la cola de procesos en espera por ser atendidos, y `CurrId` es una variable que se utiliza para generar nuevos identificadores para procesos entrantes. La inicialización de las variable de esta componente se muestran en la sección `init`

```
Dsgn Processor
in PSec: int
out Curr: ⟨ int,int ⟩
prv Buffer: list(⟨ int,int ⟩); CurrId: int
init Curr := ⟨ 0,0 ⟩ || Buffer:= [] || CurrId := 1
do load: PSec > 0, false → Buffer:= Buffer++ [⟨CurrId,PSec⟩] ‖ CurrId ++
[] prv run: Curr.snd > 0, false → Curr := ⟨ fst(Curr),snd(Curr)-1 ⟩]
[] kill: snd(Curr) = 0 ∧ Buffer≠ [], false →
   Curr := head (Buffer) ‖ Buffer:= tail(Buffer)
[] switch: snd(Curr) > 0 ∧ Buffer≠ [], false →
   Curr := head(Buffer) ‖ Buffer:= tail(Buffer) ++ [Curr]
```

Figura 1: Un diseño CommUnity de un procesador.

del diseño.

Las acciones son asignaciones múltiples con guardas, y se asume que su ejecución es atómica. Cada acción está equipada con dos guardas, una guarda de seguridad, cuya negación bloquea la acción, y una guarda de progreso, cuya validez no puede ignorarse indefinidamente [13]. Las guardas de progreso tienen especial importancia en el refinamiento de diseños, y en la demostración de propiedades de liveness, pero nosotros no haremos uso de ellos en este artículo. El diseño `Processor` cuenta con cuatro acciones. La acción `load`, que tiene como guarda de seguridad que la variable de entrada `PSec` debe contener un valor positivo, agrega un nuevo proceso a la cola, con duración `PSec` y con el identificador corriente, el cual es además incrementado. La acción `run` ejecuta el proceso corriente durante un segundo. La acción `kill` elimina el proceso corriente, cargando un nuevo proceso de la cola. La acción `switch` cambia el proceso corriente por otro en la cola de procesos.

3 Lógica Temporal Lineal

Podemos ver a la lógica temporal lineal de primer orden como una extensión de la lógica de primer orden a la cual se le agregan los operadores temporales $\Box$, $\bigcirc$, $\mathcal{U}$ y a partir de ellos podemos obtener otros como $\Diamond$ y $\mathcal{W}$. El significado intuitivo de los mismos es el siguiente:

- $\Box\alpha$: siempre en el futuro sucede α.
- $\bigcirc\alpha$: en el siguiente estado del tiempo sucede α.
- $\alpha\mathcal{U}\beta$: α ocurre hasta que ocurre β.
- $\Diamond\alpha$: en algún instante futuro sucede α.
- $\alpha\mathcal{W}\beta$: α ocurre hasta que ocurre β, o siempre ocurre α.

Teniendo en cuenta que un programa en ejecución puede interpretarse como una secuencia de estados (es decir, una traza de ejecución es una secuencia potencialmente infinita de estados), estos nuevos operadores nos permiten especificar las propiedades que debrían tener tales ejecuciones. Por ejemplo, si deseamos saber si cierta aserción

α es un invariante de un sistema dado, deberemos verificar que todas las trazas de ejecución del mismo satisfagan la fórmula temporal:

$$\Box\alpha$$

Si, en cambio, deseamos comprobar si un sistema dado posee la propiedad de que, la ocurrencia de un evento α (representado por una fórmula) produce la ejecución eventual de un evento o acción β (representado, nuevamente, por una fórmula), deberemos verificar que todas las trazas de ejecución del sistema satisfagan la fórmula temporal:

$$\Box(\alpha \rightarrow \Diamond\beta)$$

La lógica temporal lineal de primer orden nos permite especificar eventos o acciones (α y β en los ejemplos anteriores) expresables no sólo mediante proposiciones simples, sino también utilizando cuantificación de variables individuales. Por ejemplo, supongamos que tenemos un sistema que, entre otras cosas, consta de una variable entera de programa v. Si queremos especificar el hecho de que, en toda ejecución posible del sistema, v toma todos los valores enteros posibles, podemos hacerlo de la siguiente manera:

$$\forall x \in \mathsf{int} : \Diamond(v = x)$$

Por razones de espacio, nos es imposible describir en detalle esta lógica. El lector interesado puede consultar [10].

3.1 STeP

La lógica de Manna-Pnueli cuenta con una herramienta muy potente para la verificación de sistemas concurrentes. El sistema STeP permite escribir programas concurrentes usando un lenguaje de programación concurrente simple, y luego expresar propiedades temporales de los mismos usando la lógica mencionada. El ambiente STeP combina model checking simbólico con un demostrador de teoremas para verificar propiedades de los programas. Más aún, STeP también incorpora mecanismos de decisión en el demostrador de teoremas, técnicas de generación automática de invariantes, y técnicas de interpretación abstracta [3], para realizar las tareas de verificación.

En las secciones siguientes mostraremos cómo podemos aprovechar varias de estas técnicas de verificación provistas por STeP para demostrar propiedades temporales de diseños CommUnity.

4 Interpretación de Programas CommUnity en STeP

La semántica de los programas concurrentes SPL (el lenguaje de programación asociado a STeP) está basada en sistemas de transición de estados. Los programas CommUnity comparten esta semántica, con lo cual es relativamente directa la traducción de programas CommUnity a programas SPL. La dificultad principal se encuentra en la forma de interpretar las distintas características de los diseños CommUnity. Por razones de espacio, no podemos mostrar aquí los detalles de la traducción. La traducción del diseño de la Fig. 1 se muestra en la Fig. 2. En esta figura, mostramos

directamente la interpretación del diseño CommUnity como un sistema de transición de estados, en lugar de usar programas SPL.

Afortunadamente, por el momento la versión de CommUnity soportada por el CommUnity Workbench sólo hace uso de un conjunto limitado de tipos de datos. Todos ellos son representables en STeP, aunque existen algunas limitaciones, con respecto a las fórmulas verificables por el ambiente, cuando se utilizan tipos estructurados. Pos más detalles, el lector puede consultar [1].

Utilizando esta interpretación del diseño `Processor` en STeP, podemos intentar demostrar propiedades temporales del mismo, como mostraremos en la siguiente sección. Verificaremos, principalmente, propiedades de *safety* del diseño; por esto, hemos ignorado las restricciones de progreso con las cuales pueden etiquetarse las transiciones en STeP.

5 Propiedades Temporales de Programas CommUnity

Una vez traducido un diseño CommUnity a un sistema de transición de estados en STeP, podemos utilizar este ambiente para intentar demostrar, ya sea mediante model checking o mediante deducción semi-automática (o una combinación de éstas), aserciones acerca del diseño, expresables en la lógica de Manna-Pnueli. Para comenzar, consideremos la siguiente aserción:

> A lo largo de la ejecución de Processor, el generador de identificadores de procesos `CurrId` no puede tomar valores menores o iguales a 0

Esta aserción puede expresarse fácilmente, de la siguiente manera:

```
[] CurrId > 0
```

Otra aserción importante a considerar es la siguiente:

> En la cola `Buffer`, no pueden existir en ningún momento dos procesos diferentes con el mismo identificador.

Esto puede expresarse con relativa facilidad, de la siguiente forma

```
[] Forall i,j:int . i>=1 /\ i<j /\ j<=BufferLength -->
~ (#1 Buffer[i] = #1 Buffer[j]) \/ (#1 Buffer[i] = 0 /\ #1 Buffer[j]=0)
```

Las dos aserciones anteriores corresponden a propiedades de safety del sistema, que demostraremos en la siguiente sección. Nótese el uso del operador temporal $\Box$ (siempre en el futuro). Como esta fórmula es interpretada en el estado inicial del sistema, indica que la fórmula de primer orden bajo el alcance del operador temporal $\Box$ debe preservarse invariante a lo largo de la ejecución del sistema.

Consideremos ahora otra aserción interesante del sistema.

> En la cola `Buffer`, no puede existir un proceso con 0 segundos por procesar.

Nuevamente, esta aserción corresponde también a una propiedad de safety del sistema que mostramos más arriba, la cual puede expresarse en la lógica de Manna-Pnueli de la siguiente manera:

```
[] (Forall i: int . 0<=i /\ i<BufferLength -->
~ (#1 Buffer[i] > 0 /\ #2 Buffer[i] = 0)
```

5.1 Ejemplos de Verificación de Propiedades

En esta sección mostraremos cómo podemos verificar las propiedades de la sección anterior en el ambiente STeP.

- `[] CurrId > 0`

  Esta propiedad es una propiedad de safety del sistema. Debido a sus características, ésta es fácilmente demostrable mediante la regla B-INV disponible en STeP. Esta regla permite demostrar que una fórmula de la forma $\Box\alpha$ es una propiedad del sistema probando que α se satisface en el estado inicial y todas las transiciones del sistema la preservan. Claramente la regla B-INV sólo se puede probar la validez de una frmula $\Box\alpha$ si α es un *invariate inductivo*.

  El demostrador de teoremas de STeP, que incluye algorítmos de desición sofisticados para tipos como los enteros, probó la invarianza de `CurrId > 0` automáticamente. Más aún, puede utilizarse *model checking* para probar esta propiedad. Esta propiedad es, entre otras, encontrada como invariante de sistema por las herramientas de generación automática de invariantes de STeP.

- ```
  [] Forall i,j:int . i>=1 /\ i<j /\ j<=BufferLength -->
  ~ (#1 Buffer[i] = #1 Buffer[j]) \/
    (#1 Buffer[i] = 0 /\ #1 Buffer[j]=0) }
  ```

  A diferencia de la propiedad anterior, ésta no es un invariante inductivo. Luego, fallaríamos si intentáramos aplicar la regla B-INV para probarla. Para probar esta propiedad utilizamos la regla G-INV provista por STeP, G-INV es una generalización de la regla B-INV que nos permite probar que α es una invariante dentro del sistema, proveyendo una fórmula β más fuerte (en sentido lógico) que α que sea invariante inductivo. Si bien existe técnicas de generación automáticas de invariante inductivos, éstas tienen serias limitaciones y por lo tanto la fórmula β mencionada debe ser provista por el usuario. Para la prueba de la propiedad que nos concierne, la fómula β que consideramos es:

  ```
  []( (Forall i,j:int . i>=0 /\ i<j /\ j<BufferLength -->
     ~(#1 Buffer[i] = #1 Buffer[j]) \/
      (#1 Buffer[i] = 0 /\ #1 Buffer[j]=0) )
    /\
      (Forall x: int. x>=0 /\ x< BufferLength -->
     ~( (#1 Buffer[x]) = (#1 Curr) ) \/
      ( (#1 Curr = 0) /\ (#1 Buffer[x] = 0) ) )
    /\
      (Forall y: int. y>=0 /\ y< BufferLength -->
      ( #1 Buffer[y] < CurrId ))
    /\
      (#1 Curr < CurrId )
    /\
      (0<= First /\ First <BufferLength)
    /\
      (0<= Last /\ Last <BufferLength))
  ```

  La demostración de que esta fórmula es invariante inductivo en el sistema no es trivial. Por el contrario, requiere la intervención constante de usuario en la
  ```

prueba. Durante la demostración de la invarianza de esta fórmula las subtareas mas complejas fueron las correspondientes a la invarianza de `switch` y `kill` respecto del sistema.

- `[] (Forall i: int . 0<=i /\ i<BufferLength -->`
  `~ (#1 Buffer[i] > 0 /\ #2 Buffer[i] = 0)`

  Nuevamente esta propiedad no es un invariate inductivo y por lo tanto es necesario fortalecerla adecuadamente. El fortalecimiento de la misma, el cual dejamos como ejercicio para el lector interesado, involucra la relación entre el `Buffer` y `Curr` y las operaciones `load`, `kill` y `switch`.

El lector puede quedar con la sensación de que las propiedades verificadas son válidas gracias a que el sistema está correctamente implementado y en detalle. Sin embargo la versión del sistema mostrada en este artículo fué "depurada" gracias a varios errores detectados en los modelos iniciales, mediante model checking y otras técnicas, en las actividades de verificación. Por razones de presentación y espacio decidimos mostrar sólo la versión ya depurada.

6 Morfismos en CommUnity y Verificación Modular de propiedades

6.1 Morfismos de Refinamiento

Como mencionamos anteriormente, existen en CommUnity distintas relaciones entre diseños, formalizadas como morfísmos en el sentido categórico. Un tipo particular de morfismo es el denominado *morfismo de refinamiento*. Como el lector puede anticipar, la existencia de un morfismo de refinamiento entre un diseño R y uno R' indica que R' es una versión más detallada (es decir, una versión mas cercana a la implementación) del diseño R. Tales morfismos pueden aprovecharse para facilitar las actividades de verificación, de hecho, cualquier propiedad temporal de safety del diseño R será preservada por el diseño R'. Como R es más "simple" que R', las puebas de invarianza en R son más simples que en R'. Por el contrario los morfismos de refinamiento no preservan las propiedades de liveness (ejemplos de esto son ampliamente conocidos en la literatura).

6.1.1 Refinamiento de Datos

Si bien, como mencionamos anteriomente, un morfismo de refinamiento entre un diseño R y uno R', indica que R es una versión mas abstracta de R', los tipos de datos de R son preservados en R'. Es decir, los morfismos de refinamiento en CommUnity no soportan refinamiento de datos. Esto es un restricción muy importante, pues un mecanismo de simplificación de verificaciones exitoso, llamado *interpretación abstracta* [3], se basa precisamente en esta característica. Aunque los morfismo CommUnity no soportan interpretación abstracta, STeP lo hace a través de *diagramas de verificación*. A modo de ejemplo, consideremos la propiedad:

```
[] (Forall i: int . 0<=i /\ i<BufferLength -->
~ (#1 Buffer[i] > 0 /\ #2 Buffer[i] = 0)
```

En esta fórmula `#1 Buffer[i]` puede tomar en principio infinitos valores (cualquier valor entero), sin embargo en lo que concierne a la prueba de la propiedad, sólo nos

interesa saber si `#1 Buffer[i]` es 0 o mayor que 0. Luego el espacio de estados para `#1 Buffer[i]` puede reducirse a un valor booleano. De esta manera, la prueba de la propiedad anterior puede simplificarse notoriamente, y la reducción de estados puede facilitar la aplicación de técnicas que dependen de ello, como model checking.

6.2 Morfismos de Superposición

Además de los morfismos de refinamiento, CommUnity cuenta con otro tipo de morfismos, los llamados *morfismos de superposición*. Estos morfismos son el mecanismo principal de composición de componentes en CommUnity, y caracterizan la relación componente-sistema. Si existe un morfismo de superposición entre una componente R y una R', entonces R "es parte de" R'. Debido a las características de los morfismos de superposición de CommUnity, al igual que para los morfismos de refinamiento, las propiedades de safety son preservadas por los morfismos de superposición. Dado un morfismo de superposición entre una componente R y una S (S representa el sistema y R una de sus componentes), todas las propiedades de safety de R son preservadas por S. Sin embargo, no todas las propiedades de safety de S pueden demostrarse en R, incluso en el caso en que sólo hagan referencia a elementos de R; esto es debido a que tales morfismos pueden introducir *propiedades emergentes* de R en S [4]. Por lo tanto, aún en el caso en el cual una aserción no sea una propiedad de safety de R, esto no prohibe que lo sea de S. El usuario deberá ser cuidadoso con este tipo de aserciones.

Si bien los morfismos de refinamiento y de superposición permiten simplificar las tareas de verificación, éstos lo hacen de formas diferentes. La simplicidad correspondiente a morfismos de refinamiento tiene que ver con que uno puede moverse de una componente a una versión más abstracta de la misma (es decir, menos refinada). Como la versión simplificada tiene menos detalle, el espacio de estados o el número de transiciones disminuye, favoreciendo las técnicas de verificación (especialmente las automatizables). Los morfismos de superposición, en cambio, permiten movernos de un sistema a una de sus componentes. De esta manera, se modulariza la verificación, moviendose de un sistema potencialmente complejo, incluyendo las transiciones correspondientes a cada una de sus componentes (usualmente muy compleja, y correspondiente al colimite del diagrama que indica cómo están interconectadas las componentes) a sólo una componente del sistema. En esta se eliminan las transiciones correspondientes a las otras componentes, es decir, el "ambiente", desde el punto de vista de la componente factorizada.

7 Conclusiones

Hemos mostrado cómo puede emplearse el ambiente de verificación STeP para demostrar propiedades temporales de diseños CommUnity. Hemos discutido algunas formas de modularizar las actividades de verificación, aprovechando los distintos morfismos entre componentes que existen en CommUnity. Utilizamos un ejemplo, de un modelo simple de un procesador con planificador de procesos, para mostrar ejemplos de verificación de propiedades. Para este ejemplo, demostramos algunas propiedades de safety de primer orden utilizando el demostrador de teoremas de STeP y reglas de invarianza básicas, y una propiedad de safety proposicional, usando el model checker simbólico del sistema.

Parte de la traducción de diseños CommUnity a sistemas de transición de estados

en STeP ha sido automatizada (la que cubre los tipos básicos). Para programas CommUnity con tipos estructurados, como arreglos, por ejemplo, hemos tenido que realizar parte de la traducción a mano. Actualmente estamos trabajando el la modularización de las actividades de verificación mediante el aprovechamiento de morfismos, y su incorporación en la herramienta de traducción de CommUnity a STeP.

Como trabajo futuro a mediano plazo, estamos interesados en extender la traducción de CommUnity a STeP para la versión de CommUnity que soporta reconfiguración dinámica. Esto requerirá una codificación apropiada del estado del sistema, en término de las componentes activas y sus conexiones, ya que algunas operaciones del sistema generarán cambios en ésta, ya sea mediante cambios en las interconexiones, o creación y eliminación dinámica de componentes. Esta actividad tiene una importancia notoria, ya que pocas herramientas soportan el análisis de arquitecturas reconfigurables en la actualidad.

Referencias

[1] N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma and T. Uribe, *Verifying Temporal Properties of Reactive Systems: a STeP Tutorial*, in Formal Methods in System Design, vol 16, 2000.

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

[3] P. Cousot, *Abstract Interpretation*, ACM Computing Surveys 28(2), 324-328, 1996.

[4] J. Fiadeiro, *On the Emergence of Properties in Component-Based Systems*, in Proceedings of 5th International Conference on Algebraic Methodology and Software Technology AMAST 96, M. Wirsing and M. Nivat (eds), Lecture Notes in Computer Science 1101, Munich, Alemania, 1996.

[5] J. Fiadeiro and T. Maibaum, *Categorical Semantics of Parallel Program Design*, Science of Computer Programming 28(2-3), 1997.

[6] D. Garlan and R. Monroe and D. Wile, *ACME: An Architecture Description Interchange Language*, in Proceedings of CASCON'97, Toronto, Ontario, 1997.

[7] D. Garlan, *Software Architecture: A Roadmap*, in The Future of Software Engineering, A. Filkenstein (ed), ACM Press, 2000.

[8] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan and W. Mann, *Specification and Analysis of System Architecture Using Rapide*, IEEE Transactions on Software Engineering, Special Issue on Software Architecture, 21(4), 1995.

[9] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, *Specifying Distributed Software Architectures*, in Proceedings of the 5th European Software Engineering Conference (ESEC '95), Sitges, Spain, Lecture Notes in Computer Science, Springer-Verlag, 1995.

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

[11] N. Medvidovic, P. Oreizy, J. Robbins and R. Taylor, *Using Object-Oriented Typing to Support Architectural Design in the C2 Style*, in Proceedings of the ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering, ACM SIGSOFT, San Francisco, CA, 1996.

[12] N. Medvidovic and R. Taylor, *A Framework for Classifying and Comparing Architecture Description Languages*, In ESEC-FSE'97, 1997.

[13] M. Wermelinger, A. Lopes and J. Fiadeiro, *A Graph Based Architectural (Re)configuration Language*, in ESEC/FSE'01, V.Gruhn (ed), ACM Press, 2001.

```
Transition System %%Processor

in PSec: int
out Curr: (int * int)

local BufferLength : int where BufferLength = 20
local Buffer: array [1..BufferLength ] of ( int * int )
              where Forall x :[0..BufferLength-1] . (Buffer[x] = (0,0))
local CurrId: int
local First, Last : int where First = 1 /\ Last =1

Initially (Curr = (0,0)) /\ (CurrId =1) /\ First = 1 /\ Last = 1

Transition load:
modvar PSec
enable PSec > 0 /\ ((Last+1) mod BufferLength) != First
modrel Last' = ((Last+1) mod BufferLength)
modrel Buffer' = update(Buffer,(CurrId,PSec),Last)
modrel CurrId' = CurrId+1

Transition run:
enable (#2 Curr > 0 )
modrel Curr' = (#1 Curr, #2 Curr-1)

Transition kill:
enable (#2 Curr = 0) /\ Last != First
modrel Curr' = Buffer[First]
modrel Buffer' = update(Buffer,(0,0),First)
modrel First' =  ((First+1) mod BufferLength)

Transition switch:
enable (#2 Curr > 0 ) /\ Last != First
modrel Curr' = Buffer[First]
modrel Buffer' = update(update(Buffer,(0,0),First),Curr,Last)
modrel First' = ((First+1) mod BufferLength)
modrel Last' = ((Last+1) mod BufferLength)
```

Figura 2: Interpretación del diseño `Processor` como un sistema de transición de estados en STeP