Facultad de Ciencias Exactas, Ingeniería y Agrimensura

U.N.R.

Lic. en Ciencias de la Computación

Tesina de Grado

"Simulación a Eventos Discretos en Tiempo Real con ECD++ embebido en un Procesador de Red"

Ramello, Iván

iramello@fceia.unr.edu.ar

Director: Dr. Rodrigo Castro - rodrigocastro@ieee.org

Co-Director: Dr. Ernesto Kofman - kofman@fceia.unr.edu.ar

Asesor externo: Dr. Gabriel Wainer - gwainer@sce.carleton.ca

Diciembre 2010

Agradecimientos

Quiero agradacer, en primera instancia, a mi familia (Edgardo, Adriana y Cristian) por haber hecho posible que pueda estudiar y cursar ésta carrera, por su apoyo incondicional y su preocupación, en toda mi vida. Sin ellos, hubiese sido imposible para mí llegar hasta acá.

A mi director, Dr. Rodrigo Castro, por todo el tiempo invertido en mí, por su incentivo permanente, por su positivismo constante. No andaba nada, pero él te sacaba ánimo de donde sólo había Segmentation-Faults, y luego salíamos adelante. Su paciencia y supervisión fue un pilar para el desarrollo de la presente Tesina.

A todos mis compañeros de la facultad, que más que compañeros, se convirtieron en amigos. En particular, al Lic. Federico Bergero, por muchísimos motivos, desde la presentación con Rodrigo (que me permitió trabajar en ésto), pasando por consejos como de un padre, hasta haber acotado correcciones en la Tesina. Siempre estuviste, gracias Pela.

A mi novia, Dra. Eliana Moscatelli, por su paciencia y apoyo constante. Escuchar el "dale que te falta poco" por 3 años consecutivos tuvo efecto.

A mi amigo y compañero de cuarto por sólo 9 años, Ing. Damián Bártoli, por haber compartido muchos momentos durante nuestras carreras; al cual si bien tuve que aguantar sus ronquidos por los últimos 5 años, él tuvo que aguantarme a mí hablando de noche, especialmente cuando los límites tendían al infinito.

Resumen

La Tesina presenta el desarrollo de una interfaz de comunicación entre un simulador de eventos discretos basado en el formalismo DEVS y un NPU (Network Processor Unit) para aplicaciones genéricas de telecomunicaciones.

La nueva interfaz posibilita la ejecución de modelos DEVS a modo embebido y en tiempo real en un NPU, permitiendo utilizar a los propios modelos como producto final en un proceso de desarrollo de software de telecomunicaciones.

Siguiendo un desarrollo completamente basado en Modelado y Simulación DEVS, nuestra estrategia prevé combinar las ventajas de un enfoque práctico con el rigor de un método formal, conservando la continuidad de los modelos obtenidos desde las fases iniciales (especificación, simulación y verificación) hasta su aplicación en las fases finales (validación e implementación).

De este modo, se mitigan importantes problemas que surgen en las últimas fases de los desarrollos basados en Modelado y Simulación, los cuales típicamente requieren la reimplementación casi completa los modelos obtenidos para adaptarlos a las particularidades del hardware embebido destino.

La nueva interfaz desarrollada en esta Tesina permite la comunicación entre el simulador ECD++ (para ejecución de modelos DEVS en dispositivos embebidos) y el Network Processor Intel IXP2400. Este procesador integra dos jerarquías o niveles de hardware de procesamiento: uno de "alto nivel" basado en un procesador RISC Intel XScale y otro de "bajo nivel", basado en un cluster paralelo de 8 MicroEngines (microcontroladores multithreaded programables RISC) destinados a la manipulación eficiente de paquetes en buses de alta velocidad.

Se portó ECD++ para que ejecute en la jerarquía de alto nivel de IXP2400 permitiéndole realizar acciones de sensado y actuación sobre el hardware de bajo nivel. Esto hace posible que ECD++ ejecute modelos DEVS en modalidad Hardware-In-The-Loop (HIL) para diseñar controladores de tráfico de red en tiempo real.

Este trabajo representa el primer antecedente en portar un simulador DEVS a un Network Processor cerrando el lazo a modo HIL con circuitos específicos de procesamiento de paquetes.

A partir del nuevo mecanismo de comunicación bidireccional desarrollado, se pueden separar claramente las tareas de diseño de modelos DEVS (en C++) y las tareas de programación de algoritmos específicos para manipulación de paquetes (en microC o assembler), habilitando el co-desarrollo interdisciplinario de soluciones embebidas complejas en el área de telecomunicaciones.

Índice

1. Introducción

Durante décadas la comunidad de ingeniería de software ha perseguido el objetivo de crear métodos formales para el desarrollo de sistemas embebidos, y en particular para aquellos con restricciones de tiempo real. A pesar de los numerosos esfuerzos, la mayoría de los métodos existentes son todavía difíciles de escalar, y requieren de costosos esfuerzos de *testing* sin garantizar productos libres de errores. En cambio, la ingeniería de sistemas se ha valido generalmente de técnicas de modelado y simulación (MyS) para mejorar el desarrollo y obtener productos de alta calidad. El desarrollo basado en MyS es ampliamente utilizado para las etapas tempranas de un proyecto, pero cuando se evoluciona hacia la implementación en la plataforma de hardware final, usualmente los modelos iniciales son abandonados debido a los drásticos cambios de entorno, de paradigma de programación, de manipulación de datos y de control del avance del tiempo.

En la presente Tesina se propone un enfoque basado en MyS DEVS para mitigar el problema mencionado en las fases finales de desarrollo de software para sistemas embebidos en tiempo real. La estrategia prevé combinar las ventajas de un enfoque prático con el rigor de un método formal, permitiendo conservar la vigencia de los modelos (utilizados previamente para las fases especificación, simulación y verificación) para su uso en las etapas finales de validación e implementación.

Producto de la presente Tesina, se ha logrado una producción científica[6], la cual se transcribe parcialmente en el Apéndice B.

1.1. Especificación de Objetivos

El objetivo final es obtener un procedimiento completamente basado en modelos DEVS (Discrete EVent Systems specification) los cuales, luego de ser diseñados y verificados, puedan ser:

1. embebidos directamente el hardware de destino final
2. ejecutados –simulados– en tiempo real
3. conectados de forma transparente a unidades de hardware externas con capacidad de acción y reacción (modalidad Hardware-In-the-Loop, HIL)

La estrategia mantiene la continuidad de los modelos DEVS, al encapsular las funcionalidades de interacción con el hardware externo específico, dentro de modelos atómicos especiales (Mappers) capaces de interactuar con interfaces de control (Drivers) propias de cada dispositivo.

Para dichos objetivos se utilizará la herramienta ECD++ basada en DEVS presentada posteriormente, dado su diseño específico para sistemas embebidos en tiempo real. Se realizará una integración de ECD++ con un procesador de red Intel IXP2400 y se desarrollarán bibliotecas de interfaz para la operación tipo HIL entre el simulador y los dispositivos de bajo nivel de dicho procesador.

Adicionalmente se integrarán y preconfigurarán las múltiples herramientas de desarrollo necesarias para la experimentación con ECD++ y la placa de red RadiSys ENP-2611 (basada en un IXP2400), creando un laboratorio virtualizado portable que simplifique las tareas de validación e implementación de modelos.

1.2. Organización

El informe comienza con el captítulo 2 presentando algunos conceptos teóricos necesarios para la compresión del trabajo, incluyendo el modelado en el lenguaje DEVS, la presentación del simulador ECD++, y conceptos teórico-práctico definiendo las características generales de un Procesador de Red, para luego introducir las características particulares del Procesador de Red que utilizaremos, junto con la arquitectura para desarrollo. Luego en el Capítulo 3 se verá una breve introducción a la solución que se presenta con éste trabajo, junto con un ejemplo que suscita el desarrollo del mismo. Se continúa en el Capítulo 4, con los distintos aspectos que hubo que cubrir para el éxito del trabajo, desde el armado de la infraestructura por hardware y software, pasando por las decisiones de diseño e implementación tomadas, hasta la programación de los diferentes tipos de hardware que posee nuestro Procesador de Red. Se sigue comentando en el Capítulo 5, experimentos realizados utilizando el desarrollo de la presente Tesina, en primera instancia involucrando eventos virtuales, para luego pasar a interactuar con tráfico de red real. Se presenta en el Capítulo 6, las conclusiones de la presente Tesina, para finalizar en el Capítulo 7 con posibles trabajos a realizar, que puedan extender el mismo.

2. Conceptos Preliminares

En este capítulo se introducirán los conceptos teóricos y técnicos esenciales para comprender el desarrollo posterior de esta Tesina. Se explicará el formalismo DEVS y se presentará un simulador particular que implementa dicho formalismo con características especiales para ejecutar en tiempo real en entornos embebidos. Luego se presentará el concepto de Procesador de Red, un tipo particular de sistemas híbridos concentrados en un único chip, en el cual coexisten distintas jerarquías de procesamiento (de propósito genérico y de propósito dedicado). Finalmente se presentarán las tecnologías de hardware y software que serán utilizadas como plataforma de test, siguiendo la idea de los Procesadores de Red: El procesador Intel IXP2400, la arquitectura de software IXA y la placa de red Radisys ENP-2611.

2.1. El Formalismo DEVS

DEVS es un lenguaje desarrollado en 1976 por B. Zeigler [28, 22] para modelar y analizar sistemas generales que puedan ser descriptos por eventos discretos. El mismo provee un entorno para la construcción de modelos jerárquicos de una manera modular, permitiendo que el modelo sea reutilizado, disminuyendo el tiempo de desarrollo y prueba. Se especifican sistemas cuyos estados cambian, o bien por la recepción de un evento externo, o bien por la expiración de un tiempo específico. Se han desarrollado una gran variedad de métodos para utilizar este lenguaje para simular también sistemas continuos –ecuaciones diferenciales– (utilizando métodos de integración numérica que aproximan éstos tipos de sistemas[7]) y sistemas híbridos[12, 23]. Dada su versatilidad es utilizado extensamente en áreas de investigación y experimentación.

Específicamente, un modelo DEVS[29] procesa una trayectoria de eventos de entrada y –acorde a dicha trayectoria y a su propio estado inicial– produce una trayectoria de eventos de salida. Este comportamiento se muestra en la Figura 1.

Un *evento* es la representación de un cambio instantáneo en alguna parte del sistema. Como tal, puede caracterizarse mediante un valor y un tiempo de ocurrencia. El valor puede ser un número, un vector, una palabra, o en general, un elemento de un conjunto arbitrario.

Figura 1: Comportamiento Entrada/Salida de un modelo DEVS.

Una *trayectoria* queda definida por una secuencia de eventos. La misma toma el valor ϕ (o *No Event*) en casi todos los instantes de tiempo, excepto en los instantes en los que sí hay eventos. En estos instantes, la trayectoria toma el valor correspondiente al evento. La Figura 2 muestra una trayectoria de eventos que toma los valores $x2$ en el tiempo $t1$, luego toma el valor $x3$ en $t2$, etc.

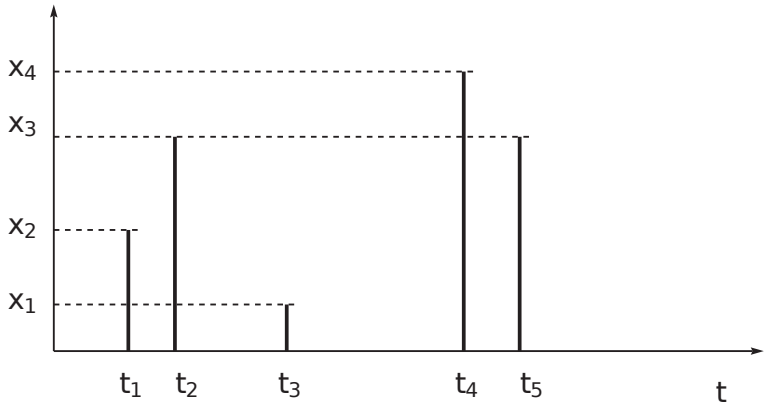

Figura 2: Ejemplo de una trayectoria de eventos.

Formalmente un modelo DEVS está descripto por los componentes que detallamos a continuación.

2.1.1. Modelos DEVS atómicos

Un modelo atómico representa la unidad "molecular" de procesamiento, en el sentido que es la pieza fundamental y más básica. Formalmente un modelo atómico esta conformado por la 7-upla

$(X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$ donde:

- X es el conjunto de valores de entrada que acepta el atómico, es decir un evento de entrada tiene como valor un elemento del conjunto X
- Y es el conjunto de valores de los eventos de salida que puede emitir el atómico.
- S es el conjunto de estados internos del modelo, en todo momento el atómico está en un estado dado, que es un elemento del conjunto S .
- ta es una función $S \rightarrow R^+$, que indica cuanto tiempo permanecerá en un estado dado el atómico, si es que no se "recibe" ningún evento de entrada.
- δ_{int} es una función $S \rightarrow S$, que indica la dinámica del sistema en el momento que el atómico realiza una transición interna. Sería el análogo a una tabla de transición en otros autómatas.
- δ_{ext} es una función $(S, e, X) \rightarrow S$, que indica el cambio de estado en la presencia de un evento externo
- λ es una función $S \rightarrow Y$ que indica qué evento se debe emitir, dado un estado interno.

Los conjuntos S, X e Y son arbitrarios, y en general infinitos, a diferencia de lo que ocurre con los autómatas de estados finitos y otros formalismos similares.

Cada posible estado s ($s \in S$) tiene asociado un *Avance de Tiempo* calculado por la *Función de Avance de Tiempo* $ta(s)$. Luego, si el estado toma el valor s_1 en el tiempo t_1, tras $ta(s_1)$ unidades de tiempo (o sea, en tiempo $ta(s_1) + t_1$) el sistema realiza una *transición interna* yendo a un nuevo estado s_2. El nuevo estado se calcula como $s_2 = \delta_{int}(s_1)$. La función δ_{int} se llama *Función de Transición Interna.*

Cuando el estado va de s_1 a s_2 se produce también un evento de salida con valor $y_1 = \lambda(s_1)$. La función λ ($\lambda : S \rightarrow Y$) se llama *Función de Salida.* Así, las funciones ta, δ_{int} y λ definen el comportamiento autónomo de un modelo DEVS.

Cuando llega un evento de entrada el estado cambia instantáneamente. El nuevo valor del estado depende no sólo del valor del evento de entrada sino también del valor anterior de estado y del tiempo transcurrido desde la última transición. Si el sistema llega al estado s_3 en el instante t_3 y luego llega un evento de entrada en el instante $t_3 + e$ con un valor

x_1, el nuevo estado se calcula como $s_4 = \delta_{ext}(s_3, e, x_1)$ (notar que $ta(s_3) > e$). En este caso se dice que el sistema realiza una *transición externa*. La función δ_{ext} se llama *Función de Transición Externa*. Durante una transición externa no se produce ningún evento de salida.

2.1.2. Modelos DEVS acoplados

La descripción de un sistema puede ser (formalmente hablando) completamente realizada utilizando modelos atómicos, aunque esto resulta un poco incómodo y confuso. Los conjuntos de estados y las funciones de transición se vuelven inmanejables en sistemas complejos, y nunca podemos asegurar haber cubierto todos los posibles estados. Para abordar este problema, el formalismo DEVS introduce lo que se llaman *modelos acoplados* (ver figura 3) que es una forma de agrupar modelos DEVS y generar nuevos modelos a partir de este agrupamiento. Hay dos formas de acoplamiento, la más general, en la cual se utilizan funciones de traducción entre los sub-sistemas y otra clase que adopta el uso de puertos para la comunicación entre sub-sistemas. Describiremos la segunda clase (ya que es la más simple y la utilizada en ECD++[1]).
Formalmente un modelo acoplado está representado por la octo-upla:

$N = (X_N, Y_N, D, \{M_d\}, EIC, EOC, IC, Select)$ donde cada componente es:

- X_n es el conjunto de eventos de entrada al modelo acoplado, representado por el producto cartesiano del conjunto de puertos de entrada $InPorts$ y el conjunto de posibles valores para este puerto, o sea un evento de entrada al acoplado está representado por un par (p, v) donde $p \in InPorts$ y $v \in X_p$.
- Y_n es el conjunto de eventos que el modelo puede emitir. Es un elemento del producto cartesiano entre el conjunto de puertos de salida $OutPorts$ y el conjunto de posibles valores para este puerto, o sea un evento de salida del acoplado está representado por un par (p, v) donde $p \in OutPorts$ y $v \in Y_p$.

[1] La forma de comunicación por puertos es un caso particular de la forma que utiliza funciones de traducción, en el sentido que, todo lo que se puede hacer con puertos puede lograrse también con funciones de traducción.

- D es el conjunto de los índices a los modelos DEVS (atómicos y acoplados) que conforman este modelo
- $\{M_d\}$ es el conjunto de los modelos (son justamente los modelos que "acopla" o "agrupa").
- EIC,EOC estos conjuntos indican la forma que los modelos internos al acoplado, se relacionan con los externos.
  - EIC (o External Input Coupling) son las conexiones de entrada al acoplado, es decir, conecta un puerto de entrada del acoplado con un puerto de entrada de un modelo perteneciente al acoplado.
  - EOC (o External Output Coupling) son las conexiones de salida del acoplado. Conecta un puerto de salida de un modelo interno del acoplado con un puerto de salida del acoplado. Formalmente:

$$EIC \in \{((N, ip_N), (d, ip_d))|ip_n \in InPorts, d \in D, ip_d \in InPorts_d\} \text{ y}$$
$$EOC \in \{((d, op_d), (N, op_N))|op_n \in OutPorts, d \in D, op_d \in OutPorts_d\}$$

- IC representa las conexiones internas del acoplado donde

$$IC \in \{((a, ip_a), (b, ip_b))|a, b \in D, ip_a \in OutPorts_a, ip_b \in InPorts_b\}$$
$$\text{donde no se permite que } a = b$$

- $Select$ es una función ($2^D \to D$) que decide qué modelo realizará primero su transición interna, si se da el caso de eventos simultáneos. Es una clase de función de "desempate" que en ciertos modelos es necesaria. Debe cumplir $Select(E) \in E$, con $E \subset 2^D$ siendo 2^D el sub-conjunto de componentes que produce la simultaneidad de eventos.

Los modelos acoplados son en sí mismos modelos DEVS válidos, formalmente el acoplamiento (como lo definimos antes) es una operación cerrada sobre el conjunto de modelos DEVS. Acoplar modelos DEVS define nuevos modelos DEVS. Sin esta cualidad el acoplamiento resultaría inútil desde del punto de vista del formalismo. También trae muchas ventajas a la hora de describir modelos DEVS y a la hora de simularlos. El acoplamiento da lugar a una estructura jerárquica de desarrollo.

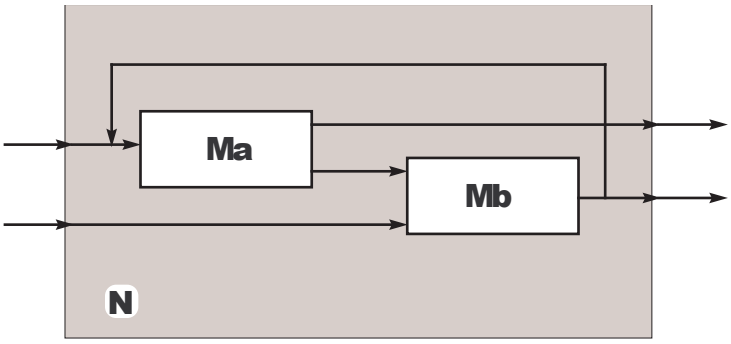

Figura 3: Modelo acoplado

2.2. Simulación Embebida y en Tiempo Real

En principio, DEVS permite representar cualquier sistema que realice un número finito de cambios en intervalos finitos de tiempo. Por esto, mediante el mismo, se puede representar cualquier tipo de sistema de eventos discretos y de tiempo discreto. Además, teniendo en cuenta que la simulación de sistemas continuos en general requiere de algún tipo de discretización, es posible simular mediante DEVS sistemas continuos aproximados mediante cualquier método de integración numérica. En otras palabras, DEVS es la herramienta más general de simulación de sistemas discretos. Esta generalidad ha impulsado el desarrollo de múltiples herramientas de simulación basadas en DEVS. Entre las más difundidas, podemos mencionar a CD++ (descrito posteriormente), DEVS-Java[30] y PowerDEVS[1].

Un sistema de tiempo real, en el sentido más general, es aquel en el cual el resultado obtenido no sólo debe ser correcto sino que también debe ser "entregado" en el momento correcto y que responde a estímulos cumpliendo restricciones temporales.

Los sistemas operativos de tiempo real son sistemas operativos que ofrecen las herramientas básicas para poder implementar tareas con restricciones temporales. Un sistema operativo debe ofrecer métodos para expresar las restricciones temporales de cada tarea, métodos de comunicación entre éstas tareas y manejo de los recursos de bajo nivel de la computadora (memoria, interrupciones, puertos, etc).

Se denomina simulación en tiempo real a las simulaciones que se realizan de manera sincronizada con el reloj físico, de manera tal que la simulación se transforma en una

emulación del sistema real. Esta sincronización es un requisito fundamental cuando la simulación debe interactuar con personas (*Man in the Loop*), o con otros equipos (*Hardware in the Loop*). A modo de ejemplo, en [2] se realizó una adaptación del simulador PowerDEVS para trabajar en tiempo real, y en [26] se extendió CD++ con el mismo objetivo (llamándose RT-CD++).

Por otro lado, se tiene un simulador embebido, ya que se encontrará corriendo directamente sobre el dispositivo sobre el que se sensará y actuará (en éste caso, la placa ENP-2611), que ejecutado en tiempo real, nos permite construir un gran entorno de simulación tipo Hardware-In-The-Loop.

2.3. Concepto Hardware In The Loop

Hardware In the Loop (HIL), es una forma de simulación en tiempo real que difiere del resto por el agregado de un componente real en el lazo. Este componente, llamado dispositivo bajo ensayo, es por lo general un sistema embebido de control. Esta topología fue concebida para tratar con la creciente complejidad de las unidades electrónicas de control diseñadas para controlar plantas complejas tales como vehículos terrestres, satélites, naves espaciales, aviones, sistemas de guerra, etc.

En éste caso particular, el componente real en el lazo será directamente la interacción y control sobre la placa ENP-2611.

2.4. El simulador ECD++

CD++ [24] es una herramienta y software orientado a objetos que implementa el mecanismo de simulación de DEVS. Las herramientas de simulación tales como CD++, son únicamente utilizadas en el mundo simulado y es muy difícil para el modelador, validar su solución en el mundo real. Ésto hace que las soluciones desarrolladas en el mundo simulado no resuelvan precisamente problemas del mundo real, porque para muchos problemas complejos, como los sistemas de tiempo real, los modelos simulados no describen con 100 % de precisión su contraparte en la vida real.

Como se comentó anteriormente en 2.2, RT-CD++ extiende a CD++ permitiendo la simulación en tiempo real. Reemplaza las funciones de manejo de tiempo virtual por

funciones de avance de tiempo sincronizadas con el tiempo físico (provisto por el reloj de la plataforma computacional).

ECD++ [27] (por *Embedded CD++*) es una adaptación de RT-CD++ que puede correr en sistemas embebidos (con recursos restringidos) y puede ejecutar modelos DEVS que interactúen con eventos del mundo real. Ésta habilidad hace que sea una herramienta muy útil para desarrollar aplicaciones de tiempo real con una filosofía hardware-in-the-loop. Dicha técnica permite la transición gradual de modelos simulados a sus contrapartes reales (hardware) y permite entornos experimentales para facilitar el testing en un ambiente libre de riesgos.

2.5. Procesadores de Red

El crecimiento exponencial de Internet produjo la aparición de nodos de red inteligentes basados en dispositivos programables. La convergencia de voz y datos respetando la calidad de servicio requiere la capacidad de procesar paquetes tomando decisiones de control complejas y sosteniendo altas velocidades. Aun más, con la rápida evolución de los protocolos, estándares y aplicaciones, la lógica de control de las redes debe adaptarse a un ritmo mucho mayor que el previsto para el recambio tecnológico del hardware subyacente. A su vez, la velocidad de las redes crecen a un ritmo que supera al crecimiento de la velocidad de clock de las CPU en aproximadamente un orden de magnitud, haciendo comparables el tiempo de transmisión de un paquete con el de acceso a una posición de memoria. Esto presenta un desafío tecnológico complejo ya que requiere combinar la flexibilidad de programación de los procesadores de propósito genérico (CPUs tradicionales) con la alta performance de los circuitos de propósito específico (Application Specific Integrated Circuit, ASIC) para transmitir paquetes a velocidad de línea. Un Procesador de Red (Network Processor, NP) es un sistema en un chip (System-on-a-Chip - SoC) diseñado para dar solución a estos problemas, concentrando dispositivos heterogéneos en un único circuito integrado: procesadores de propósito general, microcontroladores programables de propósito específico, unidades de switch, unidades criptográficas, controladores de memoria interna y externa, etc. [8].

El diseño de los NP hace factible ubicar a ECD++ en el procesador de propósito genérico (que en la mayoría de los casos opera con alguna versión embebida de Linux) y

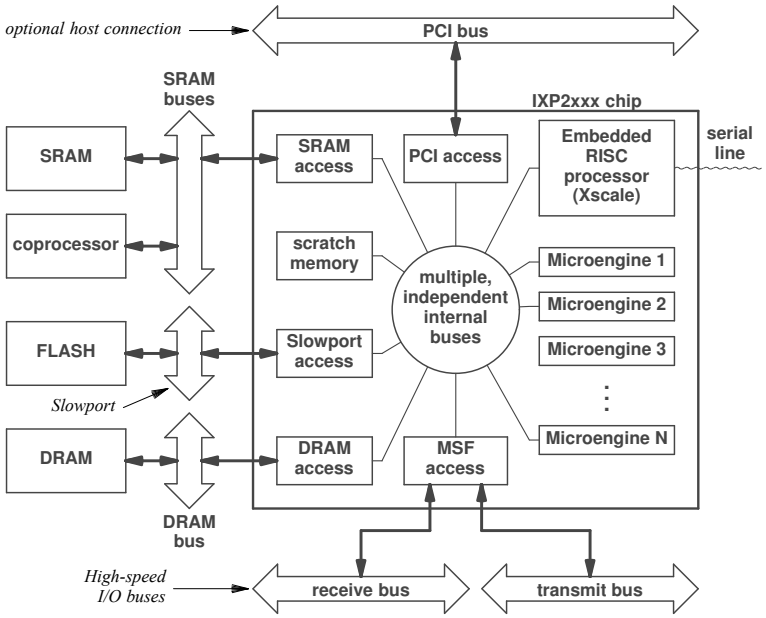

Figura 4: Arquitectura de Hardware de la familia IXP2xxx

luego valerse de las bibliotecas de desarrollo propias del NP para comunicar a ECD++ con los microcontroladores de propósito específico.

2.6. El procesador de red Intel IXP2400

El procesador Intel IXP2400 de 2.5 Gbps. [19] es un ejemplo de NP estructurado en dos niveles de procesamiento: uno de dinámica lenta (*Slow Data Path*) que consiste en un procesador RISC de propósito genérico XScale (ARM V5TE, 600 MHz, 32 bit, llamado *Core Processor*)[15], y uno de dinámica rápida (*Fast Data Path*), que consiste en un cluster de 8 microcontroladores RISC programables multithread (600 Mhz, 32 bits, llamados *MicroEngines* ó MEs) [5, 21].

La arquitectura del IXP2400 permite diseñar motores de reglas flexibles y reconfigurables residentes en el Core, de modo tal que puedan adaptarse dinámicamente sin interferir con la capacidad de las ME de sostener su performance nominal de procesamiento de paquetes [11].

En la Figura 4 se muestra un diagrama de bloques de la arquitectura de hardware

interna de la familia IXP2xxx. Se destacan a la derecha las dos unidades de procesamiento mencionadas: 1 Core (XScale) y N MicroEngines (con N=8 en el caso del IXP2400).

Los paquetes de red entran y salen vía la Media Switch Fabric (MSF) que provee acceso a los manejadores externos de la capa física de red. Los paquetes son recibidos, transformados y enviados por el código residente en las ME (MicroCode). Sólo ciertos paquetes excepcionales son escalados desde las ME hacia el Core para un tratamiento especial.

Las MEs se programan en Assembler o MicroC (una versión adaptada y reducida de ANSI C), contando con un set de más de 50 instrucciones que operan a nivel de bit, byte y long-word, se ejecutan en un pipeline de seis etapas y toman en promedio un unico ciclo de reloj en finalizar. Las MEs poseen capacidad para 8 threads cada una, y no poseen un sistema operativo. En cambio, se provee un sistema de señales por hardware con las que se administran los cambios de contexto entre threads ("hardware threads"). Es por ello que casi se alcanza latencia nula de context-switch. Son estas características, sumadas a los registros locales especiales de acceso rápido, las que hacen posible el manejo de paquetes a velocidad de línea. Adicionalmente se cuenta con un sistema de marcas de tiempo y temporizadores que ofrecen la capacidad de administrar el tiempo real con una precisión de 16 ciclos de reloj (aproximadamente 0.26 nanosegundos).

Probablemente el aspecto más importante a destacar del IXP2400 sea la organización jerárquica de los diversos tipos de memoria (interna y externa), y la posibilidad de ser compartida entre el Core (donde residirá ECD++) y las MEs. Los tipos de memoria disponibles son SRAM (8 MBytes), DRAM (256 MBytes), Scratchpad (16 KBytes) y Local (2560 bytes por cada ME). SRAM y DRAM son RAMs estática y dinámica respectivamente externas al chip, y pueden ser accedidas tanto por las MEs como por el Core. Scratchpad es una memoria de baja latencia para señalización veloz entre los threads de las MEs desde y hacia el Core, en forma de interrupciones, datos compartidos y/o estructuras en anillo.

La comunicación interna de baja latencia entre MEs es crítica, y para ello se utiliza la memoria tipo Local que no es accesible por el Core. Incluso existen registros especiales llamados Next Neighbor (NN) que pueden ser accedidos unicamente por 2 ME adyacentes.

En la Figura 5 se muestra un diagrama de bloques del hardware interno de una MicroEngine. En la misma se puede observar la presencia de los 512 registros XFER In y

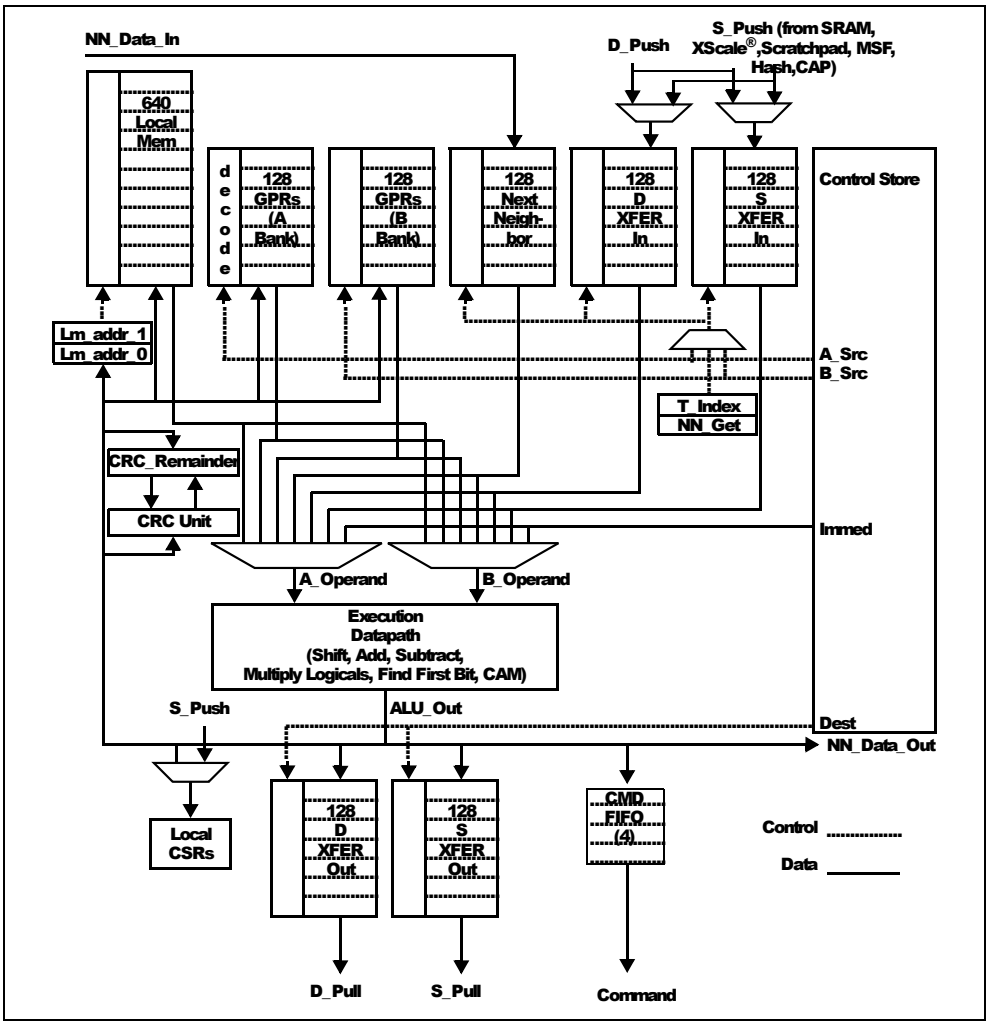

Figura 5: Arquitectura de Hardware de una MicroEngine

Out[20], los cuales posibilitan todos los tipos de accesos desde/hacia una ME, y que utilizaremos posteriormente en nuestro microcódigo, para comunicar las variables de ECD++ con las MicroEngines.

Las MEs operan mucho más rápido que la memoria externa, completando instrucciones dentro de un mismo ciclo de reloj. Luego, un simple acceso a la memoria RAM podrá bloquear la ejecución por varios ciclos de reloj. Comparativamente, mientras acceder a la memoria Local insume 1 ciclo de reloj, acceder al Scratchpad insume 60 ciclos, a la SRAM 150 ciclos y a la DRAM 300 ciclos.

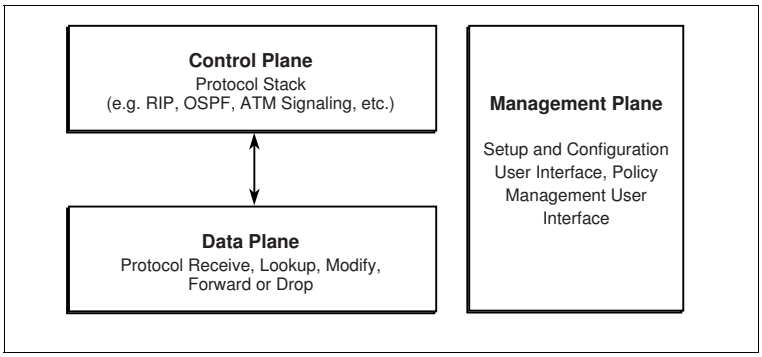

Figura 6: Estructura de una aplicación de networking

En síntesis, utilizando memorias y estructuras de datos especializadas y un reemplazo de un sistema operativo por hardware threads, las MEs proveen alta performance comparable a la de los ASICs, programabilidad comparable a la de las CPUs, y comunicación con los niveles superiores de dinámica lenta (Core) para lograr administrabilidad, tratamiento de casos excepcionales y flexibilidad ante cambios de políticas de calidad de servicio.

2.7. Arquitectura IXA para Aplicaciones de Red

Antes de presentar IXA, explicaremos brevemente la estructura general de una aplicación de red. Típicamente, una aplicación de red opera en tres planos lógicos, denominados Plano de Datos (*Data-Plane*), Plano de Control (*Control-Plane*) y Plano de Administración (*Managment-Plane*) (Figura 6).

El Plano de Datos procesa y reenvía paquetes a alta velocidad, siendo lo más determinante en cuanto a la performance, ya que todos los paquetes pasan por aquí. El Plano de Control se encarga de los mensajes de protocolos específicos, y es responsable por la configuración y actualización de tablas y datos utilizados por el plano de datos en búsquedas. El Plano de Administración es responsable por la configuración general del sistema, de la recolección y reporte de estadísticas, y del arranque/detención de aplicaciones en respuesta a solicitud del usuario, o bien por otras aplicaciones.

IXA (Internet eXchange Architecture) define un marco estándar para desarrollar aplicaciones de red modulares, basadas en bloques de código reusables y reconfigurables, y portables entre distintos NP. La idea básica es definir capas de responsabilidades y proveer

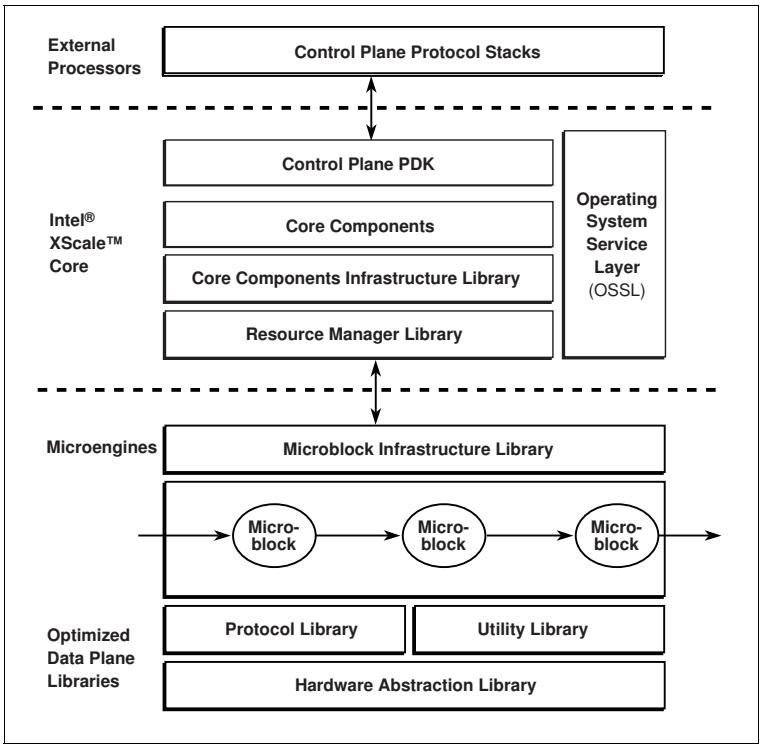

Figura 7: Arquitectura IXA

para cada capa, bibliotecas estándar accesibles mediante interfaces (Application Program Interface, API) bien definidas. IXA abarca tanto el software que ejecuta en el Core como el de las MEs, e Intel provee una gran cantidad de módulos estándar para utilizar a ambos niveles. Una aplicación de red con IXA es esencialmente una cadena de tareas aplicadas secuencialmente a un flujo constante de paquetes. Diversas tareas se asignan a distintas MEs, cada una de las cuales se encargará de cumplir esa determinada función.

IXA es implementado usando una arquitectura en capas, como muestra la Figura 7. Cada tarea asignada a cada ME posee un componente de software asociado en el Core con el cual intercambia información de control, medición y administración. IXA estructura y ordena las arquitectura de software (bibliotecas) que hacen posible orquestar esta multiplicidad de piezas de código ejecutando en modo paralelo y asincrónico. Una breve descripción de algunos componentes de la arquitectura:

- Bibliotecas optimizadas del Plano de Datos (*Optimized Data-Plane Libraries*): con-

sisten en macros en assembler o funciones en MicroEngineC, de bajo nivel, usadas para escribir microbloques o cualquier otro código para MicroEngine, las cuales están optimizadas para alta performance y poca utilización de espacio de código. Ejemplo: parsing de encabezados de protocolos.

- Microbloques (*Microblocks*): Cada uno es un macro o una función en MicroEngineC, escrita usando las bibliotecas optimizadas de bajo nivel. La inteción es que cada microbloque sea independiente de los demás, lo que mejora la reusabilidad y flexibilidad.

- Biblioteca del Administrador de Recursos (*Resource Manager*, RM): es un componente en el Core, el cual provee una API para la inicialización y configuración de hardware y la administración de recursos (que utilizaremos posteriormente); además de permitir la comunicación entre microbloques y sus componentes asociados en el Core.

- Componentes del núcleo del Intel XScale (*Core Component*, CC): implementa la configuración, administración y manejo de excepciones para un microbloque asociado (puede manejar más de uno, incluso puede llegar a existir sólo uno que administre todos los microbloques).

- Capa de Servicios del Sistema Operativo (*Operating System Service Layer*, OSSL): provee una capa de abstracción para todo el código que corre en el Intel XScale. Es usado por las otras capas para mejorar la portabilidad frente a múltiples sistemas operativos. Provee administración de threads, primitivas de sincronización, exclusión mutua, timers, administración de memoria, entre otros.

El término IXP (Internet eXchange Processor) se refiere a una familia de NPs que implementan la arquitectura IXA.

2.8. Placa de red Radisys ENP-2611

Se trabajará con una placa de red Radisys ENP-2611-256 [10] (Figura 8) como plataforma de desarrollo, que es un producto comercial basado en el NP IXP2400, e implemen-

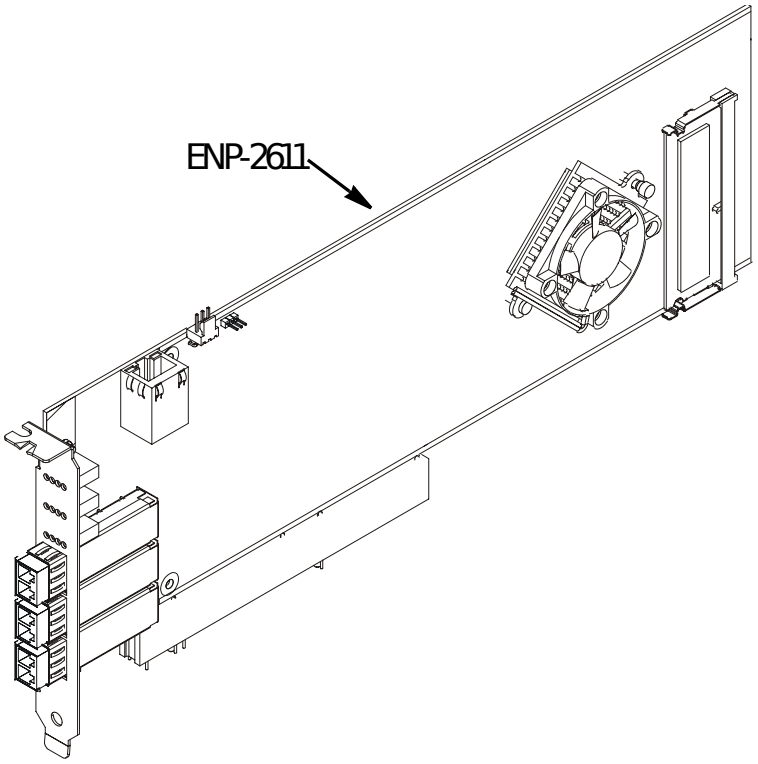

Figura 8: Radisys ENP-2611

tado con un conector PCI. Los principales componentes de esta placa de red se enumeran a continuación:

- Procesador IXP2400 a 600 MHz.
- 16 MBytes de memoria StrataFlash para almacenar el código de arranque, y donde se tiene el kernel de Linux de tiempo real.
- Puente SPI-3 FPGA, el cual conecta directamente a la interfaz del Media Switch Fabric (MSF) del IXP2400 corriendo en modo POS-PHY nivel 3 (SPI-3); y nos provee conexión y ruteo de datos entre el IXP2400 y las MACs Gigabit Ethernet PM3386/7.
- Un controlador PM3386 Dual Gigabit Ethernet que provee dos interfaces, conectado al puerto “PHY0” del puente FPGA SPI-3, con tranceivers ópticos que se ajustan

a la norma SFP con conectorizado LC, necesario para la transmisión y recepción de datos por los medios físicos.

- Un controlador PM3387 Gigabit Ethernet que provee una única interfaz, conectado al puerto "PHY1" del puente FPGA SPI-3, el cual también se ajusta a la norma SFP para su conector LC.
- Un conector RJ45 con LEDs para una conexión 10/100BaseT soldado internamente, para propósitos de depuración (controlador Intel 82559). Mediante ésta interfaz, ENP-2611 tomará la IP correspondiente, descargará su RootFS y podremos accederle por telnet.
- Un conector de 3-pines para la conexión de una interfaz serie RS-232, por la cual podremos acceder en caso de que sea necesario.

Las ubicaciones físicas se pueden observar en la Figura 9

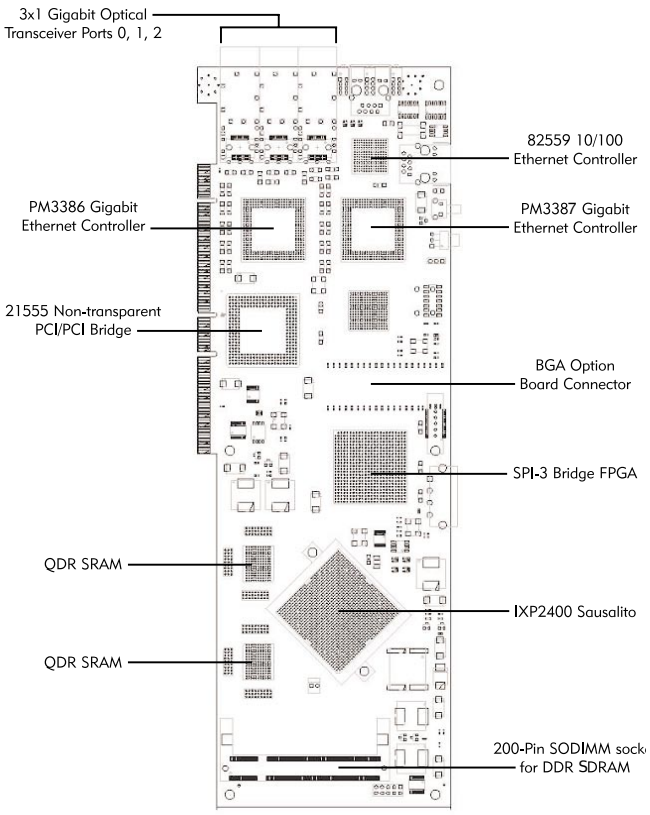

Figura 9: Diagrama de componentes ENP-2611

3. Solución propuesta

En éste capítulo se verá cómo se pretende vincular ECD++ con la arquitectura de Intel, y luego la introducción de un caso de aplicación que promueve el desarrollo de la presente Tesina.

3.1. Integración de ECD++ con la Arquitectura IXA

De todas las bibliotecas disponibles mencionadas anteriormente, las de mayor interés para lograr la integración de ECD++ son: Core Components (CC), Resource Manager (RM), y MicroBlocks (MB). Las API de CC permiten crear módulos de kernel Linux en el Core, que utilizando las API de manejadores de recursos (RM) acceden a estructuras de memoria compartidas entre el Core y las ME. A su vez, las MEs utilizan las bibliotecas MB para acceder a dicha memoria, pero utilizando una estructura de punteros completamente distinta a la utilizada desde el Core. Con esta estrategia, se hace transparente al programador los intrincados detalles específicos de comunicación entre los 2 tipos de hardware que coexisten en el IXP2400. Luego, cualquier nueva funcionalidad desarrollada para el Core y/o las MEs será reusable y portable a cualquier procesador que se ajuste a la arquitectura IXA.

Como puede observarse en la Figura 10, ECD++ se ubica en el Slow Data Path. Se desarrollará una biblioteca llamada ECD++ I/O Core, consistente en un Core Component que es invocado por ECD++ para comunicar los modelos DEVS con el código que implementa los protocolos de red en el Fast Data Path. También se desarrollará una biblioteca ECD++ IXA I/O microblock para comunicarse con los modelos DEVS en el Core. Con esta infraestructura, se pueden reemplazar los modelos DEVS que representan al sistema de red real, por puertos de comunicación hacia y desde el sistema real (Fast Data Path).

3.2. Ejemplo motivador: Un contador de eventos de red

A modo de motivación, se plantea la necesidad de emular un monitor de eventos de tráfico. Se deberán manejar desde ECD++ dos variables IXA: una que provea información de la cantidad de ocurrencias de ciertos eventos de tráfico de interés, y otra que provea la capacidad de reiniciar dicho contador. Se las llamará *EventCounter* y *ResetCommand*

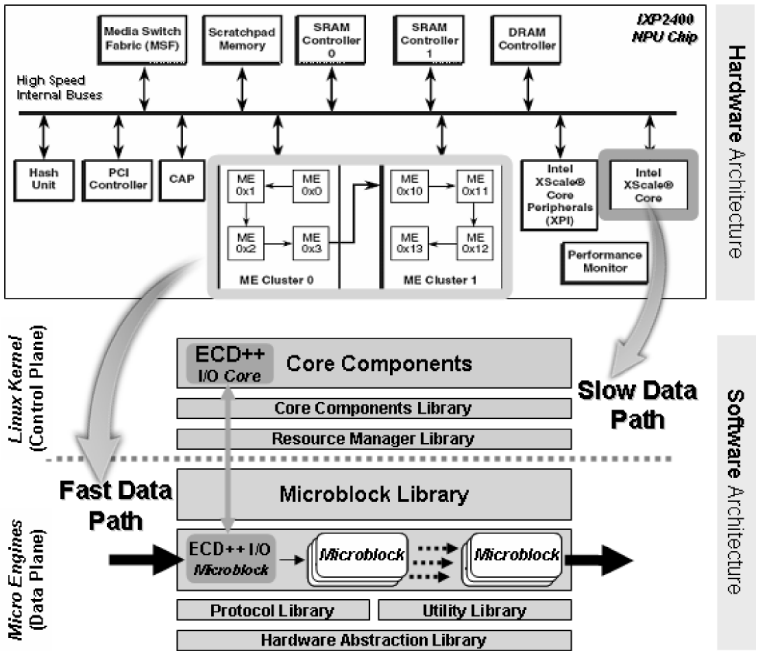

Figura 10: ECD++ embebido en un Intel IXP2400 NP

respectivamente. Cada envío de ResetCommand, además de reiniciar a cero a EventCounter, podrá transportar un código numérico para ser interpretado como un comando por el bloque de microcódigo ECD++MicroBlock (con el cual interacciona ECD++). El microcódigo deberá incrementar de alguna manera a EventCounter cada vez que acontezca un evento particular. A los efectos de este ejemplo, estos eventos se forzarán artificialmente, es decir, no estarán asociados a acontecimientos de tráfico real (como podría ser, por ejemplo, la llegada de un paquete de red corrupto). Se tendrá un archivo "IXA_EventCounter.ma", el cual modela el acoplado, siendo el resto los modelos atómicos (en éste caso uno sólo, "trafficEventCounter.cpp"). El funcionamiento de los modelos obedece a lo explicado en 2.1.

4. Implementación

En éste capítulo se explica, en 4.1, detalles de la contrucción del laboratorio que permitió integrar posteriormente la arquitectura IXA con ECD++. Luego se verá en 4.2 una herramienta de modelado visual que se usará en los ejemplos de aplicación. En 4.3 se muestra la estructura general del código utilizado, para luego en 4.4 entrar en detalle en cada submódulo implementado. En 4.5 se explican los detalles concernientes a la generación automatizada de código y su relación con ECD++. Por último, en 4.6 se muestran las facilidades que provee una extensión desarrollada para CD++Builder, basada en la presente Tesina.

4.1. Armado de un Laboratorio Virtual

La experimentación con el procesador IXP2400 y la arquitectura de referencia IXA ofrece una gran potencia y flexibilidad para desarrollar aplicaciones embebidas para control de red. Sin embargo, la instalación y puesta a punto de las numerosas herramientas y bibliotecas necesarias para obtener un laboratorio productivo consisten en tareas tediosas, muy propensas a fallas, que requiere muchos casos de la asistencia de expertos. La operatoria completa puede insumir semanas de trabajo. Esta situación atenta contra la replicación del entorno de experimentación y su adopción por parte de ingenieros, docentes y alumnos que deseen desarrollar aplicaciones basadas en modelos DEVS para ese escenario.

Los componentes de software provienen básicamente de los provistos por Intel y Radisys, involucrando tres sistemas operativos: Linux Montavista Embedded RT, Linux de propósito general y Windows. La instalación y ajuste de parámetros se basa de múltiples paquetes de software provistos por los fabricantes, representando un proceso complicado.

Como solución, se realizaron todas las instalaciones y configuraciones necesarias en *máquinas virtuales* fácilmente portables. Es decir, LinuxHOST y WindowsHOST son máquinas virtuales que ejecutan en una misma máquina física, la cual a su vez, puede tener enchufada la placa RadiSys ENP-2611 en un slot PCI (en nuestro caso utilizamos otra PC adicional, por carecer la primera, de espacio físico en el gabinete, para almacenar la placa). Esto genera un laboratorio preinstalado y totalmente replicable, eliminando prácticamente todo esfuerzo preliminar a desarrollar modelos ECD++ para IXP2400.

Cabe mencionar que la necesidad de WindowsHOST proviene del hecho de que el desarrollo de MicroCódigo para las MicroEngines de IXP2400 se realiza con una herramienta visual avanzada de desarrollo y depuración provisto por Intel (Intel Developer Workbench, ver [13]) que funciona bajo Windows XP.

Como se observa en la figura, la máquina virtual LinuxHOST incluye a ECD++ y a las nuevas herramientas presentadas en esta tesis.

Cuando se completa el diseño de un sistema embebido de control basado en DEVS, y ya se dispone del MicroCódigo que manejará los paquetes reales de red (obtenido usando Developer Workbench en WindowsHOST), ambos archivos binarios son "bajados" al procesador IXP2400 por medio de nuevos scripts para automatización de tareas. Luego, se está en condiciones de realizar ensayos en tiempo real y analizar los resultados por medio de los archivos de log generados por ECD++ en Linux Montavista, que son accesibles vía el Sevidor NFS montado en LinuxHOST. Los ejemplos de aplicaciones prácticas que se presentan posteriormente fueron desarrollados íntegramente utilizando el nuevo laboratorio virtual presentado en esta sección.

Con respecto al hardware. y además de la placa de red Radisys ENP-2611, se disponen de los siguientes dispositivos:

- Mikrotik RouterBoard RB750G (5 puertos Gigabit Ethernet),
- Dos Media Converter (MC) Trendnet TFC-1000MSC (1000BaseTX a 1000BaseSX Fibra Multimodo SC),
- Computadora personal con conexión RS-232 para instalar la placa,
- Computadora personal que ejecutará las máquinas virtuales utilizadas,
- Dos patchcords RJ45,
- Dos patchcords de fibra óptica 62,5/125μm, con conectorizado tipo MC/LC.

En la Figura 11 se puede observar cómo fueron conectados todos los dispositivos, para lograr una conexión lógica simplificada, que se observa en la Figura 12

La secuencia de arranque de la placa se puede resumir como sigue:

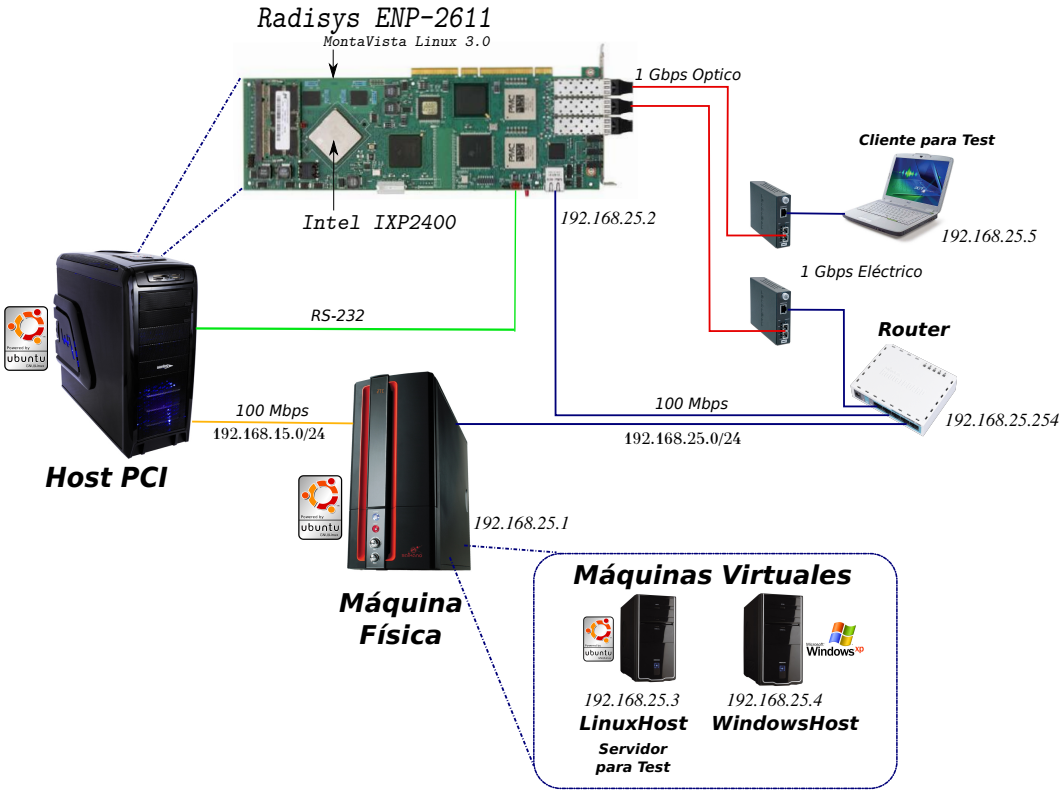

Figura 11: Disposición del hardware utilizado para el laboratorio virtual

- ENP-2611 posee un gestor de arranque Redboot. Cuando inicia, se automatizó mediante un pequeño script en Redboot, para que cargue un kernel de Linux Montavista 3.0 RT, el cual se encuentra en su memoria Flash (fue previamente grabado).
- Luego de iniciar dicho kernel, primero se solicitará una dirección IP por DHCP (Dynamic Host Configuration Protocol),
- luego se buscará el sistema de archivos principal (RootFS) mediante un directorio compartido por NFS (Network File System).

Éstas dos últimas funcionalidades se encuentran provistas por la máquina virtual Linux. Además, la misma dispone del SDK (Software Development Kit) de Intel y de Radisys, configurados con un compilador GCC cruzado para la arquitectura ARM de ENP-2611, y que permite compilar el código de ECD++ y ejecutarlo en la misma. La máquina

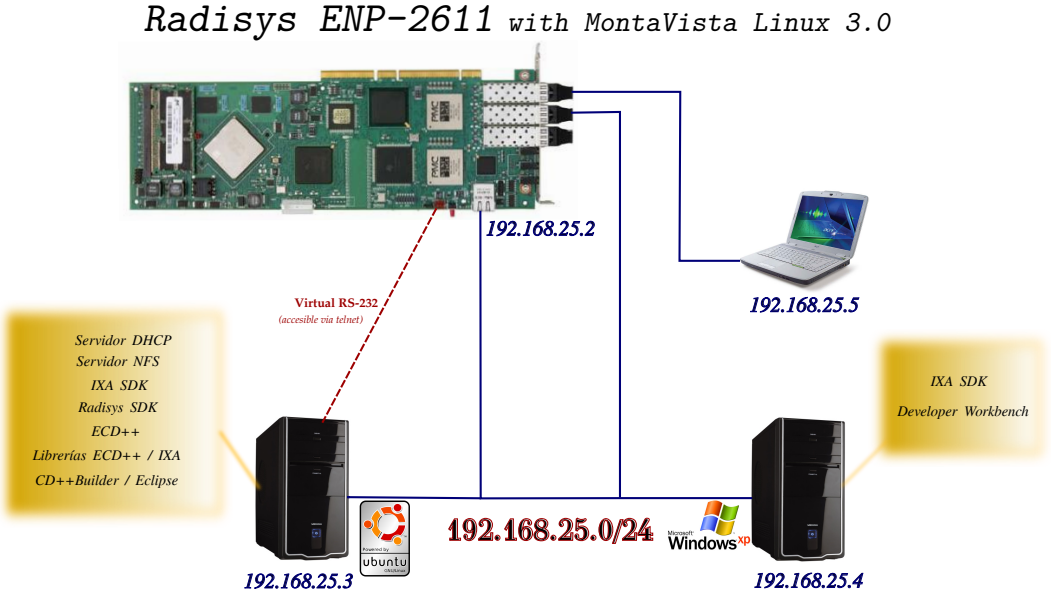

Figura 12: Conectividad en el laboratorio virtual

virtual Windows también posee el SDK de Intel, y permitirá programar bajo el framework de Intel Developer Workbench. Para detalles técnicos puntuales sobre el funcionamiento comentado, referirse al Apéndice A.3.

4.2. Herramienta Visual de Modelado basada en Eclipse

En ECD++ los modelos acoplados pueden definirse mediante un lenguaje declarativo de alto nivel, mientras que los modelos atómicos pueden definirse tanto declarativamente utilizando DEVS-Graphs o programáticamente utilizando C++. Algunas herramientas previas de ECD++ permiten crear modelos DEVS acoplados y atómicos gráficamente y visualizar los resultados de una simulación [25], pero poseen varias limitaciones: son desarrollos independientes que no utilizan interfaces estándar, son difíciles de extender, están desacopladas del simulador, entre muchos otros.

Acorde a las dificultades planteadas, se desarrolló la herramienta CD++Builder[3], un plugin para Eclipse[4] que resuelve los problemas anteriores, basado en el simulador ECD++. CD++Builder resuelve los problemas de usabilidad a través de asistentes estándar de Eclipse, diversos comandos, teclas de acceso rápido, cambio de vistas del

modelo, etc. Para los modelos atómicos basados en C++, los editores proveen una representación gráfica de la interfaz externa de los mismos. Para el análisis de los resultados de las simulaciones, se muestran animaciones de los envíos de mensajes entre modelos y los valores de los puertos de entrada y salida, lo que hace posible visualizar los resultados de modelos acoplados compuestos por modelos atómicos C++.

En la Figura 13 se muestra una pantalla del entorno de modelado y simulación.

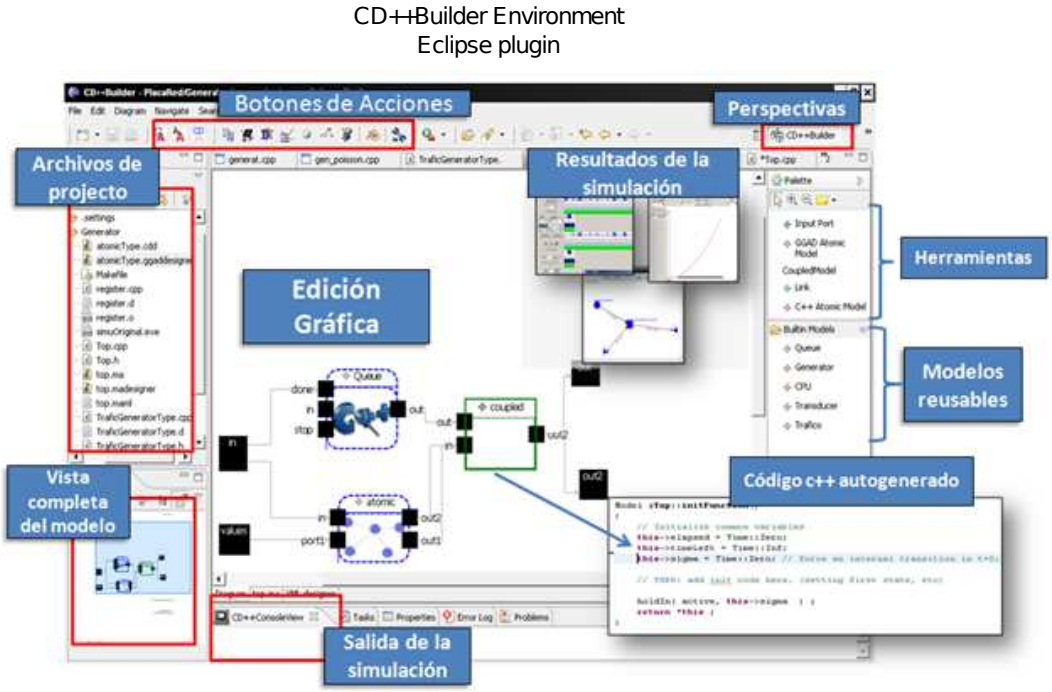

Figura 13: Entorno de Simulación CD++Builder para desarrollo de sistemas embebidos con ECD++.

Eclipse también puede ser extendido fácilmente para proveer un sitio de actualizaciones, que permite instalar y descargar actualizaciones para los plugins de un único punto accesible a todos los usuarios. Ésta herramienta será utilizada para ensayar y visualizar resultados en el ejemplo de aplicación de la sección 5.2, que presenta un control supervisorio de calidad de servicio implementado con ECD++.

4.3. Organización general de los submódulos

La comunicación se resuelve de modo transparente para el diseñador del modelo atómico, gracias a los módulos de la biblioteca *ECD++ I/O Library* para IXA que se muestran en la Figura 14. En dicha figura, un modelo atómico DEVS llamado *SW/HW Mapper* utiliza la infraestructura de comunicación desarrollada para compartir variables con el microcódigo *ECD++MicroBlock*. De este modo, pueden desarrollarse modelos tipo Mapper que manejen el acceso a distintas variables externas compartidas con IXA, encapsulando dicha funcionalidad, y separándola de la especificación dinámica de los modelos que definen el resto del sistema.

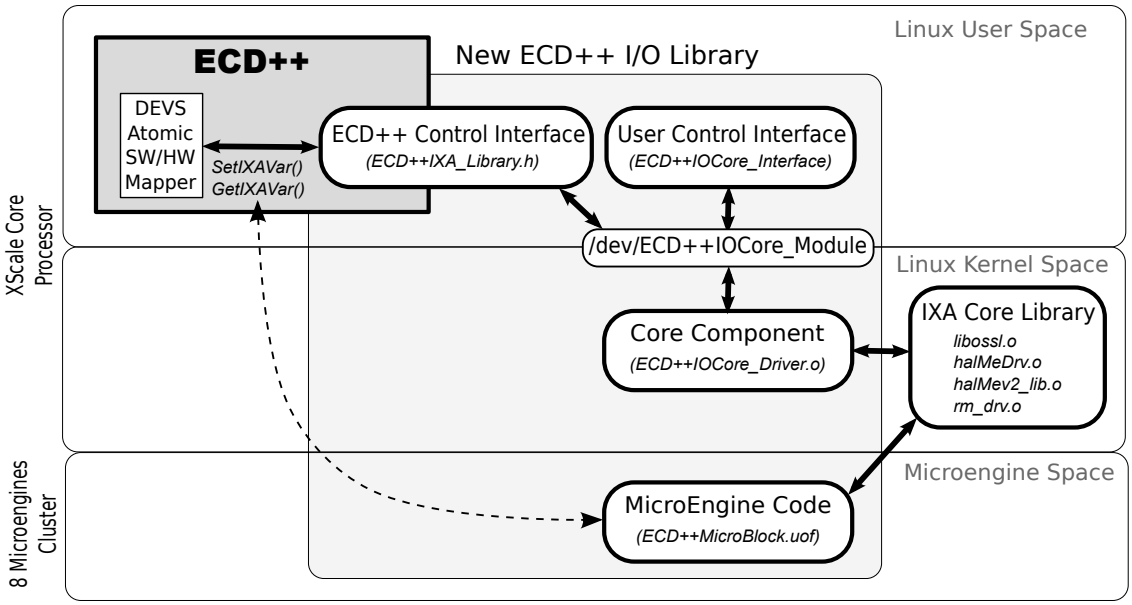

Figura 14: Nuevas bibliotecas ECD++ Input/Output para IXA

4.4. Estructura del directorio "IXA_Code"

Cada modelo DEVS en ECD++ que requiera utilizar las bibliotecas IXA, tendrá un subdirectorio denominado *"IXA_Code"*, el cual se encuentra organizado como se muestra en la Figura 15.

Dentro de *"IXA_Code"* mismo, se encuentran los códigos que son creados automáti-

Figura 15: Organización del directorio con bibliotecas IXA

camente por los generadores escritos en Perl, ubicados bajo el subdirectorio *"perl"*. Bajo el subdirectorio *"microcode"* se encuentran los microbloques que fueron escritos en μC (microcodeC) o en Assembler, compilados y enlazados en un archivo *".uof"*. Bajo el subdirectorio *"phys_control"* se encuentran los códigos necesarios para inicializar partes físicas de ENP-2611 (Bridge SPI3 y MACs PM3386). Bajo el subdirectorio *"scripts"* se encuentran scripts útiles que se copian automáticamente cuando se compila el directorio, y que sirven para determinados fines, los cuales se comentarán posteriormente. Por último, bajo

"share_modules" se encuentran los módulos compartidos utilizados en la arquitectura IXA, y que fueron descritos anteriormente en 2.7. En la Figura 14 se puede apreciar, para cada submódulo, cómo se encuentra dispuesto en la arquitectura general.

4.4.1. Cargador de microcódigo en hardware

En la arquitectura IXA, el Manejador de Recursos (RM) es probablemente el módulo más importante de todos, y el que más utilizaremos en éste componente de la biblioteca. El mismo provee acceso a las facilidades del sistema operativo, y administra la comunicación con los MicroEngines. En particular, es importante para la administración de la memoria, porque el XScale y los MicroEngines no usan el mismo espacio de direcciones. El RM tiene acceso a las tablas de traducciones para convertir entre el espacio de memoria virtual usado por el XScale y los espacios de direcciones físicos utilizados en los MicroEngines.

Las llamadas a `ix_rm_get_physical_offset()` y `ix_rm_get_virtual_address()` proveen dicha funcionalidad.

De la misma manera, `ix_rm_mem_alloc()` y `ix_rm_mem_free()` se encargan de la asignación y liberación de la memoria, debiendo especificar como parámetro, qué tipo de memoria se requiere (Scratch, SRAM, DRAM).

También se lo utilizará para cargar el código en cada uno de los MicroEngines, y arrancar y parar cada uno de los mismos. Las funciones utilizadas en éste caso serán `ix_rm_ueng_start()` y `ix_rm_ueng_stop()`.

Mediante el mismo, se usará una técnica que Intel denomina *patcheo de símbolos* para comunicar datos entre los MicroEngines y el XScale. Empleando ésta técnica, cada referencia simbólica en el microcódigo es reemplazada por su correspondiente valor constante. Ésto provee las ventajas de eficiencia y flexibilidad: por contener valores constantes, el código es más eficiente que el código que accede a valores de registros; y porque las constantes no son asociadas hasta que se cargue el código, lo hace más flexible que establecer constantes en tiempo de compilación (por lo tanto permite cambiar valores sin recompilar el código fuente). Una vez que el patcheo se encuentra completo y el microcódigo ha sido cargado en el MicroEngine, el código corre tan rápido como si el programador hubiese especificado una constante en el código fuente. En el microcódigo, se usará la sentencia

$$.import\_var \;\; CONSTANTE$$

lo cual declara *CONSTANTE* un nombre simbólico que será patcheado. Se puede encontrar una descripción más completa de las funciones mencionadas y muchas más, en [16].

El cargador de microcódigo en hardware se implementa en los archivos

`ECD++IOCore_Driver.c` y `ECD++IOCore_Driver.h`,

los cuales son generados automáticamente por

`gen_ECD++IOCore_Driver.pl` y `gen_ECD++IOCore_Driver_header.pl`

respectivamente, ambos bajo el directorio "IXA_Code".

4.4.2. Acceso desde espacio de usuario

Se implementó como acceso alternativo, una interfaz de usuario con funciones de control, lectura y escritura, para las variables compartidas entre ECD++ y los MicroEngines. El mismo código se crea por defecto en

`ECD++IOCore_Interface.c`

y es generado por

`gen_ECD++_IOCore_Interface.pl`

bajo el directorio "IXA_Code". Luego de ser compilado, su modo de uso es:

```
./ECD++IOCore_Interface [ show | set <IXA_Var> <value> ]
```

Es decir, el mismo permite establecer el valor de alguna variable en particular, o bien obtener su valor actual, mediante línea de comando del Linux Montavista que se encuentra corriendo en ENP-2611.

4.4.3. Biblioteca de Interfaz: Acceso desde ECD++

Las bibliotecas de interfaz desarrolladas permiten a cualquier modelo atómico de ECD++, acceder a variables en el espacio de los ME por medio de las funciones

`SetIXAVar()` y `GetIXAVar()`.

Se crean automáticamente dos archivos para ofrecer ambas funciones, denominados, por defecto,

`ECD++IXA_Library.h` y `ECD++IXA_Library.cpp`

los cuales son generados por

`gen_ECD++IXA_Library.pl` y `gen_ECD++IXA_Library_header.pl`,

respectivamente. Ambos se crean bajo el directorio en dónde se encuentran los modelos atómicos DEVS de ECD++, es decir, un nivel por encima del directorio "IXA_Code". El primero de ellos es el que incluirán los modelos DEVS en ECD++, para acceder a variables IXA, el cual consta básicamente del código que se observa en Código 1.

Código 1: Definición de biblioteca IXA

```
1  /* ECD++IXA_Library  */
2  /* IXA API for ECD++ */
3  /* WARNING! - This code was generated automatically and */
4  /* will be overwritten next time you compile ECD++. */
5
6  // Obtain IXA variable's value from hardware
7  int GetIXAVar(char *which);
8
9  // Set IXA variable's value from ECD++ to hardware
10 int SetIXAVar(char *which, char *value);
11 ...
```

Se mostrará su uso desde ECD++ posteriormente, en los ejemplos que se desarrollarán.

4.4.4. Interacción con el medio

Eventualmente, se puede requerir que ECD++ trabaje con tráfico real. Ésto implica inicializar adecuadamente distintos dispositivos de ENP-2611.

Controlador del Bridge SPI-3

El dispositivo controlador del bridge SPI-3 corre como Core Component en el XScale del IXP2400. Provee una interfaz de configuración y de status del mismo. No hay un

camino directo de datos para el componente, sin embargo, debe ser configurado correctamente para que los datos se muevan entre el IXP2400 y los dispositivos Gigabit Ethernet PM3386. Veremos posteriormente, que utilizaremos un controlador implementado con dos microbloques (no utilizaremos la parte de Core) para llevar los datos desde y hacia las MACs. La Figura 16 muestra la relación del controlador con los distintos componentes del sistema.

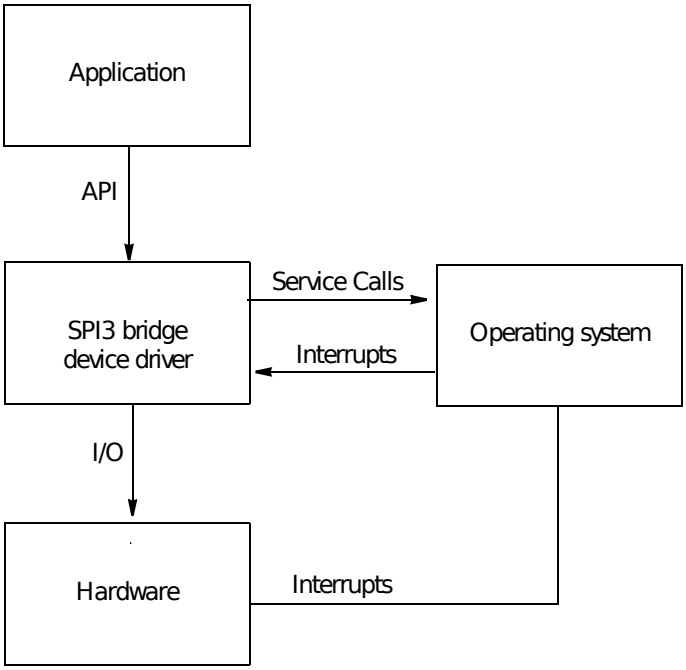

Figura 16: Interfaces del controlador SPI-3

La implementación del controlador en Linux es un dispositivo de caracteres (cargable como módulo) ubicado en ‘‘`/dev/spi3br`’’, que implementa funciones `open`, `close` e `ioctl`. Se puede observar en el Apéndice A.1, un extracto del código del controlador en el Código 11, en el cual se muestra la función para inicializar el dispositivo, y parcialmente la función para obtener la configuración del mismo, ambas que luego se utilizará en "*Inicializador de Bridge y MACs*".

Controlador de MACs PM3386/7

Análogamente al Bridge, el controlador del dispositivo PM3386 corre en el XScale del IXP2400. Además de proveer una interfaz para configuración y status, se establece un ca-

mino de comunicación de datos, que se implementa a traves de la arquitectura provista por Intel, usando microbloques y opcionalmente Core-Components. Análogamente al Bridge SPI-3, la figura 16 aplica también a éste controlador (es decir, "PM3386 device driver" se ubica en la figura, el mismo lugar que "SPI3 bridge device driver").

La implementación del controlador en Linux es un dispositivo de caracteres (cargable como módulo) ubicado en ``/dev/pm338x0'' y ``/dev/pm338x1'', que implementa funciones `open`, `close` e `ioctl`. Se puede observar un extracto del código del controlador en el Código 12 dentro del Apéndice A.1, en el cual se muestra la función para acceder al dispositivo, y parcialmente la función para inicializar el mismo, ambas que también se utilizarán en nuestro Inicializador de bridge y MACs.

Inicializador de bridge y MACs

Se implementó un intérprete de órdenes para manejar los dispositivos en cuestión, por facilidad, al mismo tiempo. Mediante el mismo, se puede inicializar y detener el bridge SPI-3, las MACs PM3386/7 y ver el estado actual en ejecución de los mismos. El archivo ejecutable se genera con nombre ``Initialize'' y su modo de uso es:

```
./Initialize [ start|stop|showmacstats|showmacs|spi3bridge|testints ]
```

Los comandos que muestran el estado, se pueden utilizar con propósito de depuración, ya que brindan información útil de transmisión y recepción en cada puerto (tipos de tramas enviadas, errores en cada una, etc). En el Código 2 se puede observar el fuente del mismo. La salida estándar de dichos comandos se puede observar en el Apéndice A.4.

Código 2: real_traffic_linux.c

```
1  /* Driver for handling initalization of Bridge and PHYs */
2  /* of ENP2611. Use in conjunction with ECD++ models. */
3
4  #include <fcntl.h>
5  #include <unistd.h>
6  #include <sys/ioctl.h>
7  #include <enpv2_types.h>
8  #include <sys/mman.h>
9
10 #include "uclo.h"
```

```
11 #include "hal_mev2.h"
12 #include "halMev2Api.h"
13
14 #define SLOW_PORT_FIX  FALSE
15 #undef USE_WORKBENCH
16
17 extern void TestSPI3Interrupts(void);
18
19 void StaticRouteStart( void ){
20 #if( SLOW_PORT_FIX == TRUE )
21   /* put slowport in corect mode if not already */
22   printf("Putting Slow port in correct mode...\n");
23   spfix();
24 #endif
25   /* init and start spi3br */
26   printf("Starting SPI3 Bridge...\n");
27   StartSpi3br();
28   /* init and start macs */
29   printf("Initiating and starting MACs...\n");
30   StartMacs();
31   printf("Started MACs and Bridge.\n");
32   // Example that we will be using on current work.
33   printf("Port0->Port1, Port1->Port2, Port2->Port0\n");
34 }
35
36 void StaticRouteStop( void ){
37   StopMacs();
38   StopSpi3br();
39 }
40
41 int main(int argc, char **argv){
42   if (argc != 2) {
43           printf("Usage: Initialize [ start | stop \
44                   | showmacstats | showmacs | spi3bridge \
45                   | testints ]\n");
```

```
46              return 0;
47      }
48      if (strncmp(argv[1], "start", 5) == 0) {
49              StaticRouteStart();
50      } else if (strncmp(argv[1], "stop", 4) == 0) {
51              StaticRouteStop();
52      } else if (strncmp(argv[1], "showmacstats", 12) == 0) {
53              ShowMacStats();
54      } else if (strncmp(argv[1], "showmacs", 8) == 0) {
55              printf("\n------- MAC 0 -------\n");
56              ShowMac(0);
57              printf("\n\n------- MAC 1 -------\n");
58              ShowMac(1);
59      } else if (strncmp(argv[1], "spi3bridge", 10) == 0) {
60              ShowSpi3br();
61      } else if (strncmp(argv[1], "testints", 8) == 0) {
62              TestSPI3Interrupts();
63      } else {
64              printf("Invalid parameter\n");
65      }
66      return 0;
67  }
```

4.5. Generación automática de código

Dado que las interfaces de comunicación con IXA proveen servicio a un conjunto de variables IXA definibles por el usuario, las bibliotecas deben regenerarse conforme se modifica la elección (agregado, remoción o renombrado) de dichas variables. Acorde a ésta necesidad, se diseñó un mecanismo para la generación parametrizable de código de bibliotecas, capaz de detectar automáticamente las variables de bajo nivel (no accesibles directamente sino a través de los servicios provistos por la arquitectura IXA) elegidas por el modelador (interpretando el archivo de acoplamiento *".ma"*).

Como se mencionó y observó anteriormente en la Figura 15, dentro del subdirectorio "perl" encontramos todos los archivos, escritos en lenguaje *Perl*, que, en base a las variables

definidas en el archivo de acoplamiento de ECD++, generarán el código necesario para comunicarse con dichas variables, a las que llamamos *"variables IXA"*.

El funcionamiento de los mismos está dado, básicamente, por un ejecutable que interpreta, a partir de una configuración dada (ver más abajo), el archivo general de modelos ".ma", buscando variables que hayan sido declaradas en forma especial, de la siguiente manera:

```
variableName : <initValue> %%useIVAVar[<ME_Number_List>]
```

Esto significa que, `<variableName>` tendrá un valor inicial de `<initValue>`, y que será una variable de tipo IXA que se utilizará en los MicroEngines `<ME_Number_List>`, siendo `<ME_Number_List>` una lista de números de MicroEngines, separados por coma.

El código que interpreta el archivo de ECD++ se puede observar en el Código 3.

Código 3: parse_dotma.pl

```
1  #!/usr/bin/perl
2  # Script that parses ".ma" of an ECD++ model.
3  # It extracts declared IXA variables in order to generate
4  # them on low level communication with ENP2611.
5  # An example of lines relevant to us:
6  # "miVariable : initValue %%useIXAVar[ME_Number]"
7
8  $CONFIG="./IXA-Generator_config.pm";
9  require $CONFIG;
10 die "Specified file does NOT exist. Bye bye.\n" if (! -e $MODEL);
11
12 my @data;
13 open(FILE, $MODEL);
14 ...
15 while(<FILE>){
16   my($line) = $_;
17   chomp($line);
18   if ($line =~ /^(\w+)\s*:\s*(\d+) %%useIXAVar(\[(\S+)\])?$/){
19     $name = $1;
20     $value = $2;
```

```
21      $valid_mes = "";
22      @me_numbers = split(',', $4);
23      foreach $me (@me_numbers){
24        if ( ($me ge 0) and ($me lt $ME_NUMBER) ){
25          $valid_mes .= ",$me";
26        }
27        else {
28          print "WARNING! Variable \"$name\" has an invalid \
29                  ME_Number. Ignoring supposed MicroEngine \
30                  \"$me\"!!!\n";
31        }
32      }
33      $valid_mes =~ s/^,//;
34  ...
35      push @data,"$name-$value-$valid_mes";
36    }
37  }
38  ...
39  system("./codeGenerator.pl @data");
40  close(FILE);
41  exit;
```

El resto del código generador, se lo puede observar en el Apéndice A.5 si se desea.

Cabe destacar que, además de los archivos generadores de código, existe un archivo de configuración, que aplica y puede modificar ligeramente la generación mencionada, de acuerdo a ciertos parámetros requeridos por el microcódigo a cargar. El mismo archivo se nombra como ‘‘`IXA-Generator_Config.pm`’’, el cual se aprovechará para describir en detalle en el Ejemplo 2 en la sección 5.2, que se desarrolla posteriormente.

4.5.1. Flujo automatizado

Cada modelo atómico que desee interactuar con las ME debe cumplir con tres requisitos:

1. **Declarar** como Variable IXA a un parámetro del modelo atómico (explicado ante-

riormente).

2. **Incluir** la biblioteca ECD++IXA Library.h en el archivo .cpp del modelo atómico. Esto permite invocar las funciones

   SetIXAVar(variable IXA,...) y GetIXAVar(variable IXA,...).

3. **Tener asociado** un bloque de microcódigo, que ejecuta en espacio de MicroEngines, el cual **importe la variable** Variable IXA desde el espacio del Core.

En la Figura 17 se resume los pasos necesarios para la obtención de un código que utilice variables IXA de bajo nivel.

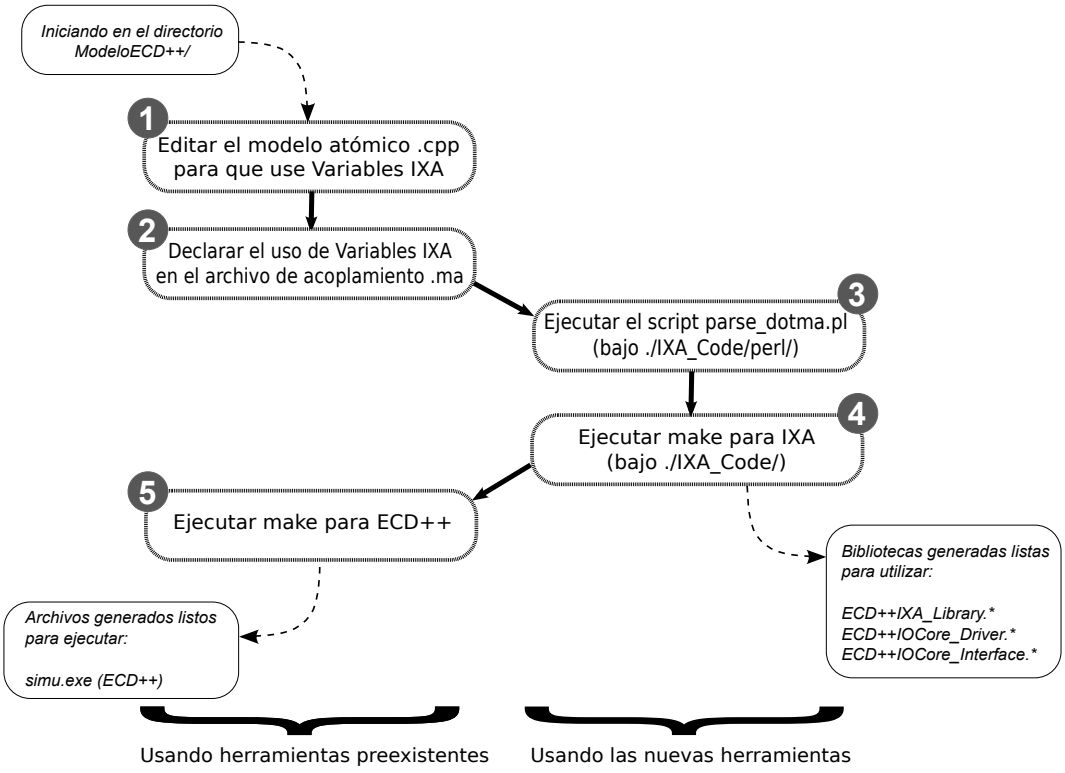

Figura 17: Generación automática de bibliotecas de interfaz.

En la misma se observa que se requieren 5 pasos desde la elección del nombre de una variable IXA, hasta la obtención de toda la infraestructura de bibliotecas necesarias para

hacerla accesible desde un modelo atómico DEVS. Los pasos 1 y 2 conciernen exclusivamente a ECD++, y fueron recién mencionados como el primer y segundo requisito para obtener un modelo con dichas características. Estos pasos pueden resolverse con un editor de texto. Los pasos 3 y 4 implican generación automática de bibliotecas, y requieren del uso de las nuevas herramientas desarrolladas. El paso 5 es de exclusiva injerencia de ECD++, es decir, forma parte del proceso usual de compilación del simulador, independientemente del uso o no de variables IXA en algún modelo atómico.

Este mecanismo automatiza y facilita enormemente la tarea de declaración de variables entre ECD++ y las MicroEngines, ocultando un gran número de detalles de implementación de bajo nivel.

La repetición del flujo de pasos descrito puede realizarse tantas veces como sea necesario siguiendo un proceso típico de desarrollo incremental incorporando nuevas funcionalidades al modelo DEVS y/o al código de las MicroEngines, utilizando probablemente nuevas variables IXA y validando resultados experimentales con aquellos verificados previamente en un contexto puramente simulado (sin interacción con el tráfico de red real).

Sin embargo, para alcanzar una condición tal que sea posible completar automática y exitosamente los pasos 3, 4 y 5, deben respetarse una cantidad de requisitos previos de modo estricto: configuraciones de variables de entorno, convenciones de nombres, ubicación de archivos en directorios, ejecución de los comandos desde lugares específicos, etc. Si bien la estructura de directorios y denominación de archivos pretende ser clara y autoexplicativa, este proceso puede ser propenso a errores, sobre todo para los usuarios sin experiencia previa.

Para solucionar este inconveniente, se tomó ventaja de la característica de extensibilidad del entorno visual de desarrollo CD++Builder presentado anteriormente, incorporándole dos interfaces básicas de asistencia al usuario: una interfaz de configuración de variables, nombres y ubicaciones, y una interfaz de ejecución automática de los pasos 3, 4 y 5. Esta herramienta avanzada se presenta en la sección a continuación.

4.6. Asistencia mediante Plugin IXA

En forma paralela al desarrollo de esta Tesina, se colaboró con un equipo independiente de trabajo que desarrolló un plugin extra para CD++Builder para trabajar con

IXA, apoyado en el hecho de que la generación automática de bibliotecas presentadas en este Capítulo puede ser parametrizada y ejecutada tanto por un usuario como por un algoritmo. Como se mencionó antes, con este nuevo plugin se incorporan dos tipos de interfaces básicas: de configuración y de generación automática de bibliotecas, tanto para minimizar la probabilidad de cometer errores manualmente como para agilizar el proceso de experimentación.

Con respecto a la interfaz de configuración, consta de varias pantallas integradas en el entorno Eclipse, las cuales ofrecen establecer los parámetros necesarios para poder compilar para IXA. En la Figura 18 podemos observar una de ellas, en donde se puede establecer la ubicación de varios directorios importantes, como son el directorio NFS, el directorio donde se encuentra el simulador ECD++, etc.

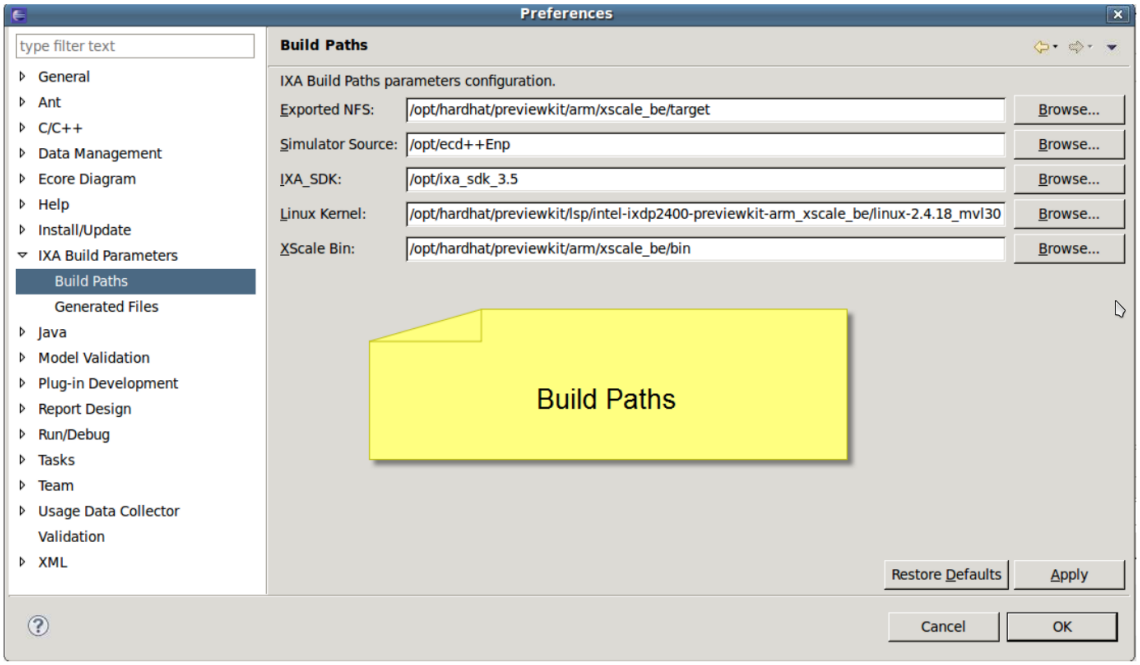

Figura 18: Una de las pantallas que ofrece el entorno para establecer la configuración de compilación contra IXA.

Con respecto a la interfaz de ejecución, se incorpora un nuevo botón “Build for IXA” (ver Figura 19a), el cual ofrece una interfaz gráfica solicitando únicamente el nombre del

modelo DEVS acoplado (.ma) para el cual desea construirse el simulador ejecutable en el IXP2400.

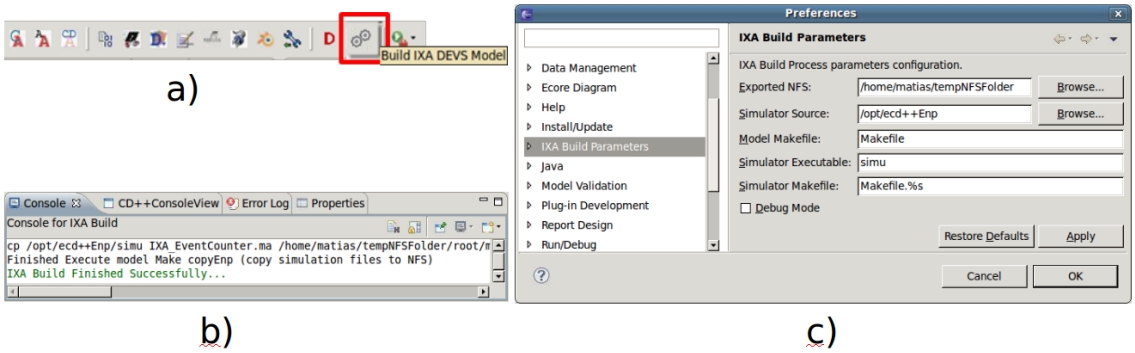

Figura 19: Nueva GUI de CD++Builder para el proceso de contrucción de ECD++. a) botón de compilación para IXA. b) consola de progreso de construcción IXA. c) preferencias de configuración IXA.

Puede observarse en la Consola de Salida (dentro del entorno gráfico de CD++Builder, Figura 19b) la evolución de los pasos principales e intermedios que va ejecutando "Build for IXA" hasta obtener el producto final, incluyendo la copia del binario al directorio NFS que es accedido por IXP2400. Luego de este paso sencillo, consistente en una acción simple, el simulador puede ser invocado para ejecutar embebido en el procesador de red.

5. Ejemplos de Aplicaciones

Se mostrará la aplicación de la biblioteca y el procedimiento explicados hasta aquí, en primera instancia, con un ejemplo práctico sencillo, emulando un monitor de eventos de tráfico. Posteriormente, se verá un ejemplo más sofisticado, dónde el modelo controlador interactúa con tráfico real.

5.1. Contador de Eventos

En el modelo desarrollado, se manejarán desde ECD++ dos variables IXA: una que provea información de la cantidad de ocurrencias de ciertos eventos de tráfico de interés, y otra que provea la capacidad de reiniciar dicho contador. Se las llamará `EventCounter` y `ResetCommand` respectivamente. Cada envío de ResetCommand, además de reiniciar a cero a EventCounter, podrá transportar un código numérico para ser interpretado como un comando por el bloque de microcódigo ECD++MicroBlock (con el cual interacciona ECD++). El microcódigo deberá incrementar en tres a EventCounter cada vez que acontezca un evento particular. A los efectos de este ejemplo, estos eventos se forzarán artificialmente, es decir, no estarán asociados a acontecimientos de tráfico real (como podría ser, por ejemplo, la llegada de un paquete de red corrupto).

5.1.1. Modelo ECD++

Se define un modelo atómico DEVS simple llamado `eventCounter` (una instancia de trafficEventCounter.cpp) con un puerto de salida llamado `counter`, el cual emite valores hacia el puerto de salida del modelo acoplado *top*, el de mayor jerarquía del sistema. El atómico eventCounter se parametriza con 4 variables que afectan su dinámica, dos de las cuales se declaran como variables IXA utilizando el comentario `useIXAVar[0]` indicando que se compartirán con un microcódigo ejecutando en la MicroEngine 0. Este modelo consultará cada `pollingPeriod` unidades de tiempo a la MicroEngine por el valor actualizado de la cantidad de eventos de tráfico acontecidos. El valor es consultado mediante la variable IXA `EventCounter`, que se inicializa con valor 0. El parámetro `counterLimit`, inicializado en `30000000`, indica que cuando EventCounter supere dicho valor, el modelo enviará el comando de reinicio de contador ResetCommand con valor 7, el cual será interpretado por

el microcódigo que recomenzará la cuenta de eventos desde 0.

En el Código 4 se puede observar el archivo de acoplamiento ECD++ donde se utilizan éstas variables IXA.

Código 4: EventCounter.ma

```
[top]
components : eventCounter@trafficEventCounter
out : counterStatus
Link : counter@eventCounter counterStatus

[eventCounter]
pollingPeriod : 00:00:02:000
counterLimit : 30000000
EventCounter : 0 %%useIXAVar[0]
ResetCommand : 7 %%useIXAVar[0]
```

En el Código 5 en C++, se observan las funciones de inicialización, de transición externa e interna y de salida, para el modelo en cuestión.

Código 5: Definición del modelo atómico trafficEventCounter.cpp

```
...
#include "trafficEventCounter.h"
/****************
Integration with IXA
****************/
#include "ECD++IXA_Library.h"
...
// *** Constructor
trafficEventCounter::trafficEventCounter(const
                  std::string &name) : Atomic(name),
counter(addOutputPort("counter"))
{}

// *** Initialization Function
Model &trafficEventCounter::initFunction(){
 std::string period( MainSimulator::Instance().getParameter( \
```

```
17                        description(), "pollingPeriod" ) ) ;
18   if( period == "" ){
19     period = "00:00:01:000"; // Defaults to 1 second between polls
20    }
21   this->pollingPeriod = period;
22   this->counterLimit = str2Int( \
23    MainSimulator::Instance().getParameter( description(), \
24                                            "counterLimit" ) ) ;
25    // IXA Variable
26   this->EventCounter = str2Int( \
27    MainSimulator::Instance().getParameter( description(), \
28                                            "EventCounter" ) ) ;
29    // IXA Variable
30   this->ResetCommand = str2Int( \
31    MainSimulator::Instance().getParameter( description(), \
32                                            "ResetCommand" ) ) ;
33  ...
34   this->state = Sleeping;
35   // programs itself for the first poll
36   holdIn(active, this->pollingPeriod);
37  ...
38  }
39
40  // *** External Transition Function
41  Model &trafficEventCounter::externalFunction( \
42                              const ExternalMessage &msg ){
43  ...
44  }
45
46  // *** Internal Transition Function
47  Model &trafficEventCounter::internalFunction( \
48                              const InternalMessage &msg ){
49  ...
50   switch (this->state) {
51     case Sleeping:
```

```
52 ...
53         this->currentCounter = GetIXAVar("EventCounter");
54         this->state =  Notifying;
55         // Programs itself for returning
56         // inmediately the obtained data
57         holdIn( active, 0);
58         break;
59
60    case Notifying:
61         if ( this->currentCounter > this->counterLimit ) {
62 ...
63             SetIXAVar("ResetCommand", strResetCommand);
64         }
65         this->state = Sleeping;
66         // Programs itself for the next poll
67         holdIn(active, this->pollingPeriod);
68         break;
69 ...
70 }
71
72 // *** Output Function
73 Model &trafficEventCounter::outputFunction( \
74                              const InternalMessage &msg ){
75 ...
76   switch (this->state) {
77 ...
78    case Notifying:
79         // send the observed information
80         sendOutput(msg.time(), counter, this->currentCounter);
81 ...
82 }
```

La biblioteca incluida en la línea 6 provee la interfaz de acceso a SetIXAVar() y GetIXAVar(). En la línea 11 se declara el puerto de salida `counter` por el cual se emitirán los valores leídos desde la MicroEngine. Durante la inicialización del modelo, en las líneas 26

y 30 se toman desde `IXA_EventCounter.ma` los valores iniciales de las variables IXA.

El modelo puede estar en dos estados definidos por `this->state`: Sleeping y Notifying. En la fase Sleeping el modelo espera a que se cumpla el tiempo pollingPeriod hasta enviar la próxima consulta de EventCounter. En la línea 34 el modelo se inicia en Sleeping, y en la línea 36 lo hace esperando pollingPeriod unidades de tiempo. Cuando se agota el tiempo de espera en estado Sleeping, en la línea 53 se lee el nuevo valor del contador con `GetIXAVar(‘‘EventCounter’’)` y en la línea 54 se cambia el estado a Notifying. En la línea 57 se fuerza una transición interna inmediata para poder ejecutar la función de salida y así emitir por el puerto de salida el valor almacenado en currentCounter. En la fase Notifying (de duración cero, un estado transitorio) primero se emite el último valor del contador (línea 80, función outputFunction) y luego, en la transición interna (línea 61), se decide si hay que enviar o no un comando de reset. Si es así, en la línea 63 se invoca `SetIXAVar(‘‘ResetCommand’’, strResetCommand)`, lo cual escribe la variable IXA ResetCommand haciendo que el microcódigo reaccione al cambio. El valor de strResetCommand deberá ser 7 acorde a lo especificado en el archivo .ma, e importado en la línea 30 del modelo atómico. Luego en las líneas 65 y 67 se cambia nuevamente al estado Sleeping durante pollingPeriod unidades de tiempo, recomenzando el ciclo. Este modelo no posee entradas, por lo que no hace uso de la función de transición externa.

Los resultados de la ejecución embebida y en tiempo real del sistema IXA_EventCounter se muestran en la Figura 20.

Se muestra un lapso de 10 segundos de ejecución, en los cuales el modelo atómico eventCounter detecta en 2 oportunidades que el contador de eventos de las MicroEngines supera el límite establecido por counterLimit. En dichas oportunidades el sistema envía un comando de reinicio, lo cual se refleja en una caída del contador por debajo del umbral en el período de medición subsiguiente. Se valida así el modelo y su interacción efectiva con el hardware especializado de bajo nivel.

5.1.2. Detalles del microcódigo

El microcódigo desarrollado en assembler, utiliza un único MicroEngine (número 0) y un único thread (número 0). El mismo importa las dos variables mencionadas anteriormente. Mientras dure la ejecución, se incrementará en uno indefinidamente a EventCounter, a

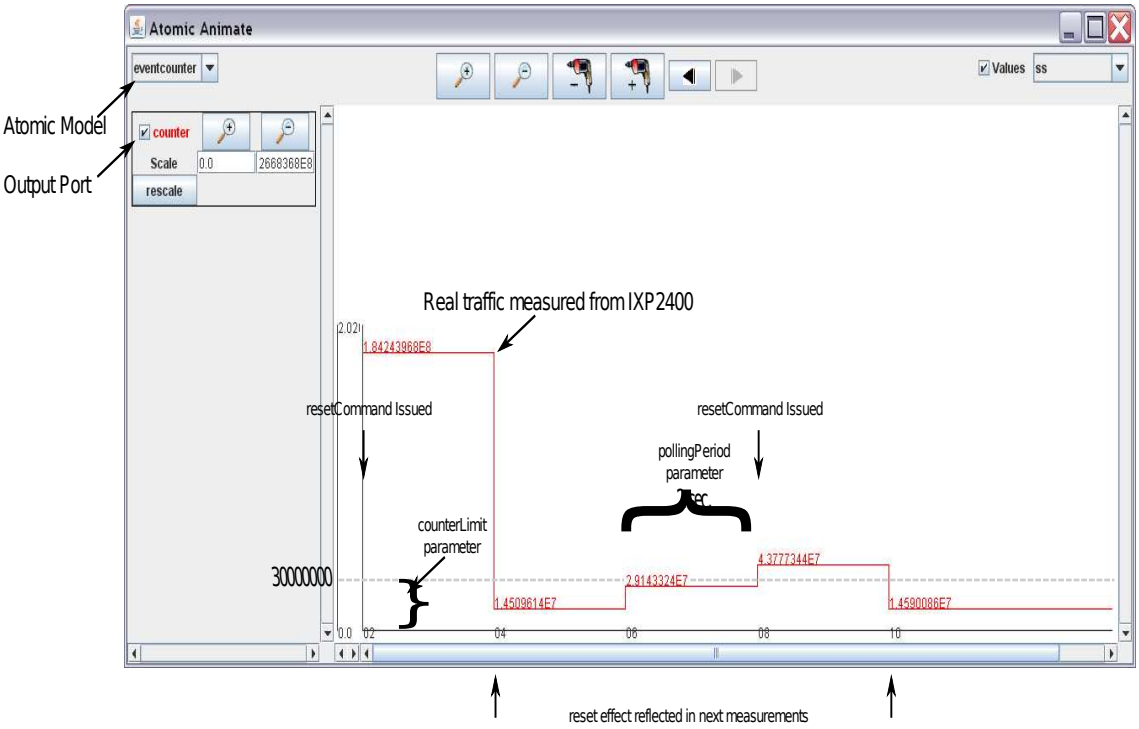

Figura 20: Resultado de ejecución de modelos ECD++ en IXP2400. Interacción con MicroEngines en tiempo real.

menos que se escriba un valor distinto de 0 en ResetCommand. Se puede observar dicho comportamiento en el Código 6, en el cual se muestra el código principal que se carga en el MicroEngine (el mismo requiere otros archivos, pero son cargados posteriormente).

Código 6: EventCounter.uc

```
1 /*
2     A simple code that counts
3     virtual events for IXP2400 MicroEngine.
4 */
5
6 // This gives us some basic macros
7 #include "xbuf.uc"
8 #include "stdmac.uc"
9
```

```
10 #ifdef  USE_IMPORT_VAR
11     .import_var EVENTCOUNTER
12     .import_var RESETCOMMAND
13 #else
14     #define EVENTCOUNTER      0
15     #define RESETCOMMAND     200
16 #endif /* USE_IMPORT_VAR */
17
18 .reg eventCounter_address
19 immed32(eventCounter_address, EVENTCOUNTER)
20 .reg resetCommand_address
21 immed32(resetCommand_address, RESETCOMMAND)
22
23 .reg aa
24 .reg bb
25 .reg cc
26 .reg read $reg_result_read
27 .reg write $reg_result_write
28 .init $reg_result_read  0
29 .init $reg_result_write 0
30 .reg $clear_flag
31 .init $clear_flag 0
32 .init aa 0
33 .init bb 0
34 .init cc 0
35
36 // This is the main part of the program
37 .begin
38   // Need a signal for SRAM accesses
39   .sig sram_sig
40   //Only do this on 1 thread
41   .if(ctx() == 0)
42     .while (1)
43       ctx_arb[voluntary]
44       // Check clear flag in SRAM
```

```
45        sram[read, $clear_flag, resetCommand_address, 0,1],\
46          ctx_swap[sram_sig]
47        // If clear bit is set (!= 0), clear the counter
48        .if ($clear_flag != 0)
49          // Clear the sram and all other reg.
50          immed32(aa,0)
51          immed32(bb,0)
52          immed32(cc,0)
53          immed32($reg_result_write,0)
54          sram[write, $reg_result_write, resetCommand_address, \
55            0, 1],ctx_swap[sram_sig]
56          sram[write, $reg_result_write, eventCounter_address, \
57            0, 1],ctx_swap[sram_sig]
58        .endif
59        // Still read from memory...
60        sram[read, $reg_result_read, eventCounter_address, 0,1],\
61          ctx_swap[sram_sig]
62        /* Do some trivial things... BEGIN */
63        move(aa,$reg_result_read)    ; copy old result from SRAM to aa
64        // a = a + 3
65        alu[cc,aa,+,1]      ; cc = aa + 1;
66        alu[bb,cc,+,1]      ; bb = cc + 1;
67        alu[aa,bb,+,1]      ; aa = bb + 1;
68        move($reg_result_write, aa)    ; $reg_result = aa;
69        /* Do some trivial things... END */
70        // Write to SRAM
71        sram[write, $reg_result_write, eventCounter_address, 0, 1],\
72          ctx_swap[sram_sig]
73      .endw
74    .endif
75    ctx_arb[kill]
76  .end
77  nop
```

5.1.3. Detalles del código C generado automáticamente

En el Código 7 se puede observar un extracto del código del cargador de microcódigo, generado por las herramientas desarrolladas. El fuente completo podremos encontrarlo en el Apéndice A.2.

En la línea 2 se encuentra el archivo que contiene el microcódigo que se cargará en hardware, que se obtiene de la configuración de la generación de código de las bibliotecas (ver 5.2.3). Luego se ve en la línea 10 las variables obtenidas de ECD++, las cuales luego serán utilizadas en la línea 52 para asignarles un bloque de memoria, y posteriormente en las líneas 102, 107 y 111, para patchear los símbolos hacia los MicroEngines. En las líneas 42 y 47 se encuentran las funciones que se encargan de crear una *"free buffer list"* (ver 5.2.3) y obtener la información con respecto a la misma (`hwFreeListInfo` es una estructura que contiene toda la información asociada con una free buffer list, como ser direcciones de inicio de las memorias de meta-datos y datos, tamaños de los elementos de ambos, etc). Dicha información será utilizada cuando se patchean los símbolos a las MicroEngines. Por último, en las líneas 64, 67, 76 y 84 se encuentran, respectivamente, las funciones del RM para detener y resetear todos los ME, establecer la imagen del microcódigo para los MEs desde un archivo, cargar el microcódigo en el almacén de los MEs y arrancar los MEs específicos habilitando los contextos que se deseen.

Código 7: ECD++IOCore_Driver.c para el modelo IXA_EventCounter

```
1  ...
2  static char Uof_file[512] = "EventCounter.uof";
3  /* Local procedures                          */
4  static int patch_microblocks(ix_buffer_free_list_info);
5  ...
6  /* List of free buffers */
7  ix_buffer_free_list_handle hwFreeList = 0;
8  /* Pointers to various run-time data structures allocated by RM */
9  /* They are taken automatically from ECD++ variables            */
10 void *ResetCommand;
11 void *EventCounter;
12 ....
13 static int MEL_ioctl(struct inode *inode, struct file *fp, \
```

```
14                          unsigned int cmd, unsigned long buf){
15 ...
16     switch (cmd) {
17         case GET_RESETCOMMAND:
18 ...
19             break;
20         case SET_RESETCOMMAND:
21 ...
22             break;
23 ...
24         default:
25             return INVALID_CMD;
26     }
27     return 0;
28 }
29
30 int init_module(void){
31     ix_error err;
32     ix_buffer_free_list_info hwFreeListInfo;
33 ...
34     /* Initialize Intel's Resource Manager */
35     err=ix_rm_init(0);
36     if (err != IX_SUCCESS) {
37         printk("Error: ix_rm_init failed\n");
38         return -1;
39     }
40
41 /* ================== START of memory allocation ==========*/
42     /* Allocate a free buffer list */
43     err = ix_rm_hw_buffer_free_list_create(NUM_BUFFERS,
44                         sizeof(ix_hw_buffer_meta), BUF_SIZE,
45                         BUF_SRAM_CHAN, BUF_DRAM_CHAN, &hwFreeList);
46 ...
47     /* Read freelist info (it will be needed later) */
48     err = ix_rm_buffer_free_list_get_info(hwFreeList, \
```

```
49                                                  &hwFreeListInfo);
50  ...
51      printk("<1>Trying to allocate memory for ME now...\n");
52      err = ix_rm_mem_alloc(IX_MEMORY_TYPE_SRAM, 0, \
53                                  RESETCOMMAND_SIZE, &ResetCommand);
54      if (err != IX_SUCCESS)
55              panic("ix_rm_mem_alloc failed for: ResetCommand\n");
56      /* Clear the buffers */
57      bzero(ResetCommand, RESETCOMMAND_SIZE);
58  ...
59  /* ========================= END of memory allocation ========================*/
60
61      /* Reset the MicroEngines */
62      printk("%s: Resetting all MicroEngines\n", \
63              ME_LOADER_DRIVER_NAME);
64      ix_rm_ueng_reset_all();
65      /* Get the microcode from the UOF file */
66      printk("%s: Setting ucode\n", ME_LOADER_DRIVER_NAME);
67      err = ix_rm_ueng_set_ucode(Uof_file);
68      if (err != IX_SUCCESS)
69          panic("ix_rm_ueng_set_ucode failed\n");
70      /* Patch the microcode symbols before actually */
71      /* loading microcode.                          */
72      if (patch_microblocks(hwFreeListInfo) < 0)
73          return(-1);
74      /* Load microcode into MicroEngines */
75      printk("%s: Loading uCode\n", ME_LOADER_DRIVER_NAME);
76      err = ix_rm_ueng_load();
77      if (err != IX_SUCCESS)
78          panic("ix_rm_ueng_load failed\n");
79      /* Start the assigned MicroEngines */
80      for (i=0; i<ME_NUM; i++) {
81          if ((ME_MASK>>i) & 0x1) {
82              printk("%s: Starting ME%i\n", \
83                      ME_LOADER_DRIVER_NAME, i);
```

```
84              err = ix_rm_ueng_start(ME_ID(i), CONTEXT_MASK);
85              if (err != IX_SUCCESS)
86                  panic("ix_rm_ueng_start failed for ME %i\n", i);
87          }
88      }
89      return 0;
90  }
91  ...
92  /*************************************************************/
93  /* Patch the microcode symbols before actually loading microcode */
94  /*************************************************************/
95  int patch_microblocks(ix_buffer_free_list_info hwFreeListInfo){
96  ...
97      ix_uint32        memChan;
98      ix_uint32        offset;
99      ix_imported_symbol importSymbols0[2];
100
101     importSymbols0[0].m_Name = "EVENTCOUNTER";
102     err = ix_rm_get_phys_offset(EventCounter, NULL, &memChan, \
103                                 &offset, NULL);
104     if (err != IX_SUCCESS)
105         panic("ix_rm_get_phys_offset failed for %s\n",
106                 importSymbols0[0].m_Name);
107     importSymbols0[0].m_Value = ME_SRAM_ADDR(offset, memChan);
108 ...
109     /* Patchs for ME 0 */
110     printk("Now trying to patch ME %i!!!\n", ME_ID(0));
111     err = ix_rm_ueng_patch_symbols(ME_ID(0),2,importSymbols0);
112 ...
113 }
```

5.2. Control Supervisorio de Calidad de Servicio basado en Medición de Tasa

Se diseña un Sistema de Control Supervisorio de Calidad de Servicio (QoS) que acepta políticas globales y monitorea la tasa de tráfico (Traffic Rate - TR) de un flujo real de paquetes. Dependiendo de las políticas globales y del estado de TR, el sistema envía información de control actualizada a los algoritmos de bajo nivel que deciden y ejecutan el descarte de paquetes. La información de monitoreo y de control desde y hacia estos algoritmos de bajo nivel se realiza a través de puertos de entrada-salida de ECD++.

La metodología incremental propuesta plantea como primera etapa la verificación del comportamiento del sistema completamente simulado en ECD++ bajo un escenario simplificado, con modelos DEVS ejecutando en el XScale Core pero sin interactuar aún con las ME.

En la Figura 21 (izquierda) se muestra un diagrama conceptual en bloques de los modelos que componen la aplicación de control. Los modelos QoS Actuator y Traffic Sensor en el nivel de baja velocidad se encargan de enviar comandos de control y sensar el estado de TR, respectivamente. Estos bloques se comunican con sus contrapartes en el nivel de alta velocidad (o Packet Processing system), que consisten en los modelos QoS Shaper y Metering System. Habiendo verificado la funcionalidad básica del QoS Controller en este escenario simplificado utilizando tráfico sintético generado con modelos DEVS, se procede a la etapa de validación del controlador, comunicando los niveles de baja velocidad (modelos DEVS que se conservan intactos) con el hardware real de alta velocidad (MiroEngines). En la Figura 21 (derecha) el hardware real de procesamiento de paquetes reemplaza al conjunto de modelos que emulaban sus funciones (Generator, Shaper, Pipeline, Meter, Consumer).

5.2.1. Modelo ECD++

Se asumirá que existe un algoritmo de bajo nivel ejecutando en las MicroEngines con la capacidad de aplicar técnicas Control de Admisión (AC) de descarte de paquetes para garantizar una longitud media de cola. Dicha longitud es un parámetro de operación que se recibe como acciones de control provenientes del Control Supervisorio. Existirá un

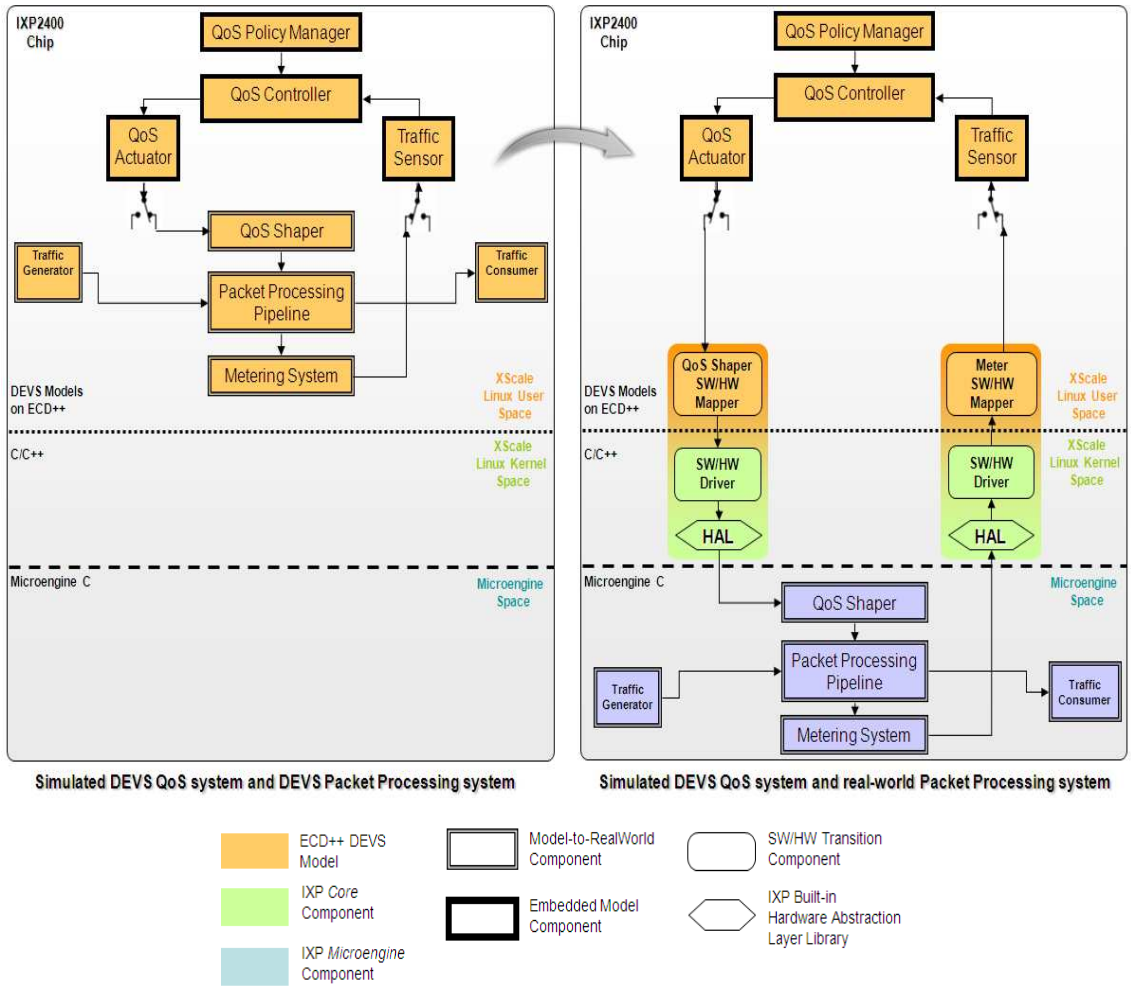

Figura 21: Diagrama Conceptual: Transición desde una simulación embebida autónoma (izquierda) hacia una simulación embebida con Hardware In The Loop (derecha).

microbloque especial corriendo en un MicroEngine que recibirá dichos comandos. A su vez, las ME tienen la capacidad de hacer accesible al Control Supervisorio la información actualizada de la cantidad de paquetes transmitidos a cada instante.

El sistema de Calidad de Servicio (QoS) que se desea diseñar con ECD++ debe enviar comandos de control al AC indicando la nueva longitud de cola que se desea obtener (acorde a la política configurada en el controlador). La Figura 22 muestra una representación de este modelo a modo de una máquina de estado. Los 4 estados reflejan la condición del

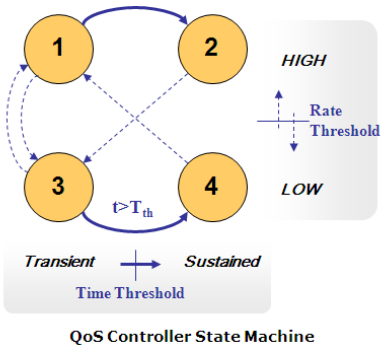

Figura 22: Representación del modelo de control presente en trafficQoSControl, como máquina de estados.

sistema de procesamiento de paquetes, combinando la métrica TR con la cantidad de tiempo *ininterrumpido* T (persistencia temporal) en la cual se sostiene un valor dado de TR. En la Figura 23 se muestra el diseño visual, con CD++Builder, del sistema de control completo en ECD++ listo para interactuar con las ME, es decir, el diseño práctico correspondiente a la etapa conceptual a la derecha de la Figura 21.

El controlador usa un umbral de tasa de tráfico $rateThreshold$ para clasificar las condiciones de tasa (alta o baja). Inicialmente, si se sensa $TR > rateThreshold$ se pasa al estado 1 y el tiempo T comienza a incrementarse desde cero. Si en cambio $TR <= rateThreshold$, el controlador pasa al estado 3. Se define el umbral de tiempo $sustainedThreshold$ para distinguir entre 2 posibles valores de persistencia de la tasa TR: Transient y Sustained. Cuando T cruza el umbral $sustainedThreshold$, el modelo evoluciona desde 1 a 2 o desde 3 a 4 dependiendo si el sistema se encontraba en un nivel de TR alto o bajo. Cuando la tasa de tráfico cruza el umbral $rateThreshold$, se reinicia T a cero y el estado toma el valor 1 o 3, según corresponda.

Ambos umbrales se definen como parámetros del modelo atómico trafficQoSControl, como puede observarse en el Código 8, en donde se configuran $rateThreshold = 5$ paquetes/segundo (línea 19) y $sustainedThreshold = 6$ segundos (línea 20).

En el mismo lugar se observan los parámetros

- $transHighCommand = 13$ (línea 21),
- $transLowCommand = 20$ (línea 22),

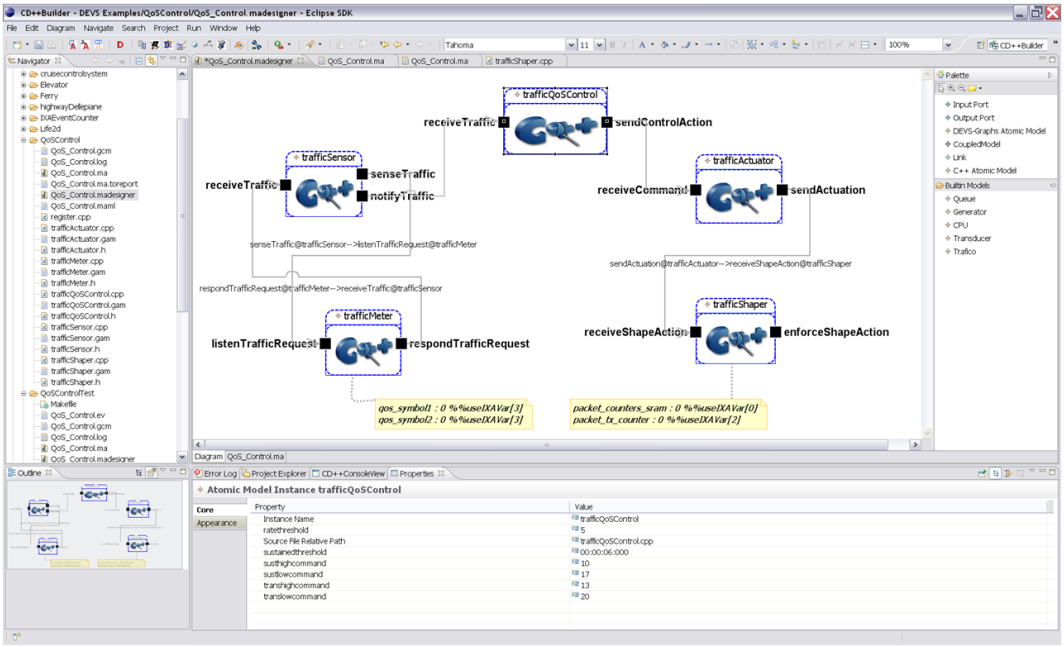

Figura 23: Modelado mediante herramienta avanzada de edición de modelos para ECD++. Modelo QoS Control para IXP2400.

- $sustHighCommand = 10$ (línea 23),
- $sustLowCommand = 17$ (línea 24).

Análogamente, fueron definidas en CD++Builder.

Código 8: QoS_Control.ma

```
1  [top]
2  components :     trafficSensor@trafficSensor
3                   trafficMeter@trafficMeter
4                   trafficActuator@trafficActuator
5                   trafficQoSControl@trafficQoSControl
6                   trafficShaper@trafficShaper
7  Link :   respondTrafficRequest@trafficMeter
8           receiveTraffic@trafficSensor
9  Link :   senseTraffic@trafficSensor
10          listenTrafficRequest@trafficMeter
11 Link :   notifyTraffic@trafficSensor
```

```
12          receiveTraffic@trafficQoSControl
13 Link :   sendControlAction@trafficQoSControl
14          receiveCommand@TrafficActuator
15 Link :   sendActuation@TrafficActuator
16          receiveShapeAction@TrafficShaper
17
18 [trafficQoSControl]
19 rateThreshold : 5
20 sustainedThreshold : 00:00:06:000
21 transHighCommand : 13
22 transLowCommand : 20
23 sustHighCommand : 10
24 sustLowCommand : 17
25
26 [trafficSensor]
27 pollingPeriod : 00:00:01:000
28 pollTimeOut    : 00:00:02:500
29 debugFile : debugfile
30
31 [trafficMeter]
32 dataFetchDelay : 00:00:00:050
33
34 [trafficShaper]
35 dataFetchDelay : 00:00:00:55
36 packet_counters_sram : 0 %%useIXAVar[0]
37 packet_tx_counter : 0 %%useIXAVar[2]
38 qos_symbol1 : 0 %%useIXAVar[3]
39 qos_symbol2 : 0 %%useIXAVar[3]
```

Los valores anteriores son los que deberán enviarse hacia el AC cada vez que el controlador experimente un cambio en su máquina interna de estados, para los estados 1, 2, 3 y 4 respectivamente. El encargado de enviar efectivamente dichos comandos es el bloque trafficActuator, el cual desconoce la diferencia entre un modelo DEVS estándar, y uno que tenga la capacidad de comunicarse con las MicroEngines. Por ello, se vale del bloque trafficShaper, el cual declara las variables IXA correspondientes para poder invocar la

función SetIXAVar(), y que gracias a las bibliotecas desarrolladas, permitirá establecer los valores de las variables de control que se encuentran en espacio de direcciones de memoria de los MicroEngines. En la siguiente sección se analiza con más detalle la estructura del microcódigo.

Siguiendo los pasos descritos en secciones anteriores de la presente Tesina, se generó el binario para el sistema QoS_Control incorporando las nuevas bibliotecas para dialogar con las MicroEngines, y se lo portó al IXP2400 para realizar ensayos de validación. Para realizar los mismos, se utilizó un generador de tráfico llamado *Packeth*, el cual se configura para enviar ráfagas de paquetes, como se observa en la Figura 24.

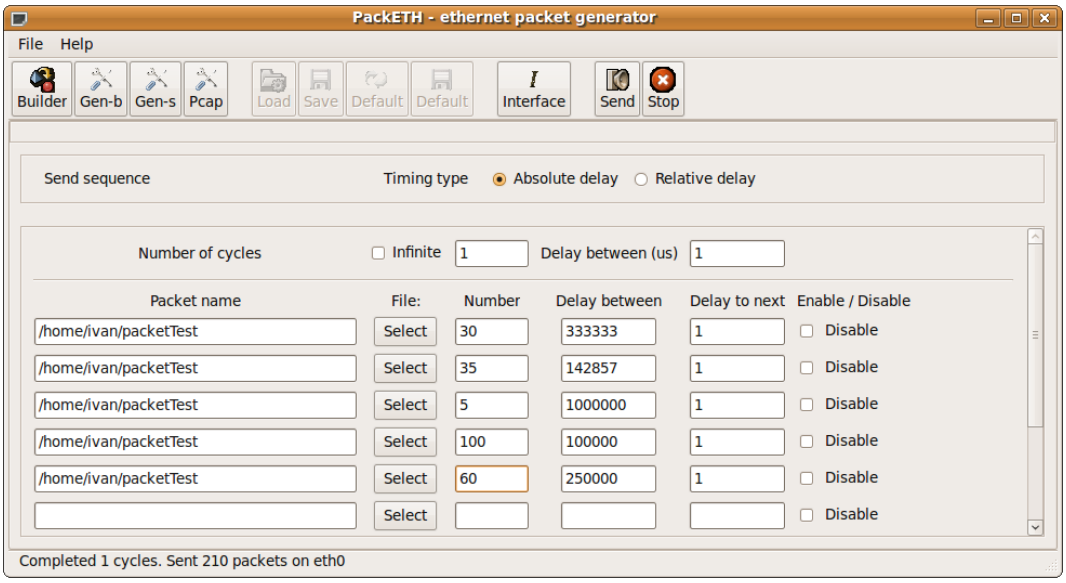

Figura 24: Configuración de envío de paquetes hacia ENP-2611.

Es decir, se usa la siguiente secuencia:

- 3 paquetes/seg. durante 10 seg. (fila 1, 30 paquetes con 333333μs de retardo c/u),
- 7 paquetes/seg. durante 5 seg. (fila 2, 35 paquetes con 142857μs de retardo c/u),
- 1 paquete/seg. durante 5 seg. (fila 3, 5 paquetes con 1000000μs de retardo c/u),
- 10 paquetes/seg. durante 10 seg. (fila 4, 100 paquetes con 100000μs de retardo c/u),

- 4 paquetes/seg. durante 15 seg. (fila 5, 60 paquetes con 250000μs de retardo c/u).

El resultado de los mismos se muestra en la Figura 25. Allí puede verificarse el nivel de tasa promedio TR medido por trafficSensor/trafficMeter cada 1 segundo (secuencia temporal superior, puerto receiveTraffic). Esta medición proviene del tráfico real de paquetes fluyendo por la placa de red, provenientes del generador mencionado anteriormente. Acorde a lo diseñado, puede validarse la secuencia de comandos enviados hacia las MicroEngines (secuencia temporal inferior, puerto receiveShapeAction). En la secuencia temporal del puerto receiveCommand se denotan los estados que va a asumiendo la máquina de estados de trafficQoSControl.

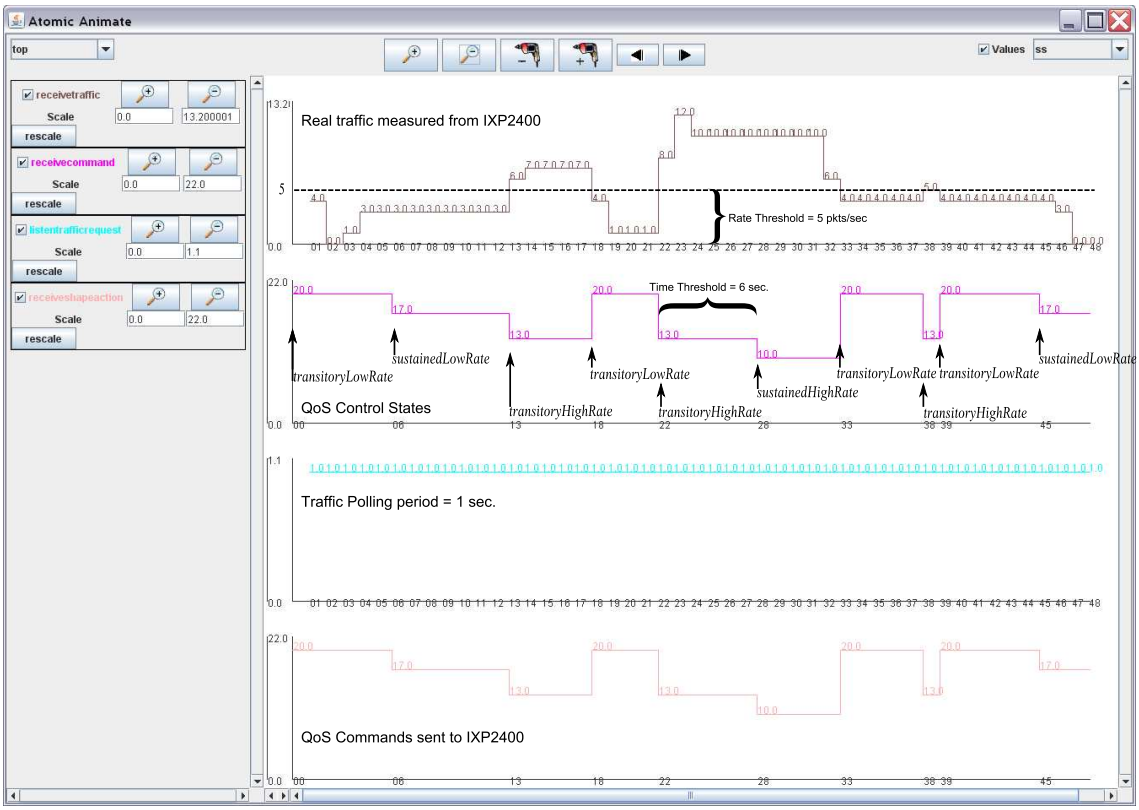

Figura 25: Resultados de la simulación en tiempo real con tráfico real en modo Hardware In The Loop.

Se observa el comportamiento esperado para el Control Supervisorio, reaccionando con el envío de comandos correcto tanto ante los cambios de niveles de tráfico sensado que

cruzan el umbral de tasa (5 paquetes/segundo), como ante la permanencia del tráfico en un nivel dado por más tiempo del establecido por el umbral de permanencia (6 segundos).

5.2.2. Detalles del microcódigo

La estructura del microcódigo empleado se compone de cuatro microbloques, los cuales son:

- Microbloque de Recepción: Se utilizó el microbloque RX de ingreso de Intel, el cual es invocado cuando llega un *mpacket* (la unidad fija en que se divide un paquete para almacenarse y transferirse internamente, configurable a 64, 128 o 256 bytes). El mismo debe copiar un mpacket desde el Búfer de Recepción (RBUF) en el Media Switch Fabric hacia un búfer en memoria. Si el mpacket es el primero de un nuevo paquete, el microbloque RX asigna un búfer. Cuando el paquete entero ha sido puesto en dicha memoria, se encarga de pasar el búfer al próximo microbloque en el pipeline, para su procesamiento.
- Microbloque de Procesamiento: Se utilizó un microbloque que, simplemente, copia lo que recibe desde el microbloque RX, hacia el microbloque TX, de una manera que siempre incrementa en 1 el puerto por el que sale. Es decir, si se recibe un paquete por el puerto físico 0, se transmitirá por el 1; si se recibe por el 1, lo hará por el 2; y finalmente si se recibe por el 2, lo transmitirá por el 0.
- Microbloque de Transmisión: Se utilizó el microbloque TX de transmisión de Intel, el cual es invocado cuando un paquete llega desde un microbloque anterior. Un microbloque de egreso desempeña dos tareas principales: envía el paquete desde la memoria hacia el puerto físico, y libera el búfer de memoria que se mantenía con el paquete. Al igual que con el ingreso, el hardware de egreso de bajo nivel sólo entiende mpackets. Por lo tanto, éste microbloque divide el paquete en una secuencia de mpackets para ser transferidos (copiando cada mpacket desde la memoria hacia el Búfer de Transmisión (TBUF) en el MSF).
- Microbloque ECD++: Éste microbloque importa dos variables desde el XScale, mediante las cuales ECD++ envía comandos de control, y estarán destinadas en un futuro, a un uso para el control del comportamiento de los bloques de RX y TX.

La Figura 26 muestra el esquema anterior.

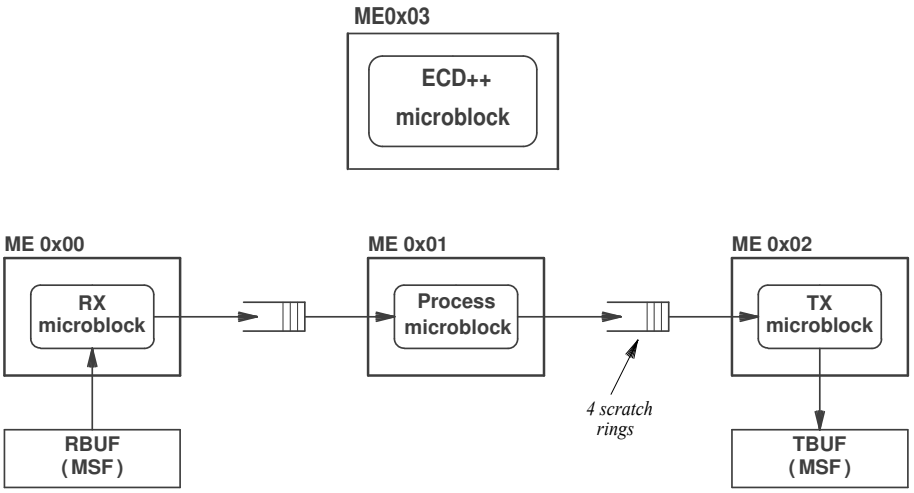

Figura 26: Organización del microcódigo empleado.

5.2.3. Archivo de configuración utilizado en la generación de bibliotecas

Como se mencionó anteriormente, la generación de bibliotecas IXA para ECD++ es altamente configurable mediante un archivo, denominado ‘‘`IXA-Generator_config.pm`’’, el cual se puede ajustar para determinados microbloques que requieran una configuración específica y no estándar. Se programó la generación de las bibliotecas mediante el uso de éste archivo con la idea de que, si el microbloque no tiene dichas necesidades, no hará falta configurar las variables de patcheo en absoluto (funcionará sin problemas la configuración por omisión).

En el Código 9 se puede observar el archivo utilizado para la generación automática de bibliotecas del ejemplo en curso. El mismo servirá para exlicar las configuraciones más relevantes.

Código 9: IXA-Generator_config.pl

```
1 ##########################################################################
2 ############ ECD++/IXA Library Configuration ##########################
3 ##########################################################################
4
```

```
5  ### Generator related config ###
6  # If DEBUG = 1, verbose output messages will be printed
7  # when generating code.
8  $DEBUG = 0;
9  # The model to be parsed
10 $MODEL = "../../QoS_Control.ma";
11 # Current Working Directory
12 $CWD = "/opt/ECD++Enp/models/QoS_Control/IXA_Code/perl";
13 # Directory where ECD++ resides. IXA Dynamic Libraries
14 # (which can be included in ECD++ code) will be
15 # generated here.
16 $IXA_LIB_OUTPUT_DIR = "/opt/ECD++Enp/models/QoS_Control";
17 # Directory where IXA code (kernel modules and userspace control)
18 # is generated. It should be under "IXA_Code" ECD++ subdir.
19 $IXA_CTL_OUTPUT_DIR = "$IXA_LIB_OUTPUT_DIR/IXA_Code";
20 ...
21 ### Generated-code related config ###
22 # Total number of MicroEngines available on board.
23 $ME_NUMBER = 8;
24 # This is the name of microcode to be loaded into MicroEngines
25 $UOF_FILE = "ECD++Microblocks.uof";
26 # This defines which MicroEngines to start.
27 # (0x01 = 0000 0001)
28 # Note that it does not need to be the same number that the
29 # number of MEs we are patching.
30 $ME_MASK = "0x0F";
31 # This defines which contexts do we enable (per ME)
32 # 255 --> enable all contexts
33 $CONTEXT_MASK = "255";
34 # Maybe it is necessary to fix some variable name before patching.
35 # Default is to add "_BASE" to variables declared on ECD++.
36 # Here you can change that behaviour for a particular variable.
37 %SUFFIX = (
38 'buf_free_list0' => '',
39 'free_list_id' => '',
```

```
40 );
41 # Maybe you want to specify additional variables sizes.
42 # If you don't declare any size for a variable, could be
43 # for 2 things: or you are using defaults, which means that
44 # is a variable that was declared on ECD++
45 # (in that case default size is 4 bytes),
46 # or could be because of a "hidden" variable (see later).
47 # Last case means you cannot GET or SET the value.
48 %SIZES = (
49 'packet_counters_sram' => 48,
50 'packet_tx_counter' => 48,
51 );
52 # Here you can define "hidden variables", from the
53 # point of view of ECD++.
54 # Use same format that parser generates:
55 # <variable_name>-<initValue>-<me_numbers>
56 @HIDDEN_VARS = ( 'dl_rel-0-0,2', 'buf_sram-0-0,2',
57                  'buf_free_list0-0-0,2', 'free_list_id-0-0' );
58 # Here you can set a specific order for variables that are
59 # going to be patched against a particular MicroEngine.
60 # If ommited, there is no order (random).
61 # Maybe there will be another ones to be patched against
62 # same ME apart from these, but order (starting at @[0])
63 # is guaranted *only* on variables declared below.
64 $VAR_ORDER_IN_ME[0] = [ 'buf_free_list0', 'buf_sram', 'dl_rel' ];
65 $VAR_ORDER_IN_ME[2] = [ 'buf_free_list0', 'buf_sram', 'dl_rel' ];
66 # Maybe it is necessary to get a custom memory offset,
67 # then enabling to get the value of the symbol
68 # to be patched. Default is to get
69 # ix_rm_get_phys_offset of variable being patched.
70 %CUSTOM_OFFSET = (
71 'buf_free_list0' => 'NO_CALC',
72 'buf_sram' => 'hwFreeListInfo.m_pMetaBaseAddress',
73 'dl_rel' => 'hwFreeListInfo.m_pDataBaseAddress',
74 'free_list_id' => 'NO_CALC',
```

```
75 );
76 # Value that will be patched to symbol. Default is to calculate
77 # ME_SRAM_ADDR(offset, memChan) (see macro definition on .c).
78 %VALUES = (
79 'buf_free_list0' => 'hwFreeListInfo.m_FreeListInfo',
80 'dl_rel' => 'offset - ((importSymbols0[1].m_Value *
81                         hwFreeListInfo.m_DataElementSize) /
82                         hwFreeListInfo.m_MetaElementSize)',
83 'free_list_id' => 'hwFreeListInfo.m_FreeListInfo1',
84 );
85 # Don't delete or Perl will annoy.
86 1;
```

Antes de comentar las líneas de la configuración anterior, se explican brevemente algunos detalles concernientes al funcionamiento de la asignación de memoria y búffers en IXP2400 que ayudarán a enteder la misma.

El mecanismo de búffers para paquetes del SDK de Intel asigna un conjunto de búffers de tamaño fijo en DRAM, usando un bloque grande y contiguo de dicha memoria. Para lograr un procesamiento de alta velocidad, IXP2XXX se basa en un apoyo por hardware para asignar memoria cuando llega un paquete al microbloque de ingreso. Utiliza un mecanismo del tipo FIFO (Firt-In-First-Out) que, a pesar de que los búffers se encuentren en DRAM, está asociado con la memoria SRAM (es decir, el hardware controlador SRAM implementa las operaciones FIFO). Para usar un FIFO para manejar una lista libre de búffers de paquetes, el software establece una correspondencia uno-a-uno entre items en la lista de SRAM y los búffers en DRAM. La Figura 27 muestra dicha correspondencia.

Por lo tanto, cuando se solicita búffers de paquetes en DRAM, el software asigna una lista contigua de elementos en SRAM. Ésta cola en SRAM se denomina *"free (buffer) list"*. Dicho ésto, volvemos al comentario de la configuración de la generación de bibliotecas para el ejemplo en curso.

En la línea 25 se especifica el nombre del archivo que contiene el microcódigo para cargarse en los MicroEngines, probablemente compilado sobre el entorno Windows utilizando Developer Workbench.

En la línea 30 se configura una máscara de bits, la cual indicará al generador de

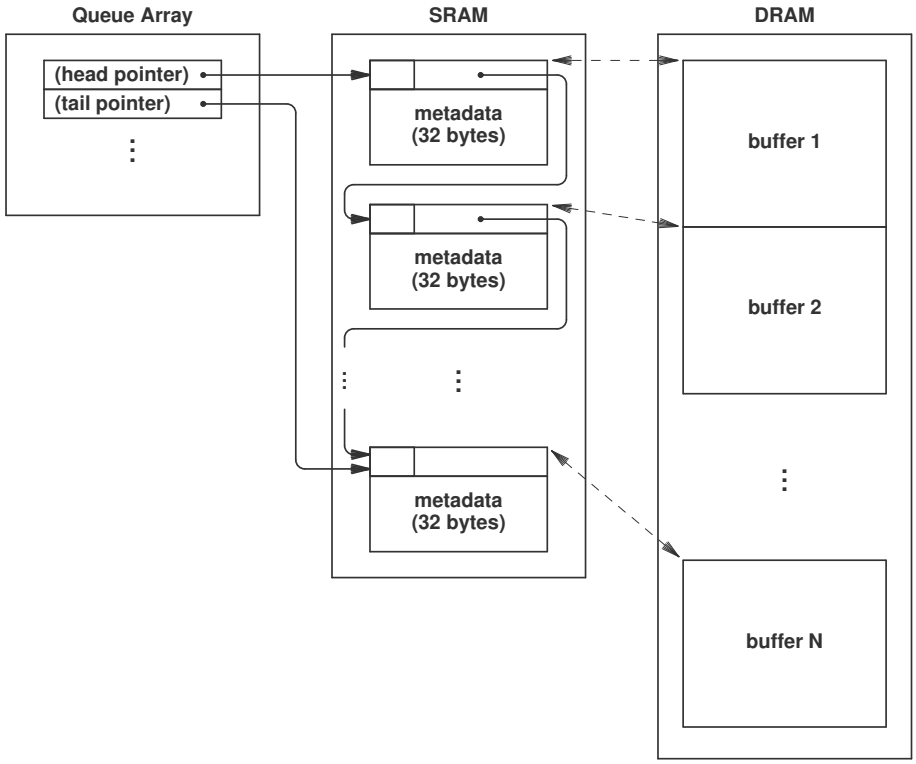

Figura 27: FIFO implementada en SRAM con Q-Array, el cual es parte del controlador SRAM, y el mapeo uno-a-uno con búffers en DRAM.

código, cuál de todos los MicroEngines arrancar. La misma se compone de dos números hexadecimales, la cuál un "1" indicará que se usará el MicroEngine correspondiente a la posición del mismo bit, y "0" lo contrario. A modo ejemplo, indicando el número $0x0F$ (que equivale a 00001111) implica que se usarán los MicroEngines 0, 1, 2 y 3.

En la línea 33 se indica un número en decimal de 8 bits, que indica cuántos contextos (threads) habilitar por MicroEngine (se pueden configurar a los MicroEngines para que usen los 8 threads, o bien habilitar sólo 4; en éste caso se ejecutan sólo los números pares de threads y los impares permanecen inactivos).

En la línea 37 se pueden forzar sufijos de variables de bajo nivel específicos. Esto es, por defecto, al definir una variable en ECD++, se le agrega el sufijo "_BASE" al nombre de la misma, para ser patcheada en los MicroEngines. Seteando una variable en ésta línea, se fuerza el sufijo al que se desee. En éste caso se utilizó para variables ya definidas en

todo el código de recepción y transmisión, que tenían definido dicho nombre.

En la línea 48 se puede definir un tamaño particular de memoria a reservar por el RM de Intel. Normalmente, no se deberán definir tamaños personalizados, en éste caso es necesario para poder acceder a las variables de los microbloques de recepción y transmisión de Intel, los cuales utilizan segmentos de memoria contiguos de 48 bytes (4 contadores - paquetes, descartados, de excepción y bytes recibidos/transmitidos - de 4 bytes, por cada uno de los 3 puertos físicos).

En la línea 56 se pueden definir variables "ocultas". Ésto significa que son variables que, si bien no fueron declaradas como tal en ECD++, deben forzosamente ser patcheadas en el microcódigo. No serán accesibles en absoluto desde ECD++ o mediante la interfaz alternativa, ya que se encuentran presentes únicamente para el correcto funcionamiento del código que corre en la parte inferior de la arquitectura. Las variables se deben especificar de la forma (sin espacios): `<nombreDeVariable>-<valorInicial>-<MEafectados>`, donde `MEafectados` en una lista de números de MicroEngines, separados por coma, en donde es necesario patchear el símbolo. En éste caso, se debió utilizar `buf_free_list0` y `buf_sram`, ambas obtenidas a partir de la asignación de memoria de la *freelist* y usadas por los ME 0 (RX) y 2 (TX); `dl_rel` que es calculada en base a parámetros de la misma asignación anterior, y `free_list_id` que también es obtenida a partir de la asignación de la freelist, pero en éste caso es sólo utilizada por el ME 0. Más abajo se explica brevemente el motivo de su existencia.

En la línea 64 se puede forzar un orden de patcheo de símbolos, caso contrario no existe un orden garantizado, se hace en forma aleatoria. Para el ME 0, podemos observar que se especifica, en primer orden, buf_free_list0, luego buf_sram y por último dl_rel. Se verá debajo, que se hace de ésta manera porque el cálculo de dl_base depende del valor de una variable que se patchea anteriormente (buf_sram). Lo mismo ocurre con el ME 2.

En la línea 70 se puede establecer el parámetro que se utilizará para calcular

`ix_rm_get_phys_offset(arg_pMemory,...)`.

Ésta función devuelve el offset físico en bytes de la memoria asignada por el RM para el puntero al bloque de memoria (arg_pMemory) que se está solicitando. Dicha función se utiliza normalmente antes de patchear un símbolo, para obtener el offset de memoria

que será aplicado al valor del símbolo. Por defecto, se calculará para la variable que se está patcheando (posiblemente proveniente de ECD++), pero se puede requerir especificar otro valor, o incluso no calcularlo. Para éste último caso, estableciendo el valor `'NO_CALC'` cumplirá dicho objetivo. En nuestro caso, no se calcula para buf_free_list0 y free_list_id (se verá a continuación que se asignan directamente de campos de una estructura).

Por último, en la línea 78 se puede establecer el valor con el que se patcheará el símbolo en los MicroEngines. Por defecto (para variables provenientes de ECD++ por ejemplo) se utlizará el valor ``‘‘ME_SRAM_ADDR(offset, memchan)’’``, el cual es una MACRO definida en ECD++IO_Core_Driver.c que permite convertir el offset y el canal de memoria en SRAM, a una dirección manejable por MicroEngines. En éste caso, se ve que para

- `buf_free_list0` se establece el valor de una estructura que especifica el índice del controlador Q-Array en SRAM para la correspondiente freelist en hardware,
- `buf_sram` (luego buf_sram_base) no se especifica, por lo que tomará el valor de la macro mencionada anteriormente, sumado al valor definido en “CUSTOM_OFFSET” definido para la misma, indicará que será un puntero al comienzo de la memoria controlada para meta-datos,
- `dl_rel` (luego dl_rel_base) es una constante que se pre-calcula, ya que al ser un valor usado con bastante frecuencia, y al contener una división, es una operación bastante costosa. Teniendo en cuenta también el valor definido en “CUSTOM_OFFSET“, la misma especifica el comienzo de la memoria controlada en donde se encuentran los datos. Equivaldría a hacer

$$DRAM\_buffer\_base = DRAM\_offset - SRAM\_freeListStart * \frac{buffer\_size}{free\_list\_element\_size]}.$$

  Teniendo éste valor, luego se puede calcular fácilmente la dirección de un búfer de paquete en DRAM con una multiplicación y suma.

- `free_list_id`, el cual se establece simplemente con un identificador (ID) que poseen las freelist por hardware.

5.2.4. Detalles del código C generado automáticamente

El código es similar al comentado en 5.1.3. Se cita lo más relevante del mismo, relacionado con el patcheo de símbolos y sus valores recién vistos, en el Código 10.

Código 10: ECD++IOCore_Driver.c para el modelo QoS_Control.

```
1  ...
2  /* List of free buffers */
3  ix_buffer_free_list_handle hwFreeList = 0;
4  /* Pointers to various run-time data structures allocated by RM */
5  /* They are taken automatically from ECD++ variables */
6  void *qos_symbol1;
7  void *qos_symbol2;
8  void *packet_counters_sram;
9  void *packet_tx_counter;
10 ...
11 /* Allocate a free buffer list */
12 err = ix_rm_hw_buffer_free_list_create(NUM_BUFFERS,
13                   sizeof(ix_hw_buffer_meta), BUF_SIZE,
14                   BUF_SRAM_CHAN, BUF_DRAM_CHAN, &hwFreeList);
15 if (err != IX_SUCCESS)
16         panic("ix_rm_hw_buffer_free_list_create failed\n");
17 /* Read freelist info (it will be needed later) */
18 err = ix_rm_buffer_free_list_get_info(hwFreeList, &hwFreeListInfo);
19 if (err != IX_SUCCESS)
20         panic("ix_rm_hw_buffer_free_list_get_info failed\n");
21 ...
22     importSymbols0[0].m_Name = "BUF_FREE_LIST0";
23     importSymbols0[0].m_Value = hwFreeListInfo.m_FreeListInfo;
24
25     importSymbols0[1].m_Name = "BUF_SRAM_BASE";
26     err = ix_rm_get_phys_offset(hwFreeListInfo.m_pMetaBaseAddress,\
27                                 NULL, &memChan, &offset, NULL);
28     if (err != IX_SUCCESS)
29             panic("ix_rm_get_phys_offset failed for %s\n",
30                     importSymbols0[1].m_Name);
```

```
31      importSymbols0[1].m_Value = ME_SRAM_ADDR(offset, memChan);
32
33      importSymbols0[2].m_Name = "DL_REL_BASE";
34      err = ix_rm_get_phys_offset(hwFreeListInfo.m_pDataBaseAddress,\
35                                  NULL, &memChan, &offset, NULL);
36      if (err != IX_SUCCESS)
37              panic("ix_rm_get_phys_offset failed for %s\n",
38                      importSymbols0[2].m_Name);
39      importSymbols0[2].m_Value = offset - \
40                                  ((importSymbols0[1].m_Value *     \
41                                  hwFreeListInfo.m_DataElementSize)\
42                                  /hwFreeListInfo.m_MetaElementSize);
43
44      importSymbols0[3].m_Name = "PACKET_COUNTERS_SRAM_BASE";
45      err = ix_rm_get_phys_offset(packet_counters_sram, NULL, \
46                                  &memChan, &offset, NULL);
47      if (err != IX_SUCCESS)
48              panic("ix_rm_get_phys_offset failed for %s\n",
49                      importSymbols0[3].m_Name);
50      importSymbols0[3].m_Value = ME_SRAM_ADDR(offset, memChan);
51
52      importSymbols0[4].m_Name = "FREE_LIST_ID";
53      importSymbols0[4].m_Value = hwFreeListInfo.m_FreeListInfo1;
54  ...
55      /* Patchs for ME 0 */
56      printk("Now trying to patch %i!!!\n", ME_ID(0));
57      err = ix_rm_ueng_patch_symbols(ME_ID(0),5,importSymbols0);
58      if (err != IX_SUCCESS)
59            panic("ix_rm_ueng_patch_symbols failed for ME Number 0 \
60                  (%i)!!!\n", ME_ID(0));
61  ...
```

6. Conclusiones

La presente Tesina introdujo nuevos aportes para la implementación de modelos DEVS en tiempo real y la experimentación en plataformas reales. Se aportaron herramientas y metodologías suficientes para obtener un procedimiento completamente basado en modelos DEVS, que puedan ser embebidos en un hardware de destino final, ejecutados en tiempo real, e interconectados con unidades externas especializadas. Se garantiza así la continuidad de los modelos DEVS, al encapsular las funcionalidades de interacción con el hardware externo en modelos atómicos especiales capaces de interactuar con interfaces de control propias de cada dispositivo.

A partir de ahora, se podrá separar la complejidad de programación en dos niveles. Es decir, a modo de ejemplo, se podrán tener dos equipos de personas, uno dedicado al diseño y programación del modelo DEVS, y otro equipo especializado dedicado a la programación de bajo nivel en el hardware, ésto sin necesidad de cruzar conocimientos entre ambos, tan sólo establecer las "variables compartidas", donde el equipo de bajo nivel podrá configurar la generación de bibliotecas y hará el proceso totalmente transparente para el equipo de desarrollo DEVS.

Todo ésto se hace posible gracias al desarrollo de las nuevas bibliotecas para comunicar ECD++ con el procesador híbrido de red IXP2400. Se obtuvo un laboratorio virtualizado y portable para realizar experimentos de modo rápido y simplificado con entornos de configuración compleja, permitiendo trabajar con una placa de red de alta capacidad basada en IXP2400. Los ejemplos prácticos estudiados demuestran que pueden llevarse a cabo con éxito las etapas finales de implementación y validación de controladores de red basados en DEVS operando en tiempo real, eliminando completamente la necesidad de adaptación de lógica ni de estructura al pasar desde las simulaciones autónomas (para propósitos de verificación) hacia las ejecuciones en modalidad Hardware-In-The-Loop (para validación e implementación final). La metodología utilizada garantiza la continuidad de los modelos y promueve el desarrollo de soluciones ingenieriles completamente basadas en modelado y simulación.

7. Trabajo futuro

Describimos aquí algunas posibles líneas de trabajo de continuación de la presente Tesina.

7.1. Control de admisión de paquetes

Se desarrolló, como se vió en 5.2.2, un microbloque denominado ECD++Microblock, el cual se encuentra corriendo en paralelo con los microbloques de recepción, procesamiento y transmisión. Lo anterior permitió que pueda recibir órdenes desde ECD++ de una manera transparente, sin embargo, carece de la posibilidad de actuar en bajo nivel, por ejemplo, descartar un paquete, ya que no se encuentra implementada una comunicación entre dicho microbloque y los demás.

Una posible línea de trabajo sería unir dicho microbloque al pipeline de procesamiento de los paquetes (es decir, comunicarlo con los demás). Ésto permitiría poder actuar directamente sobre la decisión de descarte de paquetes, conforme a una política implementada en el controlador en ECD++.

7.2. Implementación de Control de Tráfico en ECD++

Siguiendo con la idea anterior y un trabajo más avanzado, consistiría en implementar un control de tráfico completo, con manejo de múltiples colas y un scheduler que aplique alguna política final de policing (como Round Robin, Weighted Round Robin, etc); ésto sujeto al paso previo por el microbloque ECD++ el cuál, en base a parámetros internos, decidirá en cuál de todas las colas existentes del Queue Manager derivar el paquete (incluyendo una cola de descarte). La Figura 28 muestra la estructura del mismo.

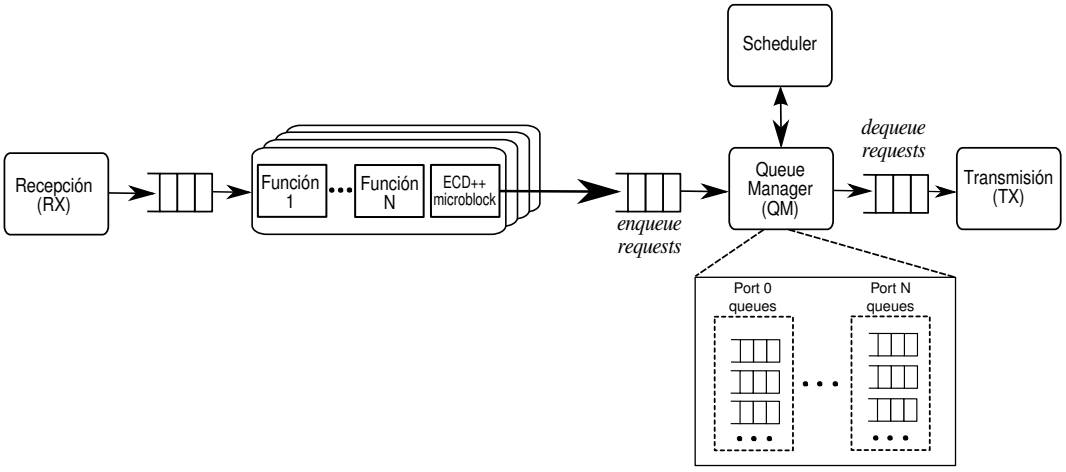

Figura 28: Una implementación de Control de Tráfico usando un administrador de colas y scheduler.

A. Apéndices

A.1. Controladores de ruteo y acceso al medio

Controladores de dispositivos SPI-3 y PM3386 que se utilizaron en la creación de la interfaz unificada para ambos. Como se ve, se "dialoga" con los dispositivos a través de llamadas a `ioctl()`, utilizando constantes definidas en la especificación del hardware particular.

Primero se observa el código del Bridge SPI-3, en donde se inicializa el dispositivo y el controlador, en la función `StartSpi3br`; y luego parte de la función `ShowSpi3br`, donde se muestran los valores en tiempo de ejecución del dispositivo (ver salida en Apéndice A.4).

Código 11: Controlador del Bridge SPI-3

```
1  ...
2  /*
3   *   StartSpi3br()
4   *   -------------
5   *   Init spi3br driver & device, open device,
6   *   install isr callback.
7   *   Device is ready for rx and tx.
8   */
9  void StartSpi3br( void ){
10   SPI3BR_STATUS           status;
11   UINT16                  intMask;
12
13   status = openSpi3br();
14   if( status != SPI3BR_SUCCESS ){
15     printf("StartSpi3br: device open failed. Device not found\n");
16     exit(-1);
17   }
18   /* enable interrupts */
19 #if( ENABLE_INTERRUPTS )
20   intMask = SPI3BR_IRQ_ALL;
21   ioctl( spi3br, SPI3BR_IOCTL_SET_INT_EN, &intMask );
22 #endif
```

```
23    closeSpi3Br();
24    printf("SPI3 bridge started\n");
25    return;
26  }
27  /*
28   *    ShowSpi3br()
29   *    ------------
30   *    Show all settings for spi3br device
31   */
32  void ShowSpi3br( void ){
33    SPI3BR_STATUS     status;
34    UINT16    p0;
35    UINT8     p1;
36    int        i;
37    SPI3BR_CHAN_CFG   chCfg;
38
39    status = openSpi3br();
40    if( status != SPI3BR_SUCCESS ){
41            printf("ShowSpi3br: device open failed, \
42                    Status = 0x%x\n", status);
43            exit(-1);
44    }
45    /* device id */
46    status = ioctl( spi3br, SPI3BR_IOCTL_GET_CHIP_ID, &p0 );
47    if( status != SPI3BR_SUCCESS ){
48            printf("SPI3BR_IOCTL_GET_CHIP_ID failed, \
49                    status = 0x%x\n", status);
50            exit(-1);
51    }
52    printf("Chip ID 0x%04x\n", p0);
53  ...
54  }
```

Luego se observa el código del controlador de los PM3386 y PM3387 (los cuales son idénticos excepto que el último posee una única interfaz, en vez de dos). Se muestran las

funciones (parciales) `StartMacs`, para inicializar el dispositivo y controlador, y la función `ShowMac` que usamos para obtener el estado en ejecución de las MACs, muy utilizada (junto con `ShowMacStats`) en la depuración.

Código 12: Controlador PM3386/7

```
1  ...
2  /*
3   *    Global storage for 2 device handles
4   */
5  static unsigned int dev0 = 0, dev1 = 0;
6  ...
7  /*
8   *    Forward declarations
9   */
10 void pm3386Callback( void *usrCtxt, PM3386_INTERRUPTS *pIntr );
11 void StartMacs();
12 void StopMacs();
13 void ShowMac(int);
14 void ShowMacStats();
15
16 static int openPM3386devices(){
17   /* open device */
18   dev0 = open("/dev/pm338x0", O_RDWR, 0);
19   if( dev0 == -1 ) {
20           dev0 = 0;
21           return PM3386_DEVICE_NOT_FOUND;
22   }
23   return PM3386_SUCCESS;
24 }
25
26 static void closePM3386devices(){
27   if (dev0) close(dev0);
28   if (dev1) close(dev1);
29   dev0 = dev1 = 0;
30 }
```

```
31
32 /*
33  *    StartMacs()
34  *    ----------
35  *    Init pm3386 driver, open macs, install isr callbacks,
36  *    and enable rx and tx
37  */
38 void StartMacs( void ){
39   PM3386_STATUS            status;
40   PM3386_CHAN_CFG          chanCfg;
41
42   status = openPM3386devices();
43   if(status != PM3386_SUCCESS ){
44     printf("StartMacs: device open failed. \
45             Device not found\n");
46     exit(-1);
47   }
48   /* enable rx and tx device 0, chan 0 */
49   chanCfg.chan = 0;
50   chanCfg.val = 1;
51   status = ioctl( dev0, PM3386_IOCTL_EGMAC_SET_RXEN, &chanCfg );
52   if( status != PM3386_SUCCESS ){
53     printf("StartMacs: ioctl failed on device 0, \
54             Status = 0x%x\n", status);
55     exit(-1);
56   }
57   status = ioctl( dev0, PM3386_IOCTL_EGMAC_SET_TXEN, &chanCfg );
58   if( status != PM3386_SUCCESS ){
59     printf("StartMacs: ioctl failed on device 1, \
60             Status = 0x%x\n", status);
61     exit(-1);
62   }
63 ...
64 /*
65  *       ShowMac()
```

```
66  *      ------------
67  *       Show all settings for a pm3386 device
68  */
69 void ShowMac( int device ){
70 ...
71   /* device id */
72   status = ioctl( dev, PM3386_IOCTL_DEV_GET_ID, &data );
73   if( status != PM3386_SUCCESS ){
74           printf("PM3386_IOCTL_DEV_GET_ID failed, \
75                   status = 0x%x\n", status);
76           exit(-1);
77   }
78   printf("Chip ID                    0x%04x\n", data);
79   /* device revision */
80   status = ioctl( dev, PM3386_IOCTL_DEV_GET_REV, &data );
81   if( status != PM3386_SUCCESS ){
82           printf("PM3386_IOCTL_DEV_GET_REV failed, \
83                   status = 0x%x\n", status);
84           exit(-1);
85   }
86   printf("Chip Rev 0x%04x\n", data);
87 ...
```

A.2. Cargador de microcódigo para EventCounter

Aquí se puede observar, casi completamente, el código generado en forma automática por la herramienta desarrollada, para el Ejemplo de la sección 5.1. Se pueden apreciar todos las librerías requeridas por el mismo, las definiciones de las *macro* utilizadas, las operaciones `ioctl()` que se aceptan desde espacio de usuario (ECD++), y todo lo referido a la interacción con el Resource Manager de Intel (reserva, asignación y copia[17] de bloques de memoria para variables, carga de microcódigo en hardware, arranque y detención de los Microengines y patcheo de símbolos provenientes de ECD++).

Código 13: ECD++IOCore_Driver.c para el modelo IXA_EventCounter

```
1  /* ECD++IOCore_Driver                                          */
2  /* ECD++ I/O Core component & driver (Linux kernel module)     */
3  /* WARNING! - This file was generated automatically.           */
4  /* Changes will be overwritten when compiling ECD++ code.      */
5  /* By iramello@fceia.unr.edu.ar                                */
6
7  /* Code runs in the Linux kernel            */
8  #include <linux/kernel.h>
9  /* The code runs as a kernel module         */
10 #include <linux/module.h>
11 /* New pseudo device is a character device  */
12 #include <linux/fs.h>
13 /* Needed for communication with user space */
14 #include <asm/uaccess.h>
15
16 /* Code uses the IXA Resource Manager       */
17 #include <ix_rm.h>
18 /* Prevents the compiler from complaining   */
19 #undef LINUX
20 /* Code uses the IXA CCI                    */
21 #include <ix_cc.h>
22 #include <ix_cci.h>
23
24 #include "ECD++IOCore_Driver.h"
25 /* Macro to convert MicroEngine sequence number */
26 /* into MicroEngine ID                          */
27 #define ME_ID(i) ((i%ME_CL_SZ)|((i/ME_CL_SZ)<<4))
28 /* Macro to convert SRAM offset and             */
29 /* memory channel into MicroEngine addressing   */
30 #define ME_SRAM_ADDR(offset,memChan) (offset|(memChan<<30))
31 /* Macro to log error message and terminate resource manager */
32 #define panic(...) { printk("%s: ",ME_LOADER_DRIVER_NAME); \
33                      printk(__VA_ARGS__); \
34                      ix_rm_term(); \
35                      return(-1); }
```

```
36 /* Macro to clear block of kernel memory */
37 #define bzero(buf,size) ix_ossl_memset(buf,0,size)
38
39 MODULE_AUTHOR("iramello@fceia.unr.edu.ar");
40 MODULE_DESCRIPTION("Automated MicroEngine Loader for IXP2XXX");
41
42 /* Variables for the kernel module      */
43 /* (use config file for modifying them) */
44 /* ME file name - UOF file name */
45 static char Uof_file[512] = "EventCounter.uof";
46 /* Local procedures                     */
47 static int patch_microblocks(ix_buffer_free_list_info);
48 static int MEL_open(struct inode *, struct file *);
49 static int MEL_release(struct inode *, struct file *);
50 static int MEL_ioctl(struct inode *, \
51                      struct file *, unsigned int, unsigned long);
52 /* List of free buffers */
53 ix_buffer_free_list_handle hwFreeList = 0;
54 /* Pointers to various run-time data structures allocated by RM */
55 /* They are taken automatically from ECD++ variables            */
56 void *ResetCommand;
57 void *EventCounter;
58 ....
59 static int MEL_ioctl(struct inode *inode, struct file *fp, \
60                      unsigned int cmd, unsigned long buf){
61 ...
62     switch (cmd) {
63         case GET_RESETCOMMAND:
64             if ((char *)buf != NULL){
65                 out = copy_to_user((char *)buf, \
66                                    ResetCommand, \
67                                    RESETCOMMAND_SIZE);
68                 ix_ossl_free(userData);
69                 return out;
70             }
```

```
71              break;
72          case SET_RESETCOMMAND:
73              if ((char *)buf != NULL){
74                  out = copy_from_user((char *)userData, \
75                                       (void *)buf, \
76                                       RESETCOMMAND_SIZE);
77                  if (strlen((char *)userData) >= RESETCOMMAND_SIZE){
78                      printk("Data from userland is bigger than \
79                              kernel memory reserved for it! \
80                              Please increase its size.\n");
81                  }
82                  else{
83                      printk("\nCopying buffer from userspace: \
84                              %s, whose size is: %d\n", \
85                              (char *)userData, \
86                              strlen((char *)userData));
87                      temp = atoi(userData);
88                      ix_ossl_memcpy(ResetCommand, &temp, 4);
89                      ix_ossl_free(userData);
90                      return out;
91                  }
92              }
93              break;
94  ...
95          default:
96              return INVALID_CMD;
97      }
98      return 0;
99  }
100
101 int init_module(void){
102     ix_error err;
103     ix_buffer_free_list_info hwFreeListInfo;
104     int i;
105
```

```
106     if (register_chrdev(ME_LOADER_MAJOR, \
107                         ME_LOADER_DRIVER_NAME, \
108                         &ME_loader_fops) < 0) {
109         printk("%s: register ME_LOADER_MAJOR failed %d\n",
110                 ME_LOADER_DRIVER_NAME, ME_LOADER_MAJOR);
111         return -1;
112     }
113     printk("%s: Trying to load %s ... \n", \
114             ME_LOADER_DRIVER_NAME, Uof_file);
115     /* Initialize Intel's Resource Manager */
116     printk("\n%s: Initializing Resource Manager\n", \
117             ME_LOADER_DRIVER_NAME);
118     err=ix_rm_init(0);
119     if (err != IX_SUCCESS) {
120         printk("Error: ix_rm_init failed\n");
121         return -1;
122     }
123
124 /* ========== START of memory allocation ==========*/
125     /* Allocate a free buffer list */
126     err = ix_rm_hw_buffer_free_list_create(NUM_BUFFERS,
127                         sizeof(ix_hw_buffer_meta), BUF_SIZE,
128                         BUF_SRAM_CHAN, BUF_DRAM_CHAN, &hwFreeList);
129     if (err != IX_SUCCESS)
130             panic("ix_rm_hw_buffer_free_list_create failed\n");
131     /* Read freelist info (it will be needed later) */
132     err = ix_rm_buffer_free_list_get_info(hwFreeList, \
133                                           &hwFreeListInfo);
134     if (err != IX_SUCCESS)
135             panic("ix_rm_hw_buffer_free_list_get_info failed\n");
136     /* Allocate buffer in SRAM used in ME       */
137     /* (for now --> SRAM, channel 0)            */
138     /* Future version of IXA_Lib will support   */
139     /* configuring memType & channel            */
140     printk("<1>Trying to allocate memory for ME now...\n");
```

```
141     err = ix_rm_mem_alloc(IX_MEMORY_TYPE_SRAM, 0, \
142                           RESETCOMMAND_SIZE, &ResetCommand);
143     if (err != IX_SUCCESS)
144             panic("ix_rm_mem_alloc failed for: ResetCommand\n");
145     /* Clear the buffers */
146     bzero(ResetCommand, RESETCOMMAND_SIZE);
147     err = ix_rm_mem_alloc(IX_MEMORY_TYPE_SRAM, 0, \
148                           EVENTCOUNTER_SIZE, &EventCounter);
149     if (err != IX_SUCCESS)
150             panic("ix_rm_mem_alloc failed for: EventCounter\n");
151     /* Clear the buffers */
152     bzero(EventCounter, EVENTCOUNTER_SIZE);
153
154 /* ========== END of memory allocation ==========*/
155
156     /* Reset the MicroEngines */
157     printk("%s: Resetting all MicroEngines\n", \
158             ME_LOADER_DRIVER_NAME);
159     ix_rm_ueng_reset_all();
160     /* Get the microcode from the UOF file */
161     printk("%s: Setting ucode\n", ME_LOADER_DRIVER_NAME);
162     err = ix_rm_ueng_set_ucode(Uof_file);
163     if (err != IX_SUCCESS)
164         panic("ix_rm_ueng_set_ucode failed\n");
165     /* Patch the microcode symbols before actually */
166     /* loading microcode.                          */
167     if (patch_microblocks(hwFreeListInfo) < 0)
168         return(-1);
169     /* Load microcode into MicroEngines */
170     printk("%s: Loading uCode\n", ME_LOADER_DRIVER_NAME);
171     err = ix_rm_ueng_load();
172     if (err != IX_SUCCESS)
173         panic("ix_rm_ueng_load failed\n");
174     /* Start the assigned MicroEngines */
175     for (i=0; i<ME_NUM; i++) {
```

```
176         if ((ME_MASK>>i) & 0x1) {
177             printk("%s: Starting ME%i\n", \
178                     ME_LOADER_DRIVER_NAME, i);
179             err = ix_rm_ueng_start(ME_ID(i), CONTEXT_MASK);
180             if (err != IX_SUCCESS)
181                 panic("ix_rm_ueng_start failed for ME %i\n", i);
182         }
183     }
184     return 0;
185 }
186
187 void cleanup_module(void){
188     ix_error err;
189     int i;
190
191     /* Stop each of the assigned MicroEngines */
192     for (i=0; i<ME_NUM; i++) {
193         if ((ME_MASK>>i) & 0x1) {
194             printk("%s: Stopping ME%i\n", \
195                     ME_LOADER_DRIVER_NAME, i);
196             err = ix_rm_ueng_stop(ME_ID(i));
197             if (err != IX_SUCCESS)
198                 printk("%s:
199                         ix_rm_ueng_stop failed for ME %i\n",
200                         ME_LOADER_DRIVER_NAME, i);
201         }
202     }
203     /* Terminate the Resource Manager */
204     ix_rm_term();
205     /* Unregister pseudo-device */
206     unregister_chrdev(ME_LOADER_MAJOR, ME_LOADER_DRIVER_NAME);
207     printk("%s: Hasta la vista, baby...\n",ME_LOADER_DRIVER_NAME);
208 }
209
210 /*****************************************************************/
```

```
211 /* Patch the microcode symbols before actually loading microcode */
212 /*****************************************************************/
213 int patch_microblocks(ix_buffer_free_list_info hwFreeListInfo){
214     ix_error err;
215     ix_uint32       memChan;
216     ix_uint32       offset;
217     ix_imported_symbol importSymbols0[2];
218
219     importSymbols0[0].m_Name = "EVENTCOUNTER";
220     err = ix_rm_get_phys_offset(EventCounter, NULL, &memChan, \
221                                 &offset, NULL);
222     if (err != IX_SUCCESS)
223         panic("ix_rm_get_phys_offset failed for %s\n",
224                 importSymbols0[0].m_Name);
225     importSymbols0[0].m_Value = ME_SRAM_ADDR(offset, memChan);
226
227     importSymbols0[1].m_Name = "RESETCOMMAND";
228     err = ix_rm_get_phys_offset(ResetCommand, NULL, &memChan, \
229                                 &offset, NULL);
230     if (err != IX_SUCCESS)
231         panic("ix_rm_get_phys_offset failed for %s\n",
232                 importSymbols0[1].m_Name);
233     importSymbols0[1].m_Value = ME_SRAM_ADDR(offset, memChan);
234
235     /* Patchs for ME 0 */
236     printk("Now trying to patch ME %i!!!\n", ME_ID(0));
237     err = ix_rm_ueng_patch_symbols(ME_ID(0),2,importSymbols0);
238     if (err != IX_SUCCESS)
239       panic("ix_rm_ueng_patch_symbols failed for ME \
240             Number 0 (%i)!!!\n", ME_ID(0));
241
242     return(1);
243 }
```

A.3. Armado de laboratorio virtual

Aquí se detalla cómo se armó el laboratorio virtualizado, comentado anteriormente en la sección 4.1 y visible en la Figura 11). Además, el siguiente procedimiento puede ser útil en caso de requerir armar un nuevo laboratorio, comenzando desde cero (como en el caso de la presente Tesina). El fabricante emitió originalmente un documento[9], el cual sirvió parcialmente para construir el procedimiento, pero se encuentra desactualizado, por lo cual mucho de lo explicado no aplica en la actualidad. Otro documento que ha sido de utilidad es [18], que, si bien fue confeccionado para la versión de IXP2400 con doble procesador (IXPD2400), igual sirven muchas explicaciones.

- Se conectó ENP-2611 en un slot PCI libre en "Host PCI" (no pudiendo hacerlo en "Máquina Física" por imposibilidad de no entrar en el gabinete),
- Se conectó un cable serial desde ENP-2611 hacia Host PCI,
- Lo demás se conectó como se muestra en dicha figura, siendo todos cables UTP, excepto los dos que comunican los Media-Converter y la placa, que son fibras ópticas de las características mencionadas en 4.1,

Apenas arranque el Host PCI, ENP-2611 comenzará su rutina de arranque y chequeo, la cual demora aproximadamente 15 segundos, luego del cual queda esperando instrucciones.

Para acceder en éste momento, se necesita de la conexión serial. Se configuró en el Host PCI para que la conexión serie esté disponible a través de telnet hacia un puerto especial (elegimos 2000), lo que permitirá acceder a la misma independientemente del sistema operativo (HyperTerminal en Windows, Minicom en Linux, etc). Para ello, se utilizó un convertidor Serie/Ethernet llamado "*ser2net*", el cual se configuró con las siguientes líneas:

```
BANNER:banner:\r\nAcceso a puerto serie virtual ENP-2611 desde puerto \p \
  dispositivo local \d [\s] (Ubuntu Linux)\r\n\r\n
2000:telnet:600:/dev/ttyS0:57600 8DATABITS NONE 1STOPBIT banner
```

Código 14: Configuración de ser2net

Con ésto, se puede entonces acceder a ENP-2611 desde cualquier parte de la red, en particular vamos a hacerlo desde el Linux-Host. Apenas lo hacemos, se apreta el único

botón que viene con ENP-2611 y sirve para reiniciar la placa. La salida para lo anterior se puede ver en el Código 15.

```
ivan@ubuntu-devsecd:~$ telnet enp2611holder 2000
Trying 192.168.15.99...
Connected to enp2611holder.
Escape character is '^]'.
Acceso a puerto serie virtual ENP-2611 desde puerto 2000
dispositivo local /dev/ttyS0 [57600 N81] (Ubuntu Linux)

Rebooting...
Please stand by while rebooting the system.
Restarting system.

 Testing Xscale Scratchpad memory... PASS
 Testing I2C Interface... PASS
 Testing SDRAM memory... PASS
Scrubbing SDRAM
Scrubbing from 00040000 to 01000000
Found IXP2400 Rev. B0
Core clock frequency is 600 MHZ
Loading FPGA Data: FPGA ready to load bytes
FPGA ready to load bytes
FPGA programmed OK
 +64bit Extension:
  Primary: Disabled
  Secondary: Enabled
Configuring Upstream/Downstream PCI windows..
Mapping US MEM1.. 1 MB Window
Mapping DS1 I/O (MEM1).. 1 MB CSR Window
Mapping DS2 to SDRAM (Window Size 64MB) (Offset into SDRAM 0x0c000000)
Mapping DS3 to SDRAM (Window Size 1MB)
Clearing Primary lockout.. PASS
Loading MAC addresses into PM3386.. PASS
Scrubbing remaining SDRAM (1000000 to 10000000)
Scrubbing..Complete (ECC Enabled)
Using MAC address from NIC EEPROM
Ethernet eth0: MAC address 00:00:50:11:1d:9b
IP: 192.168.25.2, Default server: 192.168.25.3

RedBoot(tm) for ENP-2611, version R1.24 - Prototype - release 2.0  -
built 13:28:15, Aug 10 2004
Platform: ENP-26xx (XScale) RadiSys Corp. IXP2400 packet processor
Copyright (C) 2000, 2001, Red Hat, Inc.
SDRAM(256M): 0x00000000 - 0x10000000, 0x00200000 - 0x0ffdf000 available
```

```
SRAM Channel 0(4M): 0x80000000 - 0x80400000
SRAM Channel 1(4M): 0x90000000 - 0x90400000
FLASH(16M): 0xc4000000 - 0xc5000000, 128 blocks of 0x00020000 bytes each.
CPLD Rev. is 9
FPGA Rev. is 0x19
```

Código 15: Configuración de ser2net

Se puede apreciar en la línea 16 que ya ha obtenido IP desde un servidor DHCP. El mismo se ha configurado en el Linux-Host. En el Código 36 se puede ver lo más relevante del archivo de configuración.

```
...
subnet 192.168.25.0 netmask 255.255.255.0 {
  default-lease-time 1209600;
  max-lease-time 31557600;
  allow bootp;
  allow booting;
  group {
    # Servidor TFTP (no usado)
    next-server 192.168.25.3;
    # Gateway
    option routers 192.168.25.1;
    host enp2611 {
      # Nombre para la placa
      option host-name "enp2611";
      # MAC de la placa (donde <xx:...:xx> es la MAC
      # de la conexion interna de ENP2611
      hardware ethernet <xx:xx:xx:xx:xx:xx>;
      # IP asiganda
      fixed-address 192.168.25.2;
      # Punto de montaje NFS
      option root-path "<MontavistaInstallDir>";
      # Kernel para booteo automatico en placa (no usado)
      filename "zImage.enp2611";
    }
  }
}
...
```

Código 16: Configuración del servidor DHCP.

Se instaló en *<MontavistaInstallDir>* dentro del Linux-Host, un *Root Filesystem* de Linux Montavista. Éste directorio será el que ENP-2611 buscará cuando inicie.

En éste punto, se está en un punto en que la placa inicia, pero queda en su *boot-manager*, el cual es RedBoot[14]. La misma posee un kernel Linux Montavista ya cargado, como lo muestra la última línea del Código 17, se procedió a crear un script en RedBoot para automatizar la carga del mismo, como lo muestra el Código 18.

```
RedBoot> fis list
Name              FLASH addr  Mem addr    Length      Entry point
RedBoot           0xC4000000  0x00000000  0x00040000  0x00000000
System Log        0xC4FA0000  0xC4FA0000  0x00020000  0x00000000
RedBoot config    0xC4FDF000  0xC4FDF000  0x00001000  0x00000000
FIS directory     0xC4FE0000  0xC4FE0000  0x00020000  0x00000000
fpga              0xC4040000  0x00400000  0x00100000  0x00000000
diag              0xC4140000  0x00400000  0x00100000  0x00400000
linux             0xC4240000  0x0C028000  0x00100000  0x0C028000
```

Código 17: Lista de código cargado en ENP-2611.

```
RedBoot> fconfig
Run script at boot: true
Boot script:
.. fis load linux
.. go
Enter script, terminate with empty line
>>
Boot script timeout (1000ms resolution): 2
Use BOOTP for network configuration: false true
Clear PCI Primary lockout in the RedBoot Bootmanager: true
GDB connection port: 9000
Network debug at boot time: false
Update RedBoot non-volatile configuration - are you sure (y/n)? y
... Unlock from 0xc4fc0000-0xc4fe0000: .
... Erase from 0xc4fc0000-0xc4fe0000: .
... Program from 0x00054350-0x00055350 at 0xc4fdf000: .
... Lock from 0xc4fc0000-0xc4fe0000: .
```

Código 18: Script de automatización de carga del kernel en RedBoot.

Con los pasos anteriores, se tiene hasta aquí un ENP-2611 corriendo un Linux MontaVista embebido, completamente funcional.

A.4. Scripts y salidas estándar

Aquí se observan los scripts utilizados en la inicialización de las diferentes partes de la placa, y su correspondiente salida estándar.

Se comienza mostrando la salida estándar para el comando ‘‘`Initialize`’’ y los argumentos que puede recibir, útilizados para depuración, y que fueron comentados en el Código 2.

```
# ./Initialize showmacstats
                          Dev0,Chan0      Dev0,Chan1      Dev1,Chan0
                          ___________     ___________     ___________

FramesReceivedOK          0x0000000009    0x0000000000    0x0000000000
OctetsReceivedOK          0x00000002ee    0x0000000000    0x0000000000
FramesReceived            0x0000000009    0x0000000000    0x0000000000
OctetsReceived            0x0000000390    0x0000000000    0x0000000000
UnicastFramesReceivedOK   0x0000000000    0x0000000000    0x0000000000
MulticastFramesReceivedOK 0x0000000004    0x0000000000    0x0000000000
BroadcastFramesReceivedOK 0x0000000005    0x0000000000    0x0000000000
TaggedFramesReceivedOK    0x0000000000    0x0000000000    0x0000000000
PauseMacCtrlFrameReceived 0x0000000000    0x0000000000    0x0000000000
MacCtrlFrameReceived      0x0000000000    0x0000000000    0x0000000000
FrameCheckSequenceErrors  0x0000000000    0x0000000000    0x0000000000
FramesLostInternalMacErr  0x0000000000    0x0000000000    0x0000000000
SymbolError               0x0000000000    0x0000000000    0x0000000000
InRangeLengthErrors       0x0000000000    0x0000000000    0x0000000000
FramesTooLongErrors       0x0000000000    0x0000000000    0x0000000000
Jabbers                   0x0000000000    0x0000000000    0x0000000000
Fragments                 0x0000000000    0x0000000000    0x0000000000
UndersizedFrames          0x0000000000    0x0000000000    0x0000000000
ReceiveFrames64           0x0000000001    0x0000000000    0x0000000000
ReceiveFrames65to127      0x0000000008    0x0000000000    0x0000000000
ReceivedFrames128to255    0x0000000000    0x0000000000    0x0000000000
ReceivedFrames256to511    0x0000000000    0x0000000000    0x0000000000
ReceivedFrames512to1023   0x0000000000    0x0000000000    0x0000000000
ReceivedFrames1024to1518  0x0000000000    0x0000000000    0x0000000000
ReceivedFrames1519toMax   0x0000000000    0x0000000000    0x0000000000
JumboOctetsReceivedOK     0x0000000000    0x0000000000    0x0000000000
FilteredOctets            0x0000000000    0x0000000000    0x0000000000
FilteredUnicastFrames     0x0000000000    0x0000000000    0x0000000000
FilteredMulticastFrames   0x0000000000    0x0000000000    0x0000000000
FilteredBroadcastFrames   0x0000000000    0x0000000000    0x0000000000

FramesTransmittedOK       0x0000000000    0x0000000009    0x0000000000
```

```
OctetsTransmittedOK          0x0000000000  0x00000002ee  0x0000000000
OctetsTransmitted            0x0000000000  0x00000002ee  0x0000000000
FramesLostInternalMacTxErr   0x0000000000  0x0000000000  0x0000000000
TransmitSystemError          0x0000000000  0x0000000000  0x0000000000
UnicastFramesTxAttempted     0x0000000000  0x0000000000  0x0000000000
UnicastFramesTxOK            0x0000000000  0x0000000000  0x0000000000
MulticastFramesTxAttempted   0x0000000000  0x0000000004  0x0000000000
MulticastFramesTxOK          0x0000000000  0x0000000004  0x0000000000
BroadcastFramesTxAttempted   0x0000000000  0x0000000005  0x0000000000
BroadcastFramesTxOK          0x0000000000  0x0000000005  0x0000000000
PauseMacCtrlFramesTx         0x0000000000  0x0000000000  0x0000000000
MacCtrlFramesTx              0x0000000000  0x0000000000  0x0000000000
TxFrames64                   0x0000000000  0x0000000001  0x0000000000
TxFrames65to127              0x0000000000  0x0000000008  0x0000000000
TxFrames128to255             0x0000000000  0x0000000000  0x0000000000
TxFrames256to511             0x0000000000  0x0000000000  0x0000000000
TxFrames512to1023            0x0000000000  0x0000000000  0x0000000000
TxFrames1024to1518           0x0000000000  0x0000000000  0x0000000000
TxFrames1519toMax            0x0000000000  0x0000000000  0x0000000000
JumboOctesTxOK               0x002aac9320  0x300c000000  0x1000000000
```

Código 19: Salida estándar de "Initialize showmacstats".

```
# ./Initialize showmacs

------- MAC 0 -------
Chip ID                  0x3386
Chip Rev                 0x0000
Interrupt Status         0x0000
Device Status            0xc011
ROOL Status              0x0000
DOOL Status              0x0002
Equalized tx mode        0x0000
Pause mode               0x0000
Equalization threshold   0x0006
Equalization difference  0x0006
Channel Hi Water         0x0006          0x0006
Channel Lo Water         0x0005          0x0005
Channel pkt burst        0x0003          0x0003
Channel fifo reserve     0x0007          0x0007
Channel min frame        0x0005          0x0005
Channel L32B             0x0000          0x0000
Channel L10B             0x0000          0x0000
Channel paden            0x0001          0x0001
Channel crcen            0x0000          0x0000
```

```
Channel fctx                0x0000            0x0000
Channel fcrx                0x0000            0x0000
Channel longp               0x0000            0x0000
Channel rxen                0x0001            0x0001
Channel txen                0x0001            0x0001
Channel ipg                 0x000c            0x000c
Channel stn addr            0x000050115992    0x000050115993
Channel max rx frame        0x05ee            0x05ee
channel AN enable           0x0001            0x0001
channel AN status           0x000d            0x0000
channel AN Adv Lo           0x01a0            0x01a0
channel AN Adv Hi           0x0000            0x0000
channel AN Link Lo          0x41a0            0x0000
channel AN Link Hi          0x0000            0x0000
channel host pause          0x0000            0x0000
channel pass err            0x0001            0x0001
channel pass ctrl           0x0001            0x0001
channel pause time          0xffff            0xffff
channel pause ival          0x7f67            0x7f67
channel max tx frame        0x05ee            0x05ee
channel rx fwd thsld        0x0181            0x0181
channel addr match 0        0x000000000000    0x000000000000
channel addr match 1        0x000000000000    0x000000000000
channel addr match 2        0x000000000000    0x000000000000
channel addr match 3        0x000000000000    0x000000000000
channel addr match 4        0x000000000000    0x000000000000
channel addr match 5        0x000000000000    0x000000000000
channel addr match 6        0x000000000000    0x000000000000
channel addr match 7        0x000000000000    0x000000000000
channel vid match 0         0x0000            0x0000
channel vid match 1         0x0000            0x0000
channel vid match 2         0x0000            0x0000
channel vid match 3         0x0000            0x0000
channel vid match 4         0x0000            0x0000
channel vid match 5         0x0000            0x0000
channel vid match 6         0x0000            0x0000
channel vid match 7         0x0000            0x0000
Channel mhash               0x0000000000000000  0x0000000000000000
channel fltr ctrl 0         0x0000            0x0000
channel fltr ctrl 1         0x0000            0x0000
channel fltr ctrl 2         0x0000            0x0000
channel fltr ctrl 3         0x0000            0x0000
channel fltr ctrl 4         0x0000            0x0000
channel fltr ctrl 5         0x0000            0x0000
```

```
channel fltr ctrl 6        0x0000          0x0000
channel fltr ctrl 7        0x0000          0x0000
channel mhash en           0x0000          0x0000
channel promisc mode       0x0001          0x0001
channel serdes loopback    0x0000          0x0000
inten.interface            0x1fff
inten.mstat0               0xffff
inten.mstat1               0xffff
------ MAC 1 ------
...
```

Código 20: Salida estándar de "Initialize showmacs".

```
# ./Initialize spi3bridge
Chip ID                    0x1331
Chip Rev                   0x  19
Interrupt Enable           0x07ff
Interrupt Status           0x0000
Port 0 enable              0x0003
Port 1 enable              0x0003
Port 2 enable              0x0003
Port 3 enable              0x0003
Burst Size                 0x  01
Port 0 pause               0x0000
Port 1 pause               0x0000
Port 2 pause               0x0000
Port 3 pause               0x0000
Port 0 paused              0x0000
Port 1 paused              0x0000
Port 2 paused              0x0000
Port 3 paused              0x0000
Port 0 rx fifo ctrl        0x0001
Port 1 rx fifo ctrl        0x0001
Port 2 rx fifo ctrl        0x0001
Port 3 rx fifo ctrl        0x0001
Port 0 tx fifo ctrl        0x0003
Port 1 tx fifo ctrl        0x0003
Port 2 tx fifo ctrl        0x0003
Port 3 tx fifo ctrl        0x0003
```

Código 21: Salida estándar de "Initialize spi3bridge".

En el Código 22 se puede observar el contenido de ``InitializeForRealTraffic.sh'', el cual se utiliza para cargar los controladores e inicializar simultáneamente, los controla-

dores de acceso al medio (PM3386/7) y el bridge SPI-3, lo que permitirá interactuar con tráfico real.

Código 22: Script para inicialización de PHYs y MACs de ENP-2611

```
#!/bin/sh
echo
echo "Loading PHY Drivers..."
echo
module="pm338x"
device="pm338x"
group="root"
mode="664"
major="101"

/sbin/insmod $module.o pm3386_major=$major || exit 1
rm -f /dev/${device}[0-1]
mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
chgrp $group /dev/${device}[0-1]
chmod $mode /dev/${device}[0-1]
echo
echo "Loading Bridge Drivers..."
echo
module="spi3br"
device="spi3br"
group="root"
mode="664"
major="100"

/sbin/insmod $module.o spi3br_major=$major || exit 1
rm -f /dev/${device}
mknod /dev/${device} c $major 0
chgrp $group /dev/${device}
chmod $mode /dev/${device}
echo
echo "Initializing MACs and PHYs..."
echo
./Initialize start
```

```
35 echo "Done."
```

Se continúa en el Código 23, mostrando el contenido de ‘‘ECD++IXA_IOCore_Load.sh’’, el cual se utilizó para cargar los módulos de toda la arquitectura IXA, junto con el módulo que carga el microcódigo, analizado en 4.4.1.

Código 23: Script para la carga de módulos de IXA junto con el controlador de las bibliotecas desarrolladas.

```
1  #!/bin/sh
2
3  rm -f /dev/rm_drv
4  rm -f /dev/ECD++IOCore_Device
5  insmod -f libossl.o
6  insmod -f halMeDrv.o
7  insmod -f halMev2_lib.o
8  insmod -f rm_drv.o
9  echo
10 echo "About to load ECD++ Microblock..."
11 insmod -f ECD++IOCore_Driver.o
12 mknod /dev/ECD++IOCore_Device c 236 0
13 echo "Done."
```

En lo que sigue, se muestra una salida correcta, relacionada al ejemplo de aplicación descrito en 5.2, de la aplicación de todos los comandos anteriores.

El registro del kernel Linux tendrá una salida como se muestra en el Código 24.

```
...
Found PM3386 chip at address 0xc6000000
Found PM3386 chip at address 0xc6400000
PM3386 devices initialized
MSF Initialization done
Found SPI3 bridge chip at address 0xc5000000
SPI3 Bridge initialized
ECD++IOCore_Device: Trying to load ECD++Microblocks.uof ...
ECD++IOCore_Device: Initializing Resource Manager
Trying to allocate memory for ME now...
ECD++IOCore_Device: Resetting all MicroEngines
ECD++IOCore_Device: Setting ucode
Now trying to patch ME 0x00!!!
Now trying to patch ME 0x02!!!
```

```
Now trying to patch ME 0x03!!!
ECD++IOCore_Device: Loading uCode
Patching for ME: 0x00 symbol: QARRAY_INIT_SRAM_BASE_CH1 to value: 0x00000000
Patching for ME: 0x00 symbol: QARRAY_INIT_SRAM_BASE_CH0 to value: 0x00000000
Patching for ME: 0x00 symbol: FREE_LIST_ID to value: 0x00000001
Patching for ME: 0x00 symbol: PACKET_COUNTERS_SRAM_BASE to value: 0x00048508
Patching for ME: 0x00 symbol: DL_REL_BASE to value: 0x003ef800
Patching for ME: 0x00 symbol: BUF_SRAM_BASE to value: 0x00008500
Patching for ME: 0x00 symbol: BUF_FREE_LIST0 to value: 0x00000011
Patching for ME: 0x02 symbol: PACKET_TX_COUNTER_BASE to value: 0x00048538
Patching for ME: 0x02 symbol: DL_REL_BASE to value: 0x003ef800
Patching for ME: 0x02 symbol: BUF_SRAM_BASE to value: 0x00008500
Patching for ME: 0x02 symbol: BUF_FREE_LIST0 to value: 0x00000011
Patching for ME: 0x03 symbol: QOS_SYMBOL2_BASE to value: 0x00048504
Patching for ME: 0x03 symbol: QOS_SYMBOL1_BASE to value: 0x00048500
ECD++IOCore_Device: Starting ME0
ECD++IOCore_Device: Starting ME1
ECD++IOCore_Device: Starting ME2
ECD++IOCore_Device: Starting ME3
...
```

Código 24: Logging en kernel de la aplicación de todos los comandos vistos.

Por último, se muestra la salida estándar cuando se corre ECD++ embebido, indicándole al mismo que lo va a hacer en *tiempo real* (con el parámetro "-W"). Notar cómo se corresponde ésta salida con lo visto en la Figura 25.

```
# ./simu -mQoS_Control.ma -W -t00:00:50:00 -lQoS_Control.log -oQoS_Control.out

N-CD++: A Tool to Implement n-Dimensional Cell-DEVS models
--------------------------------------------------------

Version 2.0-R.45 December-1999
Daniel Rodriguez, Gabriel Wainer, Amir Barylko, Jorge Beyoglonian
Departamento de Computacion. Facultad de Ciencias Exactas y Naturales.
Universidad de Buenos Aires. Argentina.
...
Loading models from QoS_Control.ma
Loading events from
Message log: QoS_Control.log
Output to: QoS_Control.out
Tolerance set to: 1e-08
Configuration to show real numbers: Width = 12 - Precision = 5
Quantum: Not used
Evaluate Debug Mode = OFF
```

```
DEVS Model Debug Mode = OFF
Flat Debug Mode = OFF
Debug Cell Rules Mode = OFF
Temporary File created by Preprocessor = /tmp/filexMXJGX
Printing parser information = OFF

Starting simulation. Stop at time: 00:00:50:000
Arrancando QosControl
...
 Traffic Meter SW/HW Manager: Trying to access CoreComponent...
        Current counter for packet_counters_sram:          3
 Traffic Meter SW/HW Manager: CoreComponent Accessed !
0x1f158c00:00:04:050 Traffic Meter Output value Metered signal: 3
0x1f158c Traffic Sensor EXTERNAL:: msg: 00:00:04:050 lastChange: 00:00:04:000
0x1f158c00:00:04:050 Traffic Meter Notify Internal at 00:00:04:050
0x1f158c00:00:04:050 Traffic Sensor Output value sensed traffic rate: 3
0x1f158c00:00:04:050 Traffic Sensor Notif Internal at 00:00:04:050
0x1f158c QoSControl EXTERNAL:: msg: 00:00:04:050 lastChange: 00:00:03:050
0x1f158c00:00:05:000 Traffic Sensor Output value sense signal: 1
0x1f158c00:00:05:000 Traffic Sensor Sleep Internal at 00:00:05:000
0x1f158c00:00:05:000 Traffic Sensor Goes waiting for 00:00:02:500
0x1f158c Traffic Meter EXTERNAL:: msg: 00:00:05:000 lastChange: 00:00:04:050
 Traffic Meter SW/HW Manager: Trying to access CoreComponent...
        Current counter for packet_counters_sram:          3
 Traffic Meter SW/HW Manager: CoreComponent Accessed !
0x1f158c00:00:05:050 Traffic Meter Output value Metered signal: 3
0x1f158c Traffic Sensor EXTERNAL:: msg: 00:00:05:050 lastChange: 00:00:05:000
...
0x1f158c00:00:13:050 Traffic Shaper Output value Shape Action: 13
 Traffic Shaper SW/HW Manager: Trying to access CoreComponent...
 Traffic Shaper SW/HW Manager: Value written...
 Traffic Shaper SW/HW Manager: CoreComponent Accessed !
0x1f158c00:00:13:050 Traffic Shaper Notify Internal at 00:00:13:050
0x1f158c00:00:14:000 Traffic Sensor Output value sense signal: 1
0x1f158c00:00:14:000 Traffic Sensor Sleep Internal at 00:00:14:000
0x1f158c00:00:14:000 Traffic Sensor Goes waiting for 00:00:02:500
0x1f158c Traffic Meter EXTERNAL:: msg: 00:00:14:000 lastChange: 00:00:13:050
 Traffic Meter SW/HW Manager: Trying to access CoreComponent...
        Current counter for packet_counters_sram:          7
 Traffic Meter SW/HW Manager: CoreComponent Accessed !
0x1f158c00:00:14:050 Traffic Meter Output value Metered signal: 7
0x1f158c Traffic Sensor EXTERNAL:: msg: 00:00:14:050 lastChange: 00:00:14:000
0x1f158c00:00:14:050 Traffic Meter Notify Internal at 00:00:14:050
0x1f158c00:00:14:050 Traffic Sensor Output value sensed traffic rate: 7
```

```
0x1f158c00:00:14:050 Traffic Sensor Notif Internal at 00:00:14:050
0x1f158c QoSControl EXTERNAL:: msg: 00:00:14:050 lastChange: 00:00:13:050
0x1f158c00:00:15:000 Traffic Sensor Output value sense signal: 1
0x1f158c00:00:15:000 Traffic Sensor Sleep Internal at 00:00:15:000
0x1f158c00:00:15:000 Traffic Sensor Goes waiting for 00:00:02:500
...
0x1f158c QoSControl EXTERNAL:: msg: 00:00:18:050 lastChange: 00:00:17:050
0x1f158c00:00:18:050 QoSController: QoSCommand = 20
0x1f158c Traffic Actuator EXTERNAL:: msg: 00:00:18:050 lastChange: 00:00:13:050
0x1f158c00:00:18:050 Traffic Actuator Output value Actuation Command: 20
0x1f158c Traffic Shaper EXTERNAL:: msg: 00:00:18:050 lastChange: 00:00:13:050
0x1f158c00:00:18:050 Traffic Shaper Output value Shape Action: 20
 Traffic Shaper SW/HW Manager: Trying to access CoreComponent...
 Traffic Shaper SW/HW Manager: Value written...
 Traffic Shaper SW/HW Manager: CoreComponent Accessed !
0x1f158c00:00:18:050 Traffic Shaper Notify Internal at 00:00:18:050
0x1f158c00:00:19:000 Traffic Sensor Output value sense signal: 1
0x1f158c00:00:19:000 Traffic Sensor Sleep Internal at 00:00:19:000
0x1f158c00:00:19:000 Traffic Sensor Goes waiting for 00:00:02:500
0x1f158c Traffic Meter EXTERNAL:: msg: 00:00:19:000 lastChange: 00:00:18:050
 Traffic Meter SW/HW Manager: Trying to access CoreComponent...
        Current counter for packet_counters_sram:           1
 Traffic Meter SW/HW Manager: CoreComponent Accessed !
0x1f158c00:00:19:050 Traffic Meter Output value Metered signal: 1
0x1f158c Traffic Sensor EXTERNAL:: msg: 00:00:19:050 lastChange: 00:00:19:000
0x1f158c00:00:19:050 Traffic Meter Notify Internal at 00:00:19:050
0x1f158c00:00:19:050 Traffic Sensor Output value sensed traffic rate: 1
...
0x1f158c00:00:24:000 Traffic Sensor Sleep Internal at 00:00:24:000
0x1f158c00:00:24:000 Traffic Sensor Goes waiting for 00:00:02:500
0x1f158c Traffic Meter EXTERNAL:: msg: 00:00:24:000 lastChange: 00:00:23:050
 Traffic Meter SW/HW Manager: Trying to access CoreComponent...
        Current counter for packet_counters_sram:          10
 Traffic Meter SW/HW Manager: CoreComponent Accessed !
0x1f158c00:00:24:050 Traffic Meter Output value Metered signal: 10
0x1f158c Traffic Sensor EXTERNAL:: msg: 00:00:24:050 lastChange: 00:00:24:000
0x1f158c00:00:24:050 Traffic Meter Notify Internal at 00:00:24:050
0x1f158c00:00:24:050 Traffic Sensor Output value sensed traffic rate: 10
0x1f158c00:00:24:050 Traffic Sensor Notif Internal at 00:00:24:050
0x1f158c QoSControl EXTERNAL:: msg: 00:00:24:050 lastChange: 00:00:23:050
...
0x1f158c00:00:33:000 Traffic Sensor Sleep Internal at 00:00:33:000
0x1f158c00:00:33:000 Traffic Sensor Goes waiting for 00:00:02:500
0x1f158c Traffic Meter EXTERNAL:: msg: 00:00:33:000 lastChange: 00:00:32:050
```

```
 Traffic Meter SW/HW Manager: Trying to access CoreComponent...
        Current counter for packet_counters_sram:          4
 Traffic Meter SW/HW Manager: CoreComponent Accessed !
0x1f158c00:00:33:050 Traffic Meter Output value Metered signal: 4
0x1f158c Traffic Sensor EXTERNAL:: msg: 00:00:33:050 lastChange: 00:00:33:000
0x1f158c00:00:33:050 Traffic Meter Notify Internal at 00:00:33:050
0x1f158c00:00:33:050 Traffic Sensor Output value sensed traffic rate: 4
0x1f158c00:00:33:050 Traffic Sensor Notif Internal at 00:00:33:050
0x1f158c QoSControl EXTERNAL:: msg: 00:00:33:050 lastChange: 00:00:32:050
0x1f158c00:00:33:050 QoSController: QoSCommand = 20
0x1f158c Traffic Actuator EXTERNAL:: msg: 00:00:33:050 lastChange: 00:00:28:050
0x1f158c00:00:33:050 Traffic Actuator Output value Actuation Command: 20
0x1f158c Traffic Shaper EXTERNAL:: msg: 00:00:33:050 lastChange: 00:00:28:050
0x1f158c00:00:33:050 Traffic Shaper Output value Shape Action: 20
 Traffic Shaper SW/HW Manager: Trying to access CoreComponent...
 Traffic Shaper SW/HW Manager: Value written...
 Traffic Shaper SW/HW Manager: CoreComponent Accessed !
0x1f158c00:00:33:050 Traffic Shaper Notify Internal at 00:00:33:050
...
Simulation ended!
```

Código 25: Salida de ECD++ para el ejemplo en 5.2

A.5. Generadores de código

En ésta sección se muestran los generadores de códigos desarrollados más relevantes. Además de éstos, existen algunos otros: el generador de contenido de la biblioteca para ECD++ (gen_ECD++IXA_Library), el generador del encabezado de la biblioteca para ECD++ (gen_ECD++IXA_Library_header), el parser del modelo ECD++ (visto en 3) y el secuenciador de los generadores en general (code_generator).

En primer lugar se puede ver en el Código 26, el fuente del generador del encabezado (".h") del controlador ECD++IOCore. En el mismo se observan mayormente, definiciones de constantes (posiblemente definidas en el archivo de configuración de la biblioteca).

Código 26: Generador del encabezado del controlador ECD++IOCore.

```
1 #!/usr/bin/perl
2 ####### gen_ECD++IOCore_Driver_header.pl ###############
3 # Part of general script to automate the
4 # creation of IXA variables, accesible from ECD++.
5 # This script takes care of ".h" of ECD++ I/O Core Driver.
6 ############################
7 # v1.0: iramello - 20100525
```

```
8  ##########################################################
9
10 $CONFIG="./IXA-Generator_config.pm";
11 require $CONFIG;
12
13 die "IXA Control Code output folder is NOT defined. Bye bye.\n"
14   unless defined($IXA_CTL_OUTPUT_DIR);
15 die "ECD++ I/O Core Driver header name is NOT defined. Bye bye.\n"
16   unless defined($ME_LOADER_HEADER_CODE);
17
18 # Number of variables we are going to communicate between IXP and ECD++
19 $NumArg = $#ARGV + 1;
20
21 if ($NumArg < 1) {
22   die "ERROR: At least 1 (one) IXA variable MUST be declared...\nQuitting.\n";
23 }
24
25 open (FILE_OUTPUT, ">$ME_LOADER_HEADER_CODE");
26 print FILE_OUTPUT "/* ",substr($ME_LOADER_HEADER_CODE_NAME, 0,
27                                length($ME_LOADER_HEADER_CODE_NAME)-2)," */
28 /* Constants shared by microcode and core code */
29 /* WARNING! - This file was generated automatically. Changes will */
30 /* be overwritten when compiling ECD++ code. */
31 /* By iramello\@fceia.unr.edu.ar */
32
33 /* Name of ME_loader */
34 #define ME_LOADER_DRIVER_NAME \"ECD++IOCore_Device\"
35
36 /* Packet buffer parameters: 8MB of buffers, 512 bytes per buffer     */
37 #define NUM_BUFFERS 8*1024
38 #define BUF_SIZE 512
39 /* Packet buffer parameters: 64MB of buffers, 2048 bytes per buffer   */
40 //#define NUM_BUFFERS 32*1024
41 //#define BUF_SIZE 2048
42
43 /* Memory channels for free buffer list */
44 #define BUF_SRAM_CHAN 0
45 #define BUF_DRAM_CHAN 0
46
47 /* Number of microengines */
48 #define ME_NUM $ME_NUMBER
49
50 /* Size of one microengine cluster */
51 #define ME_CL_SZ 4
52
53 /* Which MEs do we start? */
54 #define ME_MASK $ME_MASK
55
56 /* Context mask */
57 #define CONTEXT_MASK $CONTEXT_MASK
58
59 /* ========== Commands for ioctl ========================================== */
60 /* Possible ioctl commands for the ME_loader pseudo-device */
61 typedef enum MEL_cmd_t {
62 ";
63 $count = 0;
64 while ($count < $NumArg) {
65   $item = $ARGV[$count];
66   if ( (defined($SIZES{$ARGV[$count]}) && $SIZES{$ARGV[$count]} > 0)
67         || (!grep(/^$item/, @HIDDEN_VARS)) ){
```

```
68      print FILE_OUTPUT "\tGET_",uc $ARGV[$count], ",";
69      print FILE_OUTPUT "SET_",uc $ARGV[$count], ",\n";
70    }
71    $count++;
72  }
73
74  print FILE_OUTPUT "} mel_cmd;
75
76  #define INVALID_CMD -1
77
78  /* File name for ME_loader pseudo-device */
79  #define ME_LOADER_DEV_FILE \"/dev/ECD++IOCore_Device\"
80
81  /* ME_Loader device Major Number */
82  #define ME_LOADER_MAJOR 236
83
84  /* Size of the SRAM buffers used in MEs */
85  ";
86  # Ejemplo de salida siguiente:
87  ##define SRAM_CNTR_SIZE  32
88  $count = 0;
89  while ($count < $NumArg) {
90    my $size;
91    if (defined $SIZES{$ARGV[$count]}) {
92      $size = $SIZES{$ARGV[$count]};
93    }
94    else{
95      # If a variable is defined in ECD++ as int and does not have
96      # defined a particular size, this is the default.
97      print FILE_OUTPUT "//Default for a variable defined on ECD++.\n";
98      $size = "4";
99    }
100   print FILE_OUTPUT "#define ",uc $ARGV[$count],"_SIZE  $size
101 ";
102   $count++;
103 }
104 close(FILE_OUTPUT);
```

En el Código 27 se puede observar el generador de contenido del controlador ECD++IOCore.

Código 27: Generador del contenido del controlador ECD++IOCore.

```
1  #!/usr/bin/perl
2  ####### gen_ECD++IOCore_Driver.pl ##################
3  # Part of general script to automate the
4  # creation of IXA variables, accesible from ECD++.
5  # This script takes care of ".c" of I/O Core Driver.
6  ##############################
7  # v1.0: iramello - 20100525
8  ###################################################
9
10 # ARGV comes as a serie of: variableName,initValue,ME_Number
11 # Example: "variableLoca,383,3 variable17,1234567890,7"
12
13 $CONFIG="./IXA-Generator_config.pm";
14 require $CONFIG;
15
16 die "IXA Control Code output folder is NOT defined. Bye bye.\n"
17   unless defined($IXA_CTL_OUTPUT_DIR);
```

```
18 die "ECD++ I/O Core Driver name is NOT defined. Bye bye.\n"
19   unless defined($ME_LOADER_CODE);
20
21 # Number of variables we are going to communicate between IXP and ECD++
22 # #ARGV is (number of args - 1)
23 $NumArg = $#ARGV + 1;
24 if ($NumArg < 1) {
25   die "ERROR: At least 1 (one) IXA variable MUST be declared...\nQuitting.\n";
26 }
27
28 %data;
29 my @ME_Vars;
30 # First we start with "normal" variables, i.e. variables declared on ECD++.
31 foreach $tuple (@ARGV){
32   my ($name,$initValue,$me_numbers) = split("-", $tuple);
33   my @MEs = split(",", $me_numbers);
34   my $hasSize = 0;
35   # Each item of hash has a form of:
36   # 'variableName' => [ initValue, 'ME_Numbers', isHidden, hasSize ]
37   $hasSize = 1 if (exists($SIZES{$name}) && ($SIZES{$name} > 0) );
38   $data{$name} = [ $initValue, $me_numbers, 0, $hasSize ];
39   foreach $me (@MEs){
40           push @{ $ME_Vars[$me] }, $name;
41   }
42 }
43 if ($DEBUG){
44   print "\nNORMAL VARS 0: ","@{@ME_Vars[0]}","\n";
45   print "NORMAL VARS 2: ","@{@ME_Vars[2]}","\n";
46 }
47
48 # We continue with "hidden" variables, i.e. variables NOT declared
49 # on ECD++ but on this generator config.
50 foreach $tuple (@HIDDEN_VARS){
51   my ($name,$initValue,$me_numbers) = split("-", $tuple);
52   my @MEs = split(",", $me_numbers);
53   my $hasSize = 0;
54   $hasSize = 1 if (exists($SIZES{$name}) && ($SIZES{$name} > 0) );
55   $data{$name} = [ $initValue, $me_numbers, 1, $hasSize ];
56   foreach $me (@MEs){
57           push @{ $ME_Vars[$me] }, $name;
58   }
59 }
60 if ($DEBUG){
61   print "WITH EXTRA VARS 0: ","@{@ME_Vars[0]}","\n";
62   print "WITH EXTRA VARS 2: ","@{@ME_Vars[2]}","\n";
63 }
64
65 # Now we order list of variables per ME ($ME_Vars[$me]) so we
66 # can patch symbols as requested.
67 for($me = 0; $me < $ME_NUMBER; $me++){
68   if (defined(@VAR_ORDER_IN_ME[$me])) {
69     $numOrderedVars = scalar(@{$VAR_ORDER_IN_ME[$me]});
70     $numTotalVars = scalar(@{$ME_Vars[$me]});
71     while(scalar(@{$VAR_ORDER_IN_ME[$me]})){
72       $lastElem = pop(@{$VAR_ORDER_IN_ME[$me]});
73       for($i = 0; $i < $numTotalVars; $i++){
74           if($lastElem eq $ME_Vars[$me][$i]){
75                   splice(@{$ME_Vars[$me]},$i,1);
76                   unshift(@{$ME_Vars[$me]},$lastElem);
77                   last;
```

```
78              }
79          }
80          if ($i == $numTotalVars){
81              print "WARNING! Variable ",$lastElem,"
82                    defined in ORDER_ARRAY not found for MicroEngine $me!\n";
83          }
84        }
85    }
86  }
87
88  if ($DEBUG){
89   foreach $varName (keys %data){
90      print "Variable ",$varName," initial value: ",$data{$varName}[0],"\n";
91      print "Variable ",$varName," MEs: ",$data{$varName}[1],"\n";
92   }
93   $i = 0;
94   while ($i < $ME_NUMBER){
95      if (defined($ME_Vars[$i])){
96        print "ME_Vars[$i] = ";
97        foreach $var (@{ $ME_Vars[$i] }){
98              print "$var ";
99        }
100       print "\n";
101     }
102     $i++;
103  }
104 }
105
106 # This subroutine returns 0 or 1 depending
107 # on the variable, if is hidden or not.
108 sub HasSize{
109   ($data{$_[0]}[3] > 0);
110 }
111
112 # This subroutine returns 0 or 1 depending
113 # on the variable, if is has a defined size or not.
114 sub IsHidden{
115   ($data{$_[0]}[2] > 0);
116 }
117
118 # This subroutine checks if variable has defined a
119 # particular suffix for it variable_size_name.
120 sub GetSuffix{
121   if (defined($SUFFIX{$_[0]})){
122       $SUFFIX{$_[0]};
123   }
124   else{
125       '_BASE';
126   }
127 }
128
129 # We search for a particular symbol: "NO_CALC", which means no
130 # ix_rm_get_phys_offset will be fetched.
131 sub CustomOffsetNotNeeded{
132   return (defined($CUSTOM_OFFSET{$_[0]}) && $CUSTOM_OFFSET{$_[0]} eq 'NO_CALC');
133 }
134
135 # We need to get an offset different from the default (which normal symbols have).
136 sub HasCustomOffset{
137   return (defined($CUSTOM_OFFSET{$_[0]}) && $CUSTOM_OFFSET{$_[0]} ne 'NO_CALC');
```

```
138 }
139
140 sub Has4ByteSize{
141   return (not(defined($SIZES{$_[0]})) || $SIZES{$_[0]} == 4);
142 }
143
144 open (FILE_OUTPUT, ">$ME_LOADER_CODE");
145 print FILE_OUTPUT "/* ",substr($ME_LOADER_CODE_NAME, 0, length($ME_LOADER_CODE_NAME)-2)," */
146 /* ECD++ I/O Core component & driver (Linux kernel module) */
147 /* WARNING! - This file was generated automatically. */
148 /* Changes will be overwritten when compiling ECD++ code. */
149 /* By iramello\@fceia.unr.edu.ar */
150
151 #include <linux/kernel.h>   /* Code runs in the Linux kernel            */
152 #include <linux/module.h>   /* The code runs as a kernel module         */
153 #include <linux/fs.h>       /* New pseudo device is a character device  */
154 #include <asm/uaccess.h>    /* Needed for communication with user space */
155
156 #include <ix_rm.h>          /* Code uses the IXA Resource Manager       */
157 #undef LINUX                /* Prevents the compiler from complaining   */
158 #include <ix_cc.h>          /* Code uses the IXA CCI                    */
159 #include <ix_cci.h>
160
161 #include \"$ME_LOADER_HEADER_CODE_NAME\"
162
163 /* Macro to convert microengine sequence number */
164 /* into microengine ID                          */
165 #define ME_ID(i) ((i%ME_CL_SZ)|((i/ME_CL_SZ)<<4))
166
167 /* Macro to convert SRAM offset and memory channel into microengine addressing*/
168 #define ME_SRAM_ADDR(offset,memChan) (offset|(memChan<<30))
169
170 /* Macro to log error message and terminate resource manager */
171 #define panic(...) { printk(\"%s: \",ME_LOADER_DRIVER_NAME);\\
172                      printk(__VA_ARGS__);\\
173                      ix_rm_term();\\
174                      return(-1); }
175
176 /* Macro to clear block of kernel memory */
177 #define bzero(buf,size) ix_ossl_memset(buf,0,size)
178
179 MODULE_AUTHOR(\"iramello\@fceia.unr.edu.ar\");
180 MODULE_DESCRIPTION(\"Automated Microengine Loader for IXP2XXX\");
181
182 /*Variables for the kernel module (you can modify this to suit your case)*/
183 /* ME file name - UOF file name */
184 static char Uof_file[512] = \"$UOF_FILE\";
185
186 /* Local procedures    */
187 static int patch_microblocks(ix_buffer_free_list_info);
188 static int MEL_open(struct inode *, struct file *);
189 static int MEL_release(struct inode *, struct file *);
190 static int MEL_ioctl(struct inode *, struct file *, unsigned int, unsigned long);
191
192 /* List of free buffers */
193 ix_buffer_free_list_handle hwFreeList = 0;
194
195 /* Pointers to various run-time data structures allocated by RM */
196 /* They are taken automatically from ECD++ variables */
197 ";
```

```
198 foreach $varName (keys %data) {
199   if ( &HasSize($varName) || (!grep(/^$varName/, @HIDDEN_VARS)) ){
200       print FILE_OUTPUT "void *",$varName, ";\n";
201   }
202 }
203
204 print FILE_OUTPUT "
205 /* Operations for the ME_loader pseudo-device */
206 static struct file_operations ME_loader_fops = {
207         ioctl:          MEL_ioctl,
208         open:           MEL_open,
209         release:        MEL_release
210 };
211
212 static int MEL_open(struct inode *inode, struct file *filp){
213         MOD_INC_USE_COUNT;
214         return 0;
215 }
216 static int MEL_release(struct inode *inode, struct file *filp){
217         MOD_DEC_USE_COUNT;
218         return 0;
219 }
220
221 static int MEL_ioctl(struct inode *inode, struct file *fp, \
222                      unsigned int cmd, unsigned long buf){
223
224         unsigned long *userData = ix_ossl_malloc(BUF_SIZE);
225         int out = 0;
226         int temp = 0;
227
228         switch (cmd) {
229 ";
230 foreach $varName (keys %data) {
231   if (&HasSize($varName) || (!grep(/^$varName/, @HIDDEN_VARS)) ){
232     print FILE_OUTPUT "
233       case GET_",uc $varName,":
234         if ((char *)buf != NULL){
235             out = copy_to_user((char *)buf, ",$varName,", ",uc $varName,"_SIZE);
236             ix_ossl_free(userData);
237             return out;
238         }
239         break;
240       case SET_",uc $varName,":
241         if ((char *)buf != NULL){
242             out = copy_from_user((char *)userData, (void *)buf, ",uc $varName,"_SIZE);
243             if (strlen((char *)userData) > ",uc $varName,"_SIZE){
244                 printk(\"Data from userland is bigger than kernel memory reserved for it! \
245                         Please increase its size.\\n\");
246             }
247             else{";
248                 if (&Has4ByteSize($varName)){
249                         print FILE_OUTPUT "
250                 printk(\"\\nCopying buffer from userspace: %s, whose size is: %d\\n\", \
251                         (char *)userData, strlen((char *)userData));
252                 temp = atoi(userData);
253                 ix_ossl_memcpy($varName, &temp, 4);
254                 ix_ossl_free(userData);
255                 return out;";
256                 }
257                 else{
```

```
258                         print FILE_OUTPUT "
259                 printk(\"\\nZeroing memory space for variable: $varName\\n\");
260                 bzero($varName, ",uc $varName,"_SIZE);
261                 ix_ossl_free(userData);
262                 return out;";
263                 }
264             print FILE_OUTPUT "
265             }
266         }
267         break;";
268     }
269 }
270 print FILE_OUTPUT "
271       default:
272               return INVALID_CMD;
273       }
274       return 0;
275 }
276
277 ";
278
279 print FILE_OUTPUT "
280 int init_module(void){
281   ix_error err;
282   ix_buffer_free_list_info hwFreeListInfo;
283   int i;
284
285   if (register_chrdev(ME_LOADER_MAJOR, ME_LOADER_DRIVER_NAME, &ME_loader_fops) < 0) {
286           printk(\"%s: register ME_LOADER_MAJOR failed %d\\n\",
287                   ME_LOADER_DRIVER_NAME, ME_LOADER_MAJOR);
288           return -1;
289   }
290
291   printk(\"% s: Trying to load %s ... \\n\", ME_LOADER_DRIVER_NAME, Uof_file);
292
293   /* Initialize Intel's Resource Manager */
294   printk(\"\\n%s: Initializing Resource Manager\\n\", ME_LOADER_DRIVER_NAME);
295   err=ix_rm_init(0);
296   if (err != IX_SUCCESS) {
297           printk(\"Error: ix_rm_init failed\\n\");
298           return -1;
299   }
300
301 /* ========================= START of memory allocation ========================================*/
302   /* Allocate a free buffer list */
303   err = ix_rm_hw_buffer_free_list_create(NUM_BUFFERS,
304                     sizeof(ix_hw_buffer_meta), BUF_SIZE,
305                     BUF_SRAM_CHAN, BUF_DRAM_CHAN, &hwFreeList);
306   if (err != IX_SUCCESS)
307           panic(\"ix_rm_hw_buffer_free_list_create failed\\n\");
308
309   /* Read freelist info (it will be needed later) */
310   err = ix_rm_buffer_free_list_get_info(hwFreeList, &hwFreeListInfo);
311   if (err != IX_SUCCESS)
312           panic(\"ix_rm_hw_buffer_free_list_get_info failed\\n\");
313
314   /* Allocate buffer in SRAM used in ME (for now --> SRAM, channel 0) */
315   /* Future version of IXA_Lib will support configuring memType & channel */
316   printk(\"<1>Trying to allocate memory for ME now...\\n\");
317 ";
```

```
318
319 foreach $varName (keys %data) {
320   if (&HasSize($varName) || (!grep(/^$varName/, @HIDDEN_VARS)) ){
321     print FILE_OUTPUT "
322       err = ix_rm_mem_alloc(IX_MEMORY_TYPE_SRAM, 0, ",uc $varName,"_SIZE, &",$varName,");
323       if (err != IX_SUCCESS)
324           panic(\"ix_rm_mem_alloc failed for: ",$varName,"\\n\");
325       /* Clear the buffers */
326       bzero(",$varName,", ",uc $varName,"_SIZE);
327 ";
328   }
329 }
330
331 print FILE_OUTPUT "
332 /* ========================= END of memory allocation ========================================*/
333
334   /* Reset the microengines */
335   printk(\"%s: Resetting all microengines\\n\", ME_LOADER_DRIVER_NAME);
336   ix_rm_ueng_reset_all();
337
338   /* Get the microcode from the UOF file */
339   printk(\"%s: Setting ucode\\n\", ME_LOADER_DRIVER_NAME);
340   err = ix_rm_ueng_set_ucode(Uof_file);
341   if (err != IX_SUCCESS)
342           panic(\"ix_rm_ueng_set_ucode failed\\n\");
343
344   /* Patch the microcode symbols before actually */
345   /* loading microcode.                          */
346   if (patch_microblocks(hwFreeListInfo) < 0)
347           return(-1);
348
349   /* Load microcode into microengines */
350   printk(\"%s: Loading uCode\\n\", ME_LOADER_DRIVER_NAME);
351   err = ix_rm_ueng_load();
352   if (err != IX_SUCCESS)
353           panic(\"ix_rm_ueng_load failed\\n\");
354
355   /* Start the assigned microengines */
356   for (i=0; i<ME_NUM; i++) {
357       if ((ME_MASK>>i) & 0x1) {
358           printk(\"%s: Starting ME%i\\n\", ME_LOADER_DRIVER_NAME, i);
359           err = ix_rm_ueng_start(ME_ID(i), CONTEXT_MASK);
360           if (err != IX_SUCCESS)
361               panic(\"ix_rm_ueng_start failed for ME %i\\n\", i);
362       }
363   }
364   return 0;
365 }
366
367 void cleanup_module(void){
368   ix_error err;
369   int i;
370
371   /* Stop each of the assigned microengines */
372   for (i=0; i<ME_NUM; i++) {
373     if ((ME_MASK>>i) & 0x1) {
374         printk(\"%s: Stopping ME%i\\n\", ME_LOADER_DRIVER_NAME, i);
375         err = ix_rm_ueng_stop(ME_ID(i));
376         if (err != IX_SUCCESS)
377             printk(
```

```
                    \"%s: ix_rm_ueng_stop failed for ME %i\\n\", ME_LOADER_DRIVER_NAME, i);
    }
  }

  /* Terminate the Resource Manager */
  ix_rm_term();

  /* unregister pseudo-device */
  unregister_chrdev(ME_LOADER_MAJOR, ME_LOADER_DRIVER_NAME);
  printk(\"%s: Hasta la vista, baby...\\n\",ME_LOADER_DRIVER_NAME);

}

/*****************************************************************/
/* Patch the microcode symbols before actually loading microcode. */
/*****************************************************************/
int patch_microblocks(ix_buffer_free_list_info  hwFreeListInfo)
{
        ix_error err;
        ix_uint32       memChan;
        ix_uint32       offset;
";
$me_num = 0;
while ($me_num < $ME_NUMBER) {
        if (defined $ME_Vars[$me_num]){
         $len = @{ $ME_Vars[$me_num] };
         if ($len){
                print FILE_OUTPUT "\tix_imported_symbol importSymbols",\
                                $me_num,"[",$len,"];\n";
         }
        }
        $me_num++;
}

$me_num = 0;
while ($me_num < $ME_NUMBER) {
        if (defined $ME_Vars[$me_num]){
         $len = @{ $ME_Vars[$me_num] };
         if ($len){
                for($i = 0; $i < $len; $i++){
                        my $suffix;
                        $suffix = &GetSuffix($ME_Vars[$me_num][$i]);
                        print FILE_OUTPUT "
        importSymbols",$me_num,"[",$i,"].m_Name = \"",uc $ME_Vars[$me_num][$i],
                        uc $suffix,"\";";

                        # Checks if get_phys_offset is needed...
                        if (&CustomOffsetNotNeeded($ME_Vars[$me_num][$i])){
                                # Nothing to do
                        }
                        elsif (&HasCustomOffset($ME_Vars[$me_num][$i])){
                                print FILE_OUTPUT "
        err = ix_rm_get_phys_offset(",$CUSTOM_OFFSET{$ME_Vars[$me_num][$i]},", NULL, \
                                        &memChan, &offset, NULL);
        if (err != IX_SUCCESS)
                panic(\"ix_rm_get_phys_offset failed for %s\\n\",
                        importSymbols",$me_num,"[",$i,"].m_Name);";
                        }
                        else{ #Default case
                                print FILE_OUTPUT "
```

```
438         err = ix_rm_get_phys_offset(",$ME_Vars[$me_num][$i],", NULL, &memChan, &offset, NULL);
439         if (err != IX_SUCCESS)
440                 panic(\"ix_rm_get_phys_offset failed for %s\\n\",
441                         importSymbols",$me_num,"[",$i,"].m_Name);";
442                         }
443
444                         # Now we set Symbols Values...
445                         # If we said we are not going to calculate phys_offset,
446                         # we need to provide some value
447                         if (&CustomOffsetNotNeeded($ME_Vars[$me_num][$i])){
448                                 if (defined($VALUES{$ME_Vars[$me_num][$i]})){
449                                         print FILE_OUTPUT "
450         importSymbols",$me_num,"[",$i,"].m_Value = ",$VALUES{$ME_Vars[$me_num][$i]},";\n";
451                                 }else{
452                                         print "ERROR! Variable ",$ME_Vars[$me_num][$i],"
453                                           has been defined WRONGLY to not calculate physical
454                                           offset and no value has been provided!\n";
455                                 }
456                         }
457                         # A particular value has been defined for the symbol
458                         elsif (&HasCustomOffset($ME_Vars[$me_num][$i])) {
459                                 if (defined($VALUES{$ME_Vars[$me_num][$i]})){
460                                         print FILE_OUTPUT "
461         importSymbols",$me_num,"[",$i,"].m_Value = ",$VALUES{$ME_Vars[$me_num][$i]},";\n";
462                                 }
463                                 else{
464                                         print FILE_OUTPUT "
465         importSymbols",$me_num,"[",$i,"].m_Value = ME_SRAM_ADDR(offset, memChan);\n";
466                                 }
467                         }
468                         # Default case, we calculate based on physical offset obtained before
469                         else{
470                                 print FILE_OUTPUT "
471         importSymbols",$me_num,"[",$i,"].m_Value = ME_SRAM_ADDR(offset, memChan);\n";
472                         }
473                 } #endfor
474          } #endif
475         } #endif
476          $me_num++;
477 } #endwhile
478
479 $me_num = 0;
480 while ($me_num < $ME_NUMBER) {
481         if (defined $ME_Vars[$me_num]){
482          $len = @{ $ME_Vars[$me_num] };
483                 print FILE_OUTPUT "
484         /* Patchs for ME ",$me_num," */
485         printk(\"Now trying to patch ME %i!!!\\n\", ME_ID($me_num));
486         err = ix_rm_ueng_patch_symbols(ME_ID($me_num), $len, importSymbols",$me_num,");
487         if (err != IX_SUCCESS)
488              panic(\"ix_rm_ueng_patch_symbols failed for ME Number ",$me_num," (%i)!!!\\n\",\
489                     ME_ID($me_num));
490         ";
491         }
492         $me_num++;
493 }
494
495 print FILE_OUTPUT "
496
497         return(1);
```

```
498 }
499 ";
500 close(FILE_OUTPUT);
```

Por último, en el Código 28, se ve el generador de código de la interfaz de usuario alternativa, que permite ver y establecer "manualmente", valores de variables IXA desde la línea de comandos.

Código 28: Generador del contenido de la interfaz de usuario ECD++IOCore.

```
1  #!/usr/bin/perl
2  ####### gen_ECD++IOCore_Interface.pl ############################
3  # Part of general script to automate the
4  # creation of IXA variables, accesible from ECD++.
5  # This script takes care of ".c" of
6  # user interface and control functions.
7  ##############################
8  # v1.0: iramello - 20100529
9  ################################################
10
11 $CONFIG="./IXA-Generator_config.pm";
12 require $CONFIG;
13
14 die "IXA Control Code output folder is NOT defined. Bye bye.\n"
15   unless defined($ME_CONTROL_CODE_NAME);
16 die "IXA Control Code name is NOT defined. Bye bye.\n"
17   unless defined($ME_CONTROL_CODE);
18
19 # Number of variables we are going to communicate between IXP and ECD++
20 $NumArg = $#ARGV + 1;
21 if ($NumArg < 1) {
22         die "ERROR: At least 1 (one) IXA variable MUST be declared...\nQuitting.\n";
23 }
24
25 open (FILE_OUTPUT, ">$ME_CONTROL_CODE");
26 print FILE_OUTPUT "/* ",substr($ME_CONTROL_CODE_NAME, 0,
27                                length($ME_CONTROL_CODE_NAME)-2) ," */
28 /* User interface and control functions for ECD++ I/O */
29 /* Core Interface. */
30 /* WARNING! - This code was generated automatically and */
31 /* will be overwritten next time you compile ECD++. */
32
33 #include <errno.h>
34 #include <fcntl.h>
35 #include <unistd.h>
36 #include <sys/ioctl.h>
37 #include <sys/mman.h>
38 #include \"$ME_LOADER_HEADER_CODE_NAME\"
39
40 #define USAGE \\
41 \"Usage: %s [ show | set <IXA_Var> <value> ]\\n\", argv[0]
42
43 void AppShow( void ){
44         char buf[1024];
45         u_int32_t  *p1;
46
47         int MEldrfd;
```

```
48          MEldrfd = open(ME_LOADER_DEV_FILE, O_RDWR, 0);
49          if ( MEldrfd == -1 ) {
50                  printf(\"Failed to open %s\\n\", ME_LOADER_DEV_FILE);
51                  return;
52          }
53
54          p1=(u_int32_t *)buf; /* SRAM buffer address  */";
55  $count = 0;
56  while ($count < $NumArg) {
57    $varName = $ARGV[$count];
58    if ( (defined($SIZES{$ARGV[$count]}) && $SIZES{$ARGV[$count]} > 0)
59          || (!grep(/^$varName/, @HIDDEN_VARS)) ){
60          print FILE_OUTPUT "
61          ioctl(MEldrfd, GET_",uc $ARGV[$count],", p1);
62          printf(\"\\nCurrent counter for ",$ARGV[$count],":  %10u\", *p1);
63  ";
64    }
65    $count++;
66  }
67
68  print FILE_OUTPUT "
69          printf(\"\\n\");
70          return;
71  }
72
73  void SendCmd( mel_cmd command, void *value ){
74          int MEldrfd;
75          MEldrfd = open(ME_LOADER_DEV_FILE, O_RDWR, 0);
76          if ( MEldrfd == -1 ) {
77                  printf(\"Failed to open %s\\n\",ME_LOADER_DEV_FILE);
78                  return;
79          }
80          ioctl(MEldrfd, command, value);
81          close(MEldrfd);
82          return;
83  }
84
85  int main(int argc, char **argv){
86          if (argc != 2) {
87                  if (argc != 4){
88                          printf(USAGE);
89                          return 0;
90                  }
91          }
92
93          if (strncmp(argv[1], \"show\", 4) == 0) {
94                  AppShow();
95          } else if (strncmp(argv[1], \"set\", 3) == 0) {
96  \t\t";
97  $count = 0;
98  while ($count < $NumArg) {
99    $varName = $ARGV[$count];
100   if ( (defined($SIZES{$ARGV[$count]}) && $SIZES{$ARGV[$count]} > 0)
101         || (!grep(/^$varName/, @HIDDEN_VARS)) ){
102           print FILE_OUTPUT "if (strncmp(argv[2], \"",$ARGV[$count],"\", ",\
103                                   length($ARGV[$count])," ) == 0){
104                           SendCmd(SET_",uc $ARGV[$count],", argv[3]);
105                   }else ";
106         }
107         $count++;
```

```
108 }
109
110 print FILE_OUTPUT "printf(\"Invalid IXA variable\\n\");
111         } else {
112                 printf(\"Invalid parameter\\n\");
113                 printf(USAGE);
114         }
115         return 0;
116 }
117 ";
118 close(FILE_OUTPUT);
```

B. Producción científica

A continuación se transcribe el trabajo científico que será enviado a *Symposium On Theory of Modeling & Simulation 2011 (TMS/DEVS'11)*, el cual ha de llevarse a cabo en *Boston, MA, USA*, los días 4 a 9 de Abril de 2011.

Model-based Design and Implementation of Embedded Real-Time Network Controllers

Rodrigo Castro[1], Matías Bonaventura[1], Iván Ramello[2], Gabriel A. Wainer[3]

[1] Computer Science Department
Universidad de Buenos Aires
Ciudad Universitaria. Pabellón I.
(1428) Buenos Aires, Argentina.
Email: abonaven@dc.uba.ar,
rodrigocastro@ieee.org

[2] Computer Science Department
Universidad Nacional de Rosario
Av. Pellegrini 250
(2000) Rosario, Argentina
Email: iramello@fceia.unr.edu.ar

[3] Dept. of Systems and Computer Engineering
Carleton University
Centre of Visualization and Simulation (V-Sim)
1125 Colonel By Dr. Ottawa, ON, Canada.
Email: gwainer@sce.carleton.ca

Abstract

Existing methods for developing embedded real-time systems can be difficult to scale. When based on modeling and simulation techniques, models specified on the initial stages are not suitable for their implementation in the real target system, calling for recoding efforts. The DEVS modeling and simulation approach we present here allows models designs at the specification, simulation and verification stages to be directly reused and embedded on real-time target systems in a Hardware-In-The-Loop fashion. We show the development of new interface libraries that integrate the ECD++ simulator with an Intel IXP2400 network processor, allowing their real-time interaction and enabling the continuity of DEVS models through the whole development process until the final runtime product. Additionally, we present an advanced graphical tools and a virtualized portable laboratory that simplify development and experimentation efforts. Two practical examples show how DEVS models can be successfully used for developing real-time embedded controllers in the context of networking applications.

1. INTRODUCTION

For decades, the software engineering community has pursued the goal of creating formal methods for developing embedded systems, particularly for those with real time constraints. Despite numerous efforts, most existing methods are still difficult to scale, and require costly test efforts without guarantying error-free products. Instead, systems engineering has generally used modeling and simulation techniques to improve performance and obtain high quality products. Modeling and simulation development is widely used for early stages of a project, but when it evolves toward the final hardware implementation, the initial models are usually abandoned due to drastic changes in the environment, programming paradigm, data manipulation and time advance control.

This paper proposes an approach based on DEVS modeling and simulation to mitigate the previously mentioned problem in the final stages of software development for real-time embedded systems. The strategy combines the advantages of a practical approach with the strictness of a formal method, allowing them to keep the models (previously used for the specification, simulation, and verification phases) for use in the validation and implementation final stages.

The objective is to obtain a procedure completely based on DEVS models, which are a) embedded in the final target hardware, b) run –simulated- in real-time c) transparently connected to external hardware drives (Hardware-In-the -Loop, HIL) capable of action and reaction. The strategy maintains the continuity of DEVS models, by encapsulating the interaction with external hardware in special atomic models (Mappers) that are capable of interacting with control interfaces (drivers) of each device.

For the purposes a) and b) the DEVS-based ECD++ tool is used, given its specific design for real-time embedded systems. However, this task will require a code translation, as in this thesis PowerDEVS tool was proposed for the early stages of modeling, simulation, and verification. Although formally any DEVS model is interchangeable between tools based on this formalism, the code that implements the behavior of these models can differ dramatically between tools.

For objective c), ECD++ will be integrated with an Intel IXP2400 network processor and interface libraries for HIL operations between the simulator and low-level devices of the processor will be developed. Also, advanced visual tools will be provided to simplify and automate the development of ECD++ models and interface libraries.

Additionally, multiple development tools for experimentation with ECD++ and RadiSys ENP2611 network processor (based on the IXP2400) will be integrated and preconfigured, creating a virtualized portable laboratory that simplifies the validation and implementation of models.

2. BACKGROUND

Various techniques have been proposed to achieve continuity and consistency of models when implemented in embedded systems. For example BIP [1] defines components as a superposition of three layers: Behavior (a set of transitions), interactions (between transitions) and Priorities (to choose between transitions). BIP preserves properties during model composition and supports analysis and transformations between heterogeneous boundaries (timed / not timed, synchronous / asynchronous, event-triggered / data-triggered). Metropolis is an environment for electronic system design that supports specification, simulation, formal analysis and synthesis. It is based on metamodels with formal semantics, capturing designs at a high level of abstraction and implementing Models of Computation (MoC). Ptolemy II [2] allows hierarchical and structured modeling of heterogeneous systems using a specific MOC that provides data flow and control flow. ECSL (Embedded Systems Control Language) supports the development of distributed controllers [3], including a specific domain environment for automotive systems (extending Matlab family with capabilities for specification, verification, planning, performance analysis, etc.) SystemC and Esterel are system level languages used

to simulate and run models, and have been beneficiated from a growing industry adoption [4]. SystemC represents hardware and software systems at different levels of abstraction, allowing to choose the desired level for each component. Esterel is used to synthesize hardware and software through a language based on reaction and high-level statements to handle concurrency. Other major efforts include MoBIES (based on hybrid multi-agent system communication), OCAPI-XL (focus on networked devices with processors and hardware accelerators in a single chip), ForSyDe (based on modeling of heterogeneous MoCs systems, allowing Haskell-based simulation), and Foresight (using modeling and simulation based on resource constraints, graphical modeling, and generation of executable models.)

One of the most popular techniques is UML-RT, which provides an object-oriented methodology. [5] provides a comparison between UML-RT and DEVS, which shows that despite Profile UML-RT specify time, planning, and performance using UML objects, they are not formally defined.

DEVS theory provides a sound syntax and semantics for representing structure, behavior, and time, leading to a well-defined and unambiguous MoC. However, DEVS is not intended for software design and development. Therefore, it is essential to provide support for the evolution of simulation models to equivalent software components that can operate on the complexity of embedded environments.

3. ECD++ INTEGRATION WITH THE NETWORK PROCESSOR

The exponential growth of Internet resulted in the emergence of intelligent network nodes based on programmable devices. The convergence of voice and data respecting the quality of service requires the ability to process packets by making complex control decisions and holding high speeds. Furthermore, with the rapid evolution of protocols, standards and applications, the networks control logic must adapt to a much greater rate than that provided for the replacement of the underlying hardware technology. In turn, the speed of networks are growing at a rate exceeding the growth rate of the CPU clock by about an order of magnitude, comparing package transmission time with memory access. This presents a complex technological challenge since it requires combining the flexibility of programming generic purpose processors (traditional CPUs) with high performance special-purpose circuits (Application Specific Integrated Circuit, ASIC) to transmit packets at line speed.

A network processor (NP) is a system on a chip (SoC) designed to solve these problems, concentrating heterogeneous devices into a single integrated circuit, general-purpose processors, special-purpose programmable microcontrollers, switch units, cryptographic units, controllers and external memory, etc. [6].

The design of the NP makes it feasible to include ECD++ in the general-purpose processor (which in most cases operates with some embedded version of Linux) and then make use of the NP development libraries to communicate ECD++ with the specific purpose microcontrollers.

3.1. Intel IXP2400 Network Processor

The 2.5 Gbps Intel IXP2400 processor [7] is an example of an structured NP in two processing levels: the first with slow dynamics (Slow Data Path) which is a generic-purpose RISC processor XScale (ARM V5TE, 600 MHz, 32 bit, called *Core* processor), and the second with fast dynamics (Fast Data Path), which consists of a cluster of 8 programmable multithread RISC microcontrollers (600 Mhz., 32 bits, called MicroEngines, ME). [8].

The IXP2400 architecture allows the design of flexible reconfigurable rules engines in the Core, so that they can adapt dynamically without interfering with the ability of the ME to sustain their nominal packet processing performance [9].

The internal hardware architecture of the IXP2xxx family, uses 1 Core (XScale) and N MicroEngines (with N = 8 for the IXP2400) processing units.

Network packets enter and leave via the MSF (Media Switch Fabric) that provides access to external managers of the network physical layer. The packets are received, processed and transmitted by the code located in the ME (MicroCode). Only certain exceptional packages are scaled from the ME to the Core for special treatment.

The MEs are programmed in Assembler or MicroC (an adapted version of ANSI C), with a set of over 50 instructions that operate at the bit, byte and long-word levels, running on a six-stage pipeline and take an average of a single clock cycle to complete. The MEs have the 8 threads each, and do not have an operating system. Instead, system of hardware signals is provided, which manage context switching between threads ("hardware threads" technique), achieving zero latency context switching. These characteristics, together with certain special local quick-access registers, make it possible to handle packets at line speed. Additionally there is a time stamp system and timers that offer the ability to manage real time with an accuracy of 16 clock cycles (approximately 0.26 nanoseconds)

Probably the most notable highlight of the IXP2400 is the hierarchical organization of the various memory types (internal and external) with the possibility that it is shared between the ME and the Core (where ECD++ resides).

The memory types available are SRAM (8 Mbytes), DRAM (256 MByte) Scratchpad (16 Kbytes) and Local (2560 bytes per ME). SRAM and DRAM are static and dynamic RAMs respectively, external to the chip, and can be accessed both by the MEs and the Core. Scratchpad memory is a low latency for fast signaling between the ME threads from and to the Core, operating in the form of interruptions, data sharing and/or ring structures.

Low latency internal communication between MEs is critical, and for that purpose the Local memory is used, which is not accessible by the Core. There are also special registers called Next Neighbor (NN) that can be accessed only by 2 adjacent ME.

The MEs operate much faster than the external memory, completing instructions within one clock cycle. Then, a simple RAM memory access could block the execution for several clock cycles. Comparatively, while Local memory access consumes 1 clock cycle, access to the Scratchpad consumes 60 cycles, to the SRAM 150 cycles, and to the DRAM 300 cycles.

In summary, using specialized memory and data structures and replacement the operating system with hardware threads, the MEs provide a high performance comparable to that of ASICs, programmability comparable to that of CPUs, and communication with higher levels of slow dynamics (Core) to ensure manageability, treatment of exceptional cases and flexibility to changes in quality of service policies.

We will work with a network card Radisys ENP-2611 [13] as a development platform, a commercial product based on the IXP2400 NP which provides connectivity through 3 Gigabit Ethernet optical ports.

3.2. ECD++ Reference Architecture-IXA

The term IXP (Internet Exchange Processor) refers to a family of NPs that implement the IXA architecture (Internet eXchange Architecture).

IXA defines a standard framework for developing modular network applications based on reusable code blocks, reconfigurable, and portable between different NP. The basic idea is to define bounded responsibility layers and provide for each layer standard libraries accessible through well-defined interfaces (Application Program Interface, API). IXA covers both the software running on the Core and the ME, and Intel provides a lot of standard modules to use at both levels.

A network application based on IXA consists mainly of a chain of tasks (algorithms) applied sequentially to a stream of packets. Several tasks are distributed to various MEs using different load balancing strategies. Each task assigned to each ME has a software component associated with the Core, which is used to exchange control, measurement and management information.

IXA orders and structures the software architecture (libraries) orchestrate these multiplicity of code pieces running in parallel and asynchronous.

Figure 1 shows the relationship between the hardware architecture discussed in section 2 and the IXA software architecture.

Of all available libraries, the most interesting for ECD++ integration are: *Core Components* (CC), *Resource Manager* (RM) and *MicroBlocks* (MB). The CC APIs allows the creation of Linux kernel modules in the Core processor, which in turn use the resource management API (RM) to access shared memory structures between the Core and ME. However, the same physical memory location is referenced in a completely different way from the Core and from the ME. Therefore, the MEs use the MB libraries to access the shared memory using a pointer structure different from that used from the Core.

However, thanks to IXA the intricate communication details between the two types of processors that exist in the IXP2400 are completely transparent to developers of both code levels. Finally, any new features developed for the Core and/or the MEs will be reusable and portable to any processor that meets the IXA architecture.

As shown in **Figure 1**, ECD++ is located in the Slow Data Data. We developed a library called *ECD++I/O Core*, which consists of a *Core Component* that is invoked by ECD++ to communicate DEVS models with code that implements the network protocols in the Fast Data Path . We also developed the *ECD++I/O microblock* library at the MicroEngines level to communicate with DEVS models in the Core.

With this new software infrastructure, we can replace a DEVS models subsystem representing real network with ports communicating to and from the real packet processing system (Fast Data Path).

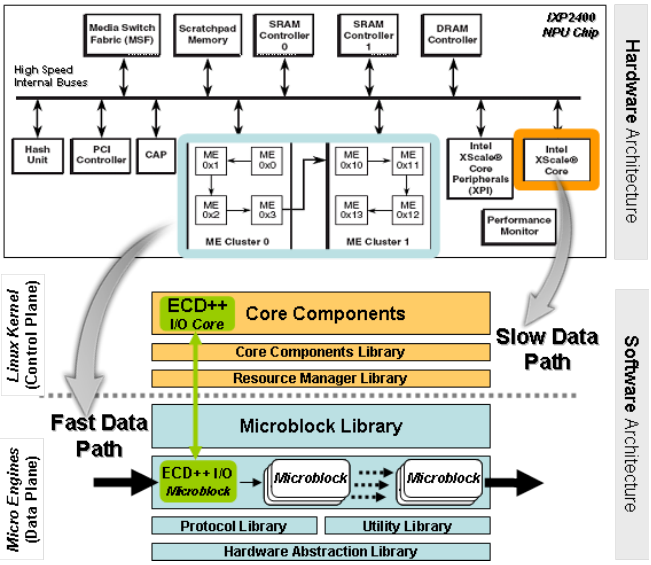

Figure 1 - ECD++ embedded on an Intel IXP2400 NP.

4. IMPLEMENTATION

In this section, we will describe here the adaptation of the real time ECD++ modeling and simulation environment, making it capable of running on the Core of an NP. Also, new libraries were developed for DEVS models to communicate with hardware layers specific for traffic management. Thus, embedded DEVS models control the hardware that manages network traffic, achieving a performance of such hardware-in-the-Loop (HIL).

4.1. Interface Library

The developed interface libraries allow any ECD++ atomic model to access variables in the space shared with the MEs by means of functions that are called SetIXAVar() and GetIXAVar(). Each atomic model that want to interact with the MEs must meet three requirements:

1. **Declare** as *IXA Variable* a parameter of the atomic model.
   This statement is made on the coupling .ma file that ECD++ uses to specify the structure of the complete coupled system with boot parameters of each atomic DEVS model.

2. **Include** the new *ECD++IXA_Library.h* library in the .cpp file of the atomic model.
   This enables the SetIXAVar(*IXA Variable*, ...) and GetIXAVar(*IXA Variable*, ...) functions to be invoked.

3. **Have an associated** block of microcode, which runs on the Microengines space, which **imports the variable** *IXA Variable* from the Core space.

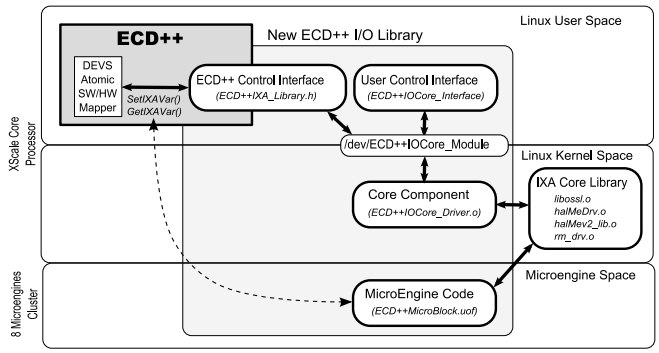

Figure 2 - New libraries ECD++ Input / Output for IXA.

Communication is resolved in a transparent manner for the designer of the atomic model, through the *ECD++I/OLibrary* library modules for IXA shown in **Figure 2**. In this figure, a DEVS atomic model called *SW/HWMapper* uses the communication infrastructure developed to share variables with *ECD++Microblock* microcode. Thus, Mapper models developed to manage access to various external variables shared with IXA, encapsulating this functionality, and away from the dynamic specification of models that define the rest of the system.

A command line interface for accessing the IXA Variables as developed, called *ECD++IOCore_Interface*, with the purpose of monitoring and/or manual testing. This interface can be used simultaneously with the execution of ECD++, operating on the same variables.

4.2. Interface Automatic Generation

Since the IXA communication interfaces provide services to a set IXA variables defined by the user, libraries must regenerate as you adjust the choice of these variables (added, removed or renamed). According to this need, we designed a configurable mechanism for the generation of libraries, which can automatically detect the IXA Variables chosen by the modeler (interpreting the .ma coupling file)

As shown in **Figure 3**, from choosing the IXA Variable name to obtaining all necessary infrastructure, five steps are required to make libraries accessible from a DEVS atomic model.

Steps 1 and 2 relate only to ECD++, and were mentioned in the previous section as the first and second requirements for a DEVS model with ability to access IXA variables. These are simple steps that are part of the usual practice of modeling with ECD++ and can be solved with a text editor.

Steps 3 and 4 imply automatic generation of libraries, and require the use of new developed tools, consisting mainly of Perl scripts.

Step 5 is part of the usual process of compiling the ECD++ simulator, regardless of the use of IXA Variables in an atomic model. However, the binary generated will be unique to the hardware architecture of the target embedded processor (in this case, IXP2400).

This procedure assumes that you always have a microcode to run on the Microengines, which explicitly states its intention to share the IXA Variables with the Core space. Microcode technical characteristics will not be described in this thesis as it escapes the central focus of this paper.

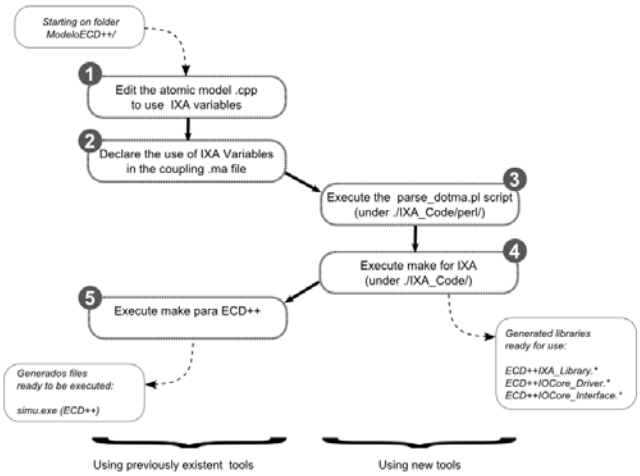

Figure 3 - Automatic generation of interface libraries ECD++IXA* and ECD++IOCore*

This mechanism automates and greatly facilitates the declaration of variables between ECD++ and Microengines, hiding a large number of low-level implementation details. The following example shows the result of applying the steps above to a real case with the IXP2400 processor.

4.3. Event Counter

Now we will show the procedure explained previously applied to a simple practical example, emulating a traffic event monitor. The example handles two IXA variables from ECD++: one that provides information about the number of occurrences of certain traffic events, and one that provides the ability to restart the counter. They are called *EventCounter* and *ResetCommand* respectively. *EventCounter* will be updated by the MEs, as monitored events occur. Each time the *ResetCommand* is sent, the *EventCounter* is restarted to zero and it may carry a code number that is interpreted as a command by the microcode block *ECD++MicroBlock* (which ECD++ interacts with). The microcode will increase *EventCounter* by one whenever a particular event happens. In order to validate this example, these events will be artificially forced from the MEs, i.e., they are not associated with real network traffic events (as it could be, for example, the arrival of a corrupt network packet).

The following Listing shows the coupling .ma file:

```
[top]
components                                    :
eventCounter@trafficEventCounter
out : counterStatus
Link : counter@eventCounter counterStatus
[eventCounter]
pollingPeriod : 00:00:02:000
counterLimit  : 30000000
EventCounter  : 0 %%useIXAVar[0]
ResetCommand  : 7 %%useIXAVar[0]
```

Figure 4 - IXA_EventCounter.ma Coupling Model

It consists of a single atomic model *eventCounter* (a particular instance of the *trafficEventCounter* model type). It has an output port called *counter*, which sends values to another output port, this time from the *top* coupled model

(the highest in the system hierarchy). The atomic *eventCounter* is parameterized with 4 variables that affect its dynamics, two of which are declared as IXA variables using the comment *useIXAVar[0]* indicating that they are shared with microcode running on the the MicroEngine 0. This model will consult the Microengine each pollingPeriod time units the value of the amount of traffic events occurred. The value is accessed through the *EventCounter* IXA variable, which is initialized to 0. The CounterLimit parameter (initialized at 30000000) indicates that when *EventCounter* exceeds this value, the model must send a *ResetCommand* with value 7. This command will be interpreted by the microcode, which will restart the events account from 0.

The following Listing shows the sections and the main code lines of the trafficEventCounter.cpp file, related to IXA variables and with the expected behavior described above.

```
#include "trafficEventCounter.h"
#include "ECD++IXA_Library.h"
...
// *** Constructor
trafficEventCounter::trafficEventCounter(..
.):Atomic(name),counter(addOutputPort("coun
ter")){}
// *** Model Initialization ***
Model  &trafficEventCounter::initFunction()
{
...
 this->EventCounter       =       str2Int(
MainSimulator::Instance().getParameter(
description(), "EventCounter" ) ) ; // IXA
Varaible
 this->ResetCommand       =       str2Int(
MainSimulator::Instance().getParameter(
description(), "ResetCommand" ) ) ; // IXA
Varaible
...
 this->state = Sleeping;
 holdIn(active,  this->pollingPeriod);  //
Programs itself for the first poll
...}
//  *** DEVS External Transition Function
***
Model
&trafficEventCounter::externalFunction(...)
{
...}
//  *** DEVS Internal Transition Function
***
Model
&trafficEventCounter::internalFunction(...)
{
 switch (this->state) {
  case Sleeping:
...
   this->currentCounter                    =
GetIXAVar("EventCounter");
   this->state = Notifying;
   holdIn( active, 0); // Programs itself
for emmiting inmediately the obtained data
 case Notifying:
...
  if   (   this->currentCounter   >   this-
>counterLimit ) {
   SetIXAVar("ResetCommand",
strResetCommand);}
  this->state = Sleeping;
  holdIn(active,  this->pollingPeriod);  //
Programs itself for the next poll
...}
// *** DEVS Output Function ***
Model
&trafficEventCounter::outputFunction(...) {
 switch (this->state) {
...
  case Notifying:
   sendOutput(msg.time(),  counter,  this-
>currentCounter);  //  Emit  the  observed
information
...}
```

Figure 5 – trafficEventCounter.cpp Atomic model definition

The library included in line 2 provides the interface to SetIXAVar() and GetIXAVar(). Row 5 declares the *counter* port which will output values read from the Microengine. During initialization of the model, in rows 9 and 10 the initial values for the IXA variables are taken from IXA_EventCounter.ma. The model can be in two states defined by the *state* variable: Sleeping and Notifying. At the Sleeping state the model waits a pollingPeriod time to send the next query of EventCounter. In row 12, the model starts in Sleeping state, and in row 13 it waits for a pollingPeriod time units. When time runs out in the Sleeping state, line 23 reads the new value of the counter with GetIXAVar("EventCounter") and row 24 changes the model to Notifying state. Row 25 forces an immediate internal transition to run the output function and sent though the eventCounter port the value stored in currentCounter. In the Notifying phase (with duration zero, a transient state) the last counter value is first sent (row 38, outputFunction), then the internal transition (line 28), decides whether or not to send a command reset. If so, line 29 invokes SetIXAVar("ResetCommand", strResetCommand) which writes the IXA variable ResetCommand causing the microcode to react to change. The StrResetCommand value should be 7 according to the file specified in IXA_EventCounter.ma, and imported into the atomic model in row 10. Then in rows 30 and 31 change the model back to the Sleeping state for pollingPeriod units of time, restarting the cycle. This model has no input ports, so it does not use the external transition function.

The execution of the IXA_EventCounter embedded real-time system is shown in the following Figure:

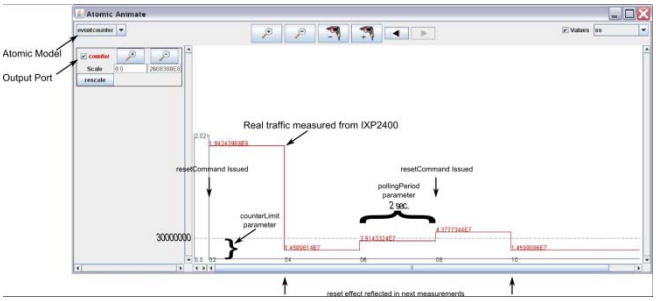

Figure 6 – Running a result of ECD++ models in IXP2400. Interaction with Real-Time MicroEngines.

It shows 10 seconds of execution, in which the atomic model eventCounter detected 2 times that the MicroEngines event counter exceeded the limit established by counterLimit. In such occasions, the system sends a reset command, which is reflected in a drop of the counter below the threshold for the subsequent measurement period. This way the model is validated and its effective interaction with the low-level specialized hardware.

5. ADVANCED TOOLS AND METHODS

In this section, new tools to implement advanced real-time embedded DEVS models using ECD++. The particular aim is to obtain practical tools that simplify the design process of embedded systems for network control. We will show a new visual interface for modeling and simulation with ECD++ and a portal virtual laboratory is implemented for developing DEVS models and embedded them in the NIC RadiSys ENP-2611 IXP2400 based.

5.1. Eclipse based Visual Modeling Tool

In ECD++ coupled models are defined using a high-level declarative language, while the atomic models can be defined both declaratively using DEVS-Graphs and programmatically using C++. Some previous ECD++ tools allow the graphical creation of coupled and atomic DEVS models and support visualizing the results of a simulation [10]. Nevertheless, these tools have several limitations: they are independent developments that do not use standard interfaces, they have several have several general usability shortcomings, they have no distribution mechanisms for updates, they are difficult to extend, they are decoupled from the simulation environment, and they use different formats for representing the models, which makes it difficult to visualize pre-existing models and making it difficult to maintain the graphical definition consistent with executable models.

According these difficulties, the CD++ Builder [11] tool was developed, an Eclipse plugin [12] that solves the above problems. The integration of various associated tools within the same environment reduces the learning curve for new users and simplifies the modeling and simulation of DEVS models, avoiding the need for multiple applications. The new CD++Builder graphical editors for coupled and atomic models solve the usability problems through standard Eclipse GUI, and allow users to specify complex DEVS models without programming, reusing existing libraries. These new features allow opening model definitions created with previous tools and automatically generate a graphical representation, which is very beneficial as the vast models in the library are already implemented and tested, but they did not have a graphical representation that simplifies the understanding of its structure and operation. Additionally, since Eclipse is cross platform, CD++Builder allows you to develop DEVS models in both Linux and Windows systems. On the other hand, Eclipse is a standard widely adopted platform for developing IDEs, providing a familiar interface and easy to understand for new users and provides a framework designed to be extensible, making it easy to add new DEVS tools integrated with pre-existing ones, within the same platform.

Though several of CD++Builder features could be very useful for developing ECD++ models, it was originally developed based on CD++ standalone version, and some of its features are not suitable for developing embedded real time models. Additionally, while the directory structure and file naming is intended to be clear and self-explanatory, the manual build process described on **Figure 3** is error-prone, especially for users without previous experience, and imply running several console-based commands outside the development environment.

To overcome these limitations new functionalities were developed, enabling ECD++ DEVS models to be graphically created and visualized using CD++Builder tool and simplifying and automatizing the build process for the embedded simulator. The following figure shows CD++Builder including the new features that target the development of DEVS embedded IXA models:

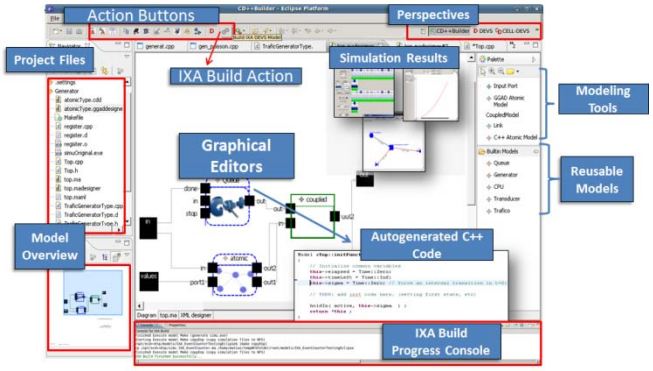

Figure 7 – CD++Builder Simulation Environment to develop embedded systems with ECD++.

The graphical editor for DEVS coupled models (shown on **Figure 7**, center) was extended to support the new IXA variable declaration in coupling .ma files. This allows atomic models that make use of IXA variables to be represented graphically and visualization of IXA ECD++ simulation results (shown on **Figure 7**, top right) is now possible. Additionally, these features were integrated with CD++Builder update site, which lets you install and download plugin updates from a single point accessible to all users, facilitating the distribution of new tools and solutions to bugs.

On the other hand, the CD++Builder simulation compilation process was re written to support ECD++ simulator and to execute the special tasks described on **Figure 3**, necessary for generating the IXA libraries. In this sense, two new functionalities were incorporated to CD++Builder: the ability to run automatically the ECD++ build process (using an action button shown on **Figure 8** a.), which generates the IXA libraries, and the possibility to customize the build process, configuring the different parameters involved (shown on **Figure 8** c.).

Regarding the build process, now CD++Builder automatically performs the all the steps previously described to generate the IXA libraries and the ECD++ simulator executable. This reduces to minimum the error possibilities when running the IXA ECD++ build tasks, and it expedites the modeling and experimentation process, as it removes the need of manually executing commands, and allows the build process to be run within the same modeling tool used to develop the DEVS models and visualizing simulation results. The build progress and the result of each of the tasks is shown on Eclipse's console (shown on **Figure 8** b.), which allows to track errors (printed in red), the executed commands, and successful builds (printed in green).

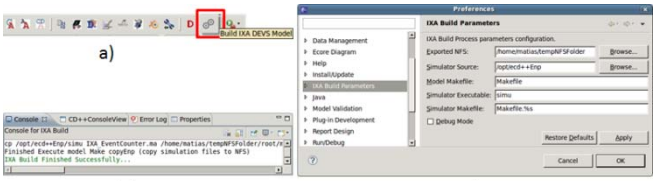

Figure 8 – New CD++Builder GUI for the ECD++ build process. a) IXA build action button b) IXA build progress console c) IXA build configuration preferences.

The build process is by default set to run within the experimental environment (described in the following section), which involves several configurations that must be accurately set. As the tool is intended to be used on any system with the necessary libraries installed, these configurations are parameterized and can be updated using Eclipse's standard preferences GUI. This allows several variables, file names; folder paths, etc. involved in the build process to be easily configured, such as ECD++ source code path, the NFS exported folder shared with the network processor, etc., as shown on **Figure 8** c.

The new tool presented here will be used to design, test and display results in Example 2 of Section 5.3, which has a supervisory QoS control implemented in ECD++.

5.2. Virtualized Lab

Experimentation with the IXP2400 processor and the reference architecture IXA offers great power and flexibility to develop embedded applications for network control. However, installation and commissioning of the many tools and libraries needed for a productive laboratory consist of tedious tasks, very error-prone, and in many cases requiring expert assistance. The operations can expend full working weeks. This situation attempts against the experimental environment replication and its adoption by engineers, teachers and students who wish to develop applications based on DEVS models for that scenario.

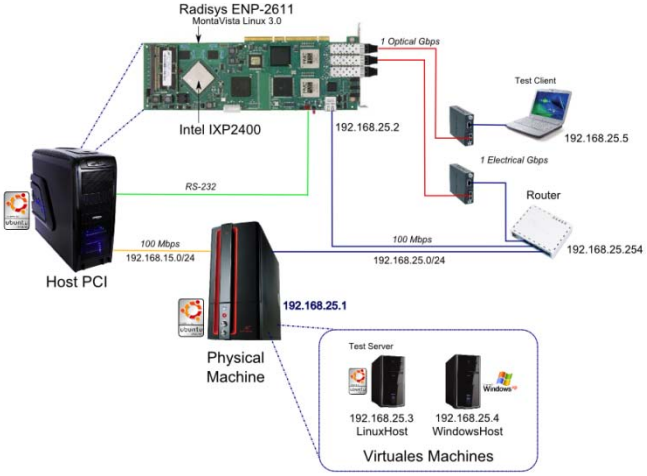

Figure 9 – Logical-Physical Conceptual Diagram: Laboratory for Virtual Machines based on development and IXP2400 ECD++ including new integration library.

Figure 9 shows a scheme that combines logical and physical information of the newly developed reference laboratory and the software components that are located in each node. The software components are primarily those provided by Intel and RadiSys, involving three operating systems: Embedded Linux MontaVista, general-purpose Linux and Windows. The installation and adjustment of the parameters is based on multiple software packages provided by manufacturers, representing a complicated process.

To solve this problem, we made all the necessary installations and configurations in easily portable virtual machines. LinuxHOST and WindowsHOST are virtual machines running on one physical machine. In another separate physical machine (Host PCI) the RadiSys ENP-2611 board is connected in a PCI slot, which is administered via a dedicated subnet and a text terminal via RS232. This creates a pre-installed and fully replicable laboratory, virtually eliminating any preliminary effort to develop models for IXP2400 with the CD++Builder tool.

The WindowsHOST is needed because the development of the MicroEngines microcode for of IXP2400 is performed with an advanced development and debugging environment provided by Intel (Intel Developer Workbench), which runs only under Windows XP.

As shown in the figure, the LinuxHOST virtual machine includes ECD++ and the new tools presented in this paper: libraries ECD++/IXA and advanced GUI CD++Builder.

When the design of an control embedded system based on DEVS is completed (obtained using CD+++Builder in the LinuxHOST), and the microcode that handles actual network packets is already available (obtained using the Developer Workbench in the WindowsHOST), both binary files are "downloaded" to the processor IXP2400 through new automated scripts. Then, you are ready to perform real-time testing and analyzing the results using the log files generated by ECD++ on the Linux MontaVista, which are accessible via the NFS server host mounted on the LinuxHOST.

The practical example presented below was developed entirely using the new virtual laboratory presented in this section.

5.3. Example 2. Supervisory Control based on Service Quality Measurement Rate

As a case study of the model-based embedded simulation methodology, we design a Supervisory Control System, which accepts configurable global policies and monitors the traffic rate (TR) of a real packet flow.

Depending on the global policies and the state of TR, the system sends updated control information to low-level algorithms that decide and execute the packet drop. The monitoring and control information to and from these low-level algorithms is done through input/output ports. The IXP2400 architecture allows building flexible reconfigurable rules engines, located on the NPs which can be adapted dynamically without interfering with the ability to sustain the IXP2400 nominal packet processing performance.

The incremental methodology proposes as a first step the verification of the system behavior completely simulated in ECD++ under a simplified scenario, with DEVS models running on the XScale Core but without interacting with the ME. **Figure 10** shows a conceptual block diagram of the models on the control application. The Actuator and Traffic Sensor QoS models in the low speed level are responsible for sending control commands and sense the state of TR, respectively. These blocks communicate with their counterparts in the high speed level (or Packet Processing System), consisting of the Shaper and Metering System QoS models. Having verified the basic functionality of the QoS Controller in this simplified scenario using synthetically generated traffic with DEVS models, we proceed to the validation stage of the controller, communicating the low speed levels (DEVS models are kept intact) with the real high speed hardware (MiroEngines). In **Figure 10**(right) the real packet processing hardware replaces the set of models that emulate its functions (Generator, Shaper, Pipeline, Meter, Consumer). This is possible thanks to new Mapper DEVS models (trafficShaper and trafficMeter) that act as an interface with IXP2400 through IXA Variables.

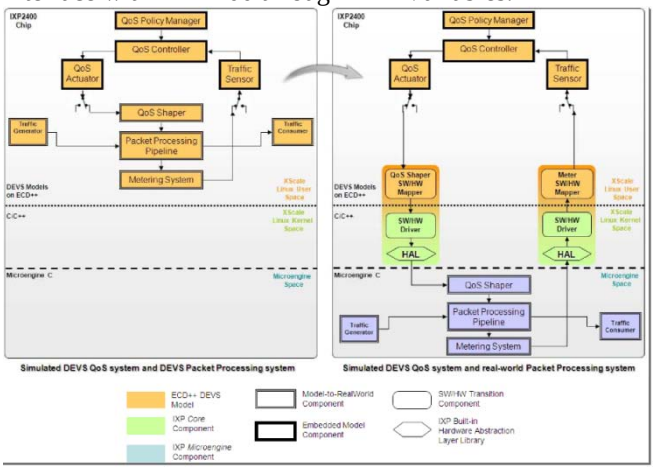

Figure 10 – Transition from an autonomous embedded simulation (left, verification phase) to a Hardware-In-The-Loop simulation (right, validation phase).

It is assumed that there is a low-level algorithm running on the MicroEngines with the ability to apply Admission Control (AC) techniques that drop packets to ensure an average queue length. This length is an operation parameter that is updated on each action control from the Supervisory Control. On the other hand, the MEs can make available to the Supervisory Control the updated information of the number of packets transmitted at every moment.

The quality system that is designed with ECD++ sends commands to control the AC, indicating the new desired average queue length. This indication, in turn, may vary according to the policy configured on the controller. **Figure 11** shows a state machine representation of this model. The 4 states reflect the condition of the packet processing system, combining the TR metric with the amount of uninterrupted time T (time persistence) during which the given TR level is maintained. The figure shows the complete visual design of the control system using the CD++ Builder advanced tool, being ECD++ compiled and ready to interact with the ME. This phase represents the right of **Figure 10** the practical case corresponding to the validation conceptual stage.

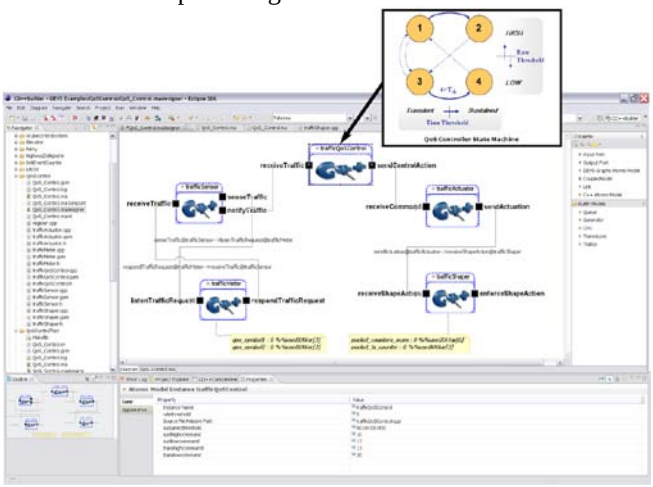

Figure 11 – Advanced editing tool for ECD++ models. QoS_Control model for IXP2400.

The controller uses the *rateThreshold* traffic rate threshold to classify the intensity of traffic (high or low). Initially, if *TR>rateThreshold*, the model transitions to state 1 and the time T begins to increase from zero. If instead TR ≤ *rateThreshold*, the controller transitions to state 3. We define the *sustainedThreshold* time threshold to distinguish between 2 possible persistence values for the TR rate: Transient and Sustained. When T crosses the *sustainedThreshold* threshold, the model evolves from 1 to 2 depending on whether the system was in a high TR level or low TR level, respectively. When the traffic rate crosses the *rateThreshold* threshold T is reset to zero and the state returns 1 or 2, as appropriate.

Both thresholds are defined as parameters for the trafficQoSControl atomic model, as shown in the Properties tab at the bottom of **Figure 12**, where it sets up *sustainedThreshold = 6* second and *rateThreshold = 5* packets/second.

In the same place the parameters *transHighCommand = 23*, *sustHighCommand = 10*, *transLowCommand = 20* and sustLowCommand = 17 are shown. These are the commands to be sent to the AC every time the driver experience a change in its internal machine states (for states 1, 2, 3 y 4 respectively). The responsible for sending these commands is the trafficActuator block, which ignores the difference between a standard DEVS model and a DEVS model Mapper capable of communicating

with MicroEngines. This role is accomplished by the trafficShaper block, which declares the IXA variables needed to invoke the function SetIXAVar ().

Following the steps outlined in previous sections, the executable file is generated for the QoS_Control system incorporating the new libraries to communicate with the MicroEngines, and the IXP2400 was ported for validation testing. The experiment consists on placing a traffic generator in the LinuxHost server of the virtual laboratory (IP 192.168.25.3, see **Figure 6**) generating packets to a Test Client (192.168.25.5). The route implies that the traffic will pass over the RadiSys ENP-2611 and the consequent action of the designed control system to reacting to real traffic measurement package.

The result is shown in **Figure 12**, where the measured TR average level by trafficSensor/trafficMeter every 1 second can be verified (upper temporal sequence, receiveTraffic port). According to the expected behavior of the state machine, we can validate the command sequence sent to the MicroEngines (lower temporal sequence, receiveShapeAction port). In turn, the temporal sequence of the receiveCommand port shows the states that the trafficQoSControl state machine transitions to.

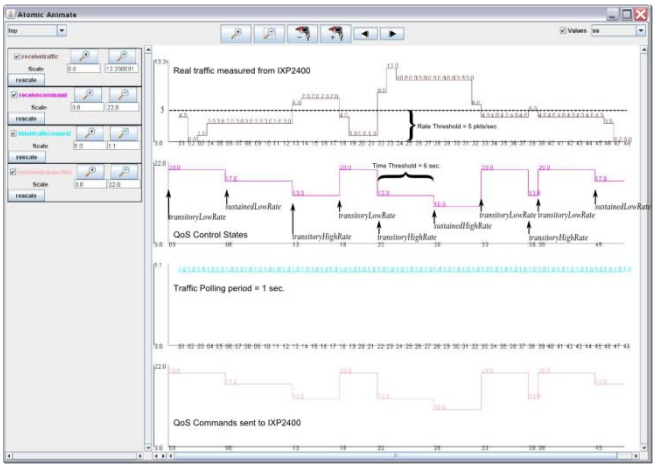

Figure 12 – Real time simulation results with real traffic Hardware In The Loop.

The expected behavior is observed for the Supervisory Control, reacting with the proper sending of commands when thresholds are exceeded. This way the system is validated, both to changes in sensed rate level (crossing the threshold rate of 5 packets / second) as well as traffic sustained at longer levels than the sustained established threshold (6 seconds).

6. CONCLUSIONS

We introduced contributions for the implementation of real-time embedded network controllers, and experiments on real platforms. Sufficient tools and methodologies were provided to obtain a completely intuitive process based on DEVS models that can be embedded in a final target hardware, implemented in real-time, and interconnected with specialized packet processing hardware. Continuity of DEVS models that define a controller is ensured, by encapsulating specific interaction functionality with external hardware in special atomic models (adaptable for interacting with communication interfaces of each device). This was possible thanks to the development of new libraries that communicate ECD++ residing on the IXP2400 Core processor with the IXP2400's MicroEngine cluster. New advanced graphic tools assist in the process of designing embedded systems for network control. We designed a portable virtualized laboratory that allows fast and simplified experiments with complex configuration settings. The practical examples studied show that the final implementation stages and validation of DEVS based network drivers can be carried out successfully in real-time, completely eliminating the need for adapting the logic and structure when moving from independent simulations (for functional verification purposes) to Hardware-In-The-Loop executions (for final validation and implementation). The methodology ensures the continuity of the models and promotes the development of complete engineering solutions based on modeling and simulation.

7. REFERENCES

[1] M. Bozga A. Basu and J. Sifakis. Modeling heterogeneous real-time components in BIP. In Proceedings of SEFM 2006, New York, NY, USA, 2006.

[2] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity-the Ptolemy approach. Proceedings of the IEEE, 91(1):127–144, 2003.

[3] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing applications using model-driven design environments. Computer, 39(2):33–40, 2006.

[4] J. Brandt and K. Schneider. How different are Esterel and SystemC. Embedded Systems Specification and Design Languages, pages 3–13, 2008.

[5] D. Huang and H. Sarjoughian. Software and simulation modeling for real-time software-intensive systems. In Distributed Simulation and Real-Time Applications, 2004. DS-RT 2004. Eighth IEEE International Symposium on, pages 196–203. IEEE, 2004.

[6] Douglas E. Comer. Network Systems Design Using Network Processors: Intel 2XXX Version. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[7] Intel. Intel IXP2400 Network Processors. Intel Press, 2004.

[8] B. Carlson. Intel Internet Exchange Architecture and Applications: A Practical Guide to Intel's Network Processors. Intel Press, 2003.

[9] A. Gavrilovska, S. Kumar, and K. Schwan. The execution of event-action rules on programmable network processors. In Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS 2004), held with ASPLOS-XI, 2004.

[10] G.A. Wainer. Discrete-Event Modeling and Simulation: A Practitioner's Approach. CRC Press (in print), 2009.

[11] Matias Bonaventura, Gabriel A. Wainer, and Rodrigo Castro. Advanced ide for modeling and simulation of discrete event systems. In Proceedings of 2010 Spring Simulation Conference (Spring-Sim10), DEVS Symposium. SCS, April 2010.

[12] F. Budinsky, S.A. Brodsky, and E. Merks. Eclipse modeling framework. Pearson Education, 2003.

[13] Radisys. ENP-2611 Data Sheet. http://www.radisys.com/products/datasheets/ENP-2611.pdf, 2006.

Referencias

[1] Federico Bergero and Ernesto Kofman. PowerDEVS. A Tool for Hybrid System Modeling and Real Time Simulation. *Simulation: Transactions of the Society for Modeling and Simulation International*, 2010. In Press.

[2] Federico Bergero, Ernesto Kofman, C. Basabilbaso, and J. Zúccolo. Desarrollo de un Simulador de Sistemas Híbridos en Tiempo Real. In *Proceedings of AADECA 2008*, Buenos Aires, Argentina, 2008.

[3] Matias Bonaventura, Gabriel Wainer, and Rodrigo Castro. Advanced IDE for Modeling and Simulation of Discrete Event Systems. In *Proceedings of 2010 Spring Simulation Conference (SpringSim10), DEVS Symposium*. SCS, April 2010.

[4] F. Budinsky, S.A. Brodsky, and E. Merks. *Eclipse modeling framework*. Pearson Education, 2003.

[5] B. Carlson. *Intel Internet Exchange Architecture and Applications: A Practical Guide to Intel's Network Processors*. Intel Press, 2003.

[6] Rodrigo Castro, Iván Ramello, Matias Bonaventura, and Gabriel Wainer. Model-based Design and Implementation of Embedded Real-Time Network Controllers, 2010.

[7] F.E. Cellier and Ernesto Kofman. Continuous System Simulation. Springer, New York, 2006.

[8] Douglas E. Comer. *Network Systems Design Using Network Processors: Intel 2XXX Version*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[9] Radisys Corporation. ENP Software Development Kit Quick Start Guide, December 2005.

[10] Radisys Corporation. ENP-2611 Data Sheet, 2006.

[11] A. Gavrilovska, S. Kumar, and K. Schwan. The execution of event-action rules on programmable network processors, 2004.

[12] N. Giambiasi, B. Escude, and S. Ghosh. *GDEVS: A generalized Discrete Event specification for accurate modeling of dynamic systems*, chapter 17(3):120–134. Transactions of SCS, 2000.

[13] Donald Hooper. *Using IXP2400/2800 Development Tools: A Hands-On Approach to Network Processor Software Design.* Intel Press, 2003.

[14] Red Hat Inc. *RedBoot Users Guide R1.24.* Red Hat Corporate, August 2001.

[15] Intel. *Intel XScale Core Developers Manual.* Intel Press, December 2000.

[16] Intel. *Intel Internet Exchange Architecture Portability Framework Reference Manual.* Intel Press, November 2003.

[17] Intel. *Intel IXP2400/IXP2800 Network Processors Software API Reference Manual.* Intel Press, January 2003.

[18] Intel. *Intel IXDP2400 Advanced Development Platform System Users Guide.* Intel Press, November 2004.

[19] Intel. *Intel IXP2400 Network Processors.* Intel Press, 2004.

[20] Intel. *Intel IXP2400 and IXP2800 Network Processor Programmers Reference Manual.* Intel Press, July 2005.

[21] Erik Johnson and Aaron Kunze. *IXP2400/2800 Programming: The Complete Microengine Coding Guide.* Intel Press, 2003.

[22] Ernesto Kofman. Introducción a la Modelización y simulación de Sistemas de Eventos Discretos con el Formalismo DEVS. *Notas de Clase - Licenciatura en Ciencias de la Computación. FCEIA, UNR, Argentina*, 2000.

[23] James Nutaro. *Parallel Discrete Event Simulation with Application to Continuous Systems.* PhD thesis, The University of Arizona, 2003.

[24] Gabriel Wainer. CD++: a toolkit to develop DEVS models. *Software – Practice and Experience. Vol. 32, pp. 1261 – 1306.*, 2002.

[25] Gabriel Wainer. *Discrete-Event Modeling and Simulation: A Practitioner's Approach.* CRC Press (in print), 2009.

[26] Gabriel Wainer, Ezequiel Glinsky, and Peter MacSween. *A Model-Driven Technique for Development of Embedded Systems Based on the DEVS Formalism*, chapter 7. Model-driven Software Development - Volume II of Research and Practice in Software Engineering. S. Beydeda and V. Gruhn eds. Springer-Verlag, 2005.

[27] Gabriel Wainer and Yinfeng Henry Yu. ECD++: an engine for executing DEVS models in embedded platforms. *Society for Computer Simulation International San Diego, CA, USA*, 2007.

[28] Bernard Zeigler. Theory of modeling and simulation, 1976.

[29] Bernard Zeigler, T. G. Kim, and H. Praehofer. *Theory of Modeling and Simulation.* 2nd. edition. Academic Press, New York, 2000.

[30] Bernard Zeigler and Hessam Sarjoughian. Introduction to DEVS Modeling and Simulation with JAVA: A Simplified Approach to HLA-Compliant Distributed Simulations.