# UNIVERSIDAD DE CANTABRIA




Departamento de Tecnología Electrónica, Ingeniería de Sistemas y Automática

# Especificación Heterogénea y Generación Automática de Software desde SystemC para Sistemas Embebidos

**Memoria** presentada para optar al grado de

DOCTOR INGENIERO DE TELECOMUNICACIÓN

por Fernando Herrera Casanueva,

Ingeniero de Telecomunicación,

Especialidad Microelectrónica

Santander, 2008.

# UNIVERSIDAD DE CANTABRIA

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Departamento de Tecnología Electrónica, Ingeniería de Sistemas y Automática


# Especificación Heterogénea y Generación Automática de Software desde SystemC para Sistemas Embebidos

**MEMORIA**

presentada para optar al grado de
DOCTOR INGENIERO DE TELECOMUNICACIÓN
Por el Ingeniero de Telecomunicación,


Fernando Herrera Casanueva

**EL DIRECTOR**
D.Eugenio Villar Bonet
Catedrático del Departamento de
Tecnología Electrónica, Ingeniería de Sistemas y Automática
Universidad de Cantabria


Santander, 2008

# Agradecimientos

A mi familia le debo lo que soy, lo que sé y el tiempo que he dedicado a este trabajo. Lo que valga este trabajo, es también suyo. Gracias a Tere, Bernardino, Maite, Berna, a ti Emma, que eres ahora mi familia del día a día, y al resto de mi familia. También un trocito es de aquellos que os considero mis amigos fuera de mi trabajo: Cristian, Fonso, Marce, Marcos, Michi, Óscar, Jaime, y un, espero, largo etcétera.

A mi tutor de tesis, el catedrático Eugenio Villar, agradezco la dirección de este trabajo. Es precursor y partícipe de esta investigación. Destacaría su orientación en las líneas estratégicas, la planificación de los trabajos (que siento haber roto alguna vez) y el consejo ofrecido respecto a cursos, seminarios, conferencias y publicaciones. No sin numerosas discusiones técnicas y revisiones de estilo, orientación y contenido de mis publicaciones, a la postre también suyas, éstas hubieran alcanzado la calidad que acreditan. También le agradezco la formación en aspectos que, creía, iban más allá de la investigación, pero que son indispensables para asimilar una mínima capacidad de eficacia, síntesis y estructuración en la labor investigadora. Espero que este trabajo responda a la confianza que ha depositado en mí hasta la fecha.

A mis compañeros de laboratorio del GIM (Adrián, Rubén, Yolanda, Carlos Iván, Bea, María, Susana, Chuchi, Miriam, Javi, Juan, Edu, Iván, David, etc), agradezco el buen ambiente de trabajo y el apoyo en las tareas más cotidianas. Además de escucharme, he aprendido mucho de ellos. Además de la camaradería, a los profesores del GIM: Maite, Pablo, Víctor, Héctor e Íñigo, les agradezco sus ideas, discusiones técnicas, aportaciones y, en algunos casos, clases. Quiero reconocer también el trato, consonante con el del resto de compañeros, y el trabajo del personal de administración (Sara y Bea) y técnico (Antonio y Fran) del departamento TEISA, indispensable en nuestras tareas de investigación.

Agradezco a los profesores A. Jantsch, I. Sander y al resto de miembros del laboratorio ECS de la Universidad Técnica Real (KTH), su agradable acogida en Estocolmo y las interesantes discusiones que he podido mantener con ellos. También a otros colegas y profesores de OFFIS, TUV, DS2 y Thales. Entre todos han influido notablemente en este trabajo y espero que también en colaboraciones actuales y futuras.

Agradezco asimismo la atención del personal de la biblioteca de la UC y a aquellos que desinteresadamente me han aclarado dudas por correo, en foros, cursos y conferencias. Agradezco y animo también en general a los autores que han invertido un esfuerzo adicional en hacer que sus artículos, documentos y libros sean, además de públicos, autoexplicativos y didácticos. Esto ahorra mucho tiempo a un investigador.

Finalmente, este trabajo no hubiera sido posible sin la financiación de la Consejería de Cultura del Gobierno de Cantabria (España), a través de una beca predoctoral, y de distintos proyectos del MITyC del Gobierno de España y Europeos (FEDER, ToolIP y ANDRES) en los que he estado involucrado como miembro del Grupo de Ingeniería Microelectrónica (GIM) de la Universidad de Cantabria.

"*Somos como enanos a hombros de gigantes.*" – Bernardo de Chartres

"*En fin, señor y amigo mío —proseguí—, yo determino que el señor don Quijote se quede sepultado en sus archivos en la Mancha, hasta que el cielo depare quien le adorne de tantas cosas como le faltan, porque yo me hallo incapaz de remediarlas, por mi insuficiencia y pocas letras, y porque naturalmente soy poltrón y perezoso de andarme buscando autores que digan lo que yo me sé decir sin ello.*" – Prólogo del Quijote – Miguel de Cervantes

"*Hazlo siguiendo el libro, pero sé tú el autor.*" – Quinta ley de Peter

*"La mejor forma de aburrir es contarlo todo."* – François Marie Arouet (Voltaire)

# Resumen de la Tesis

Los sistemas embebidos tienen una gran importancia en el mercado de sistemas electrónicos. Son tan ubícuos como transparentes, ya que usualmente se incrustan en sistemas mayores en los que realizan una función específica. En otros casos, como el de los teléfonos móviles, son productos finales con múltiples capacidades de entrada/salida. Habitualmente, los sistemas embebidos actuales son sistemas hardware/software (HW/SW), lo que permite establecer un compromiso específico entre costo, velocidad, consumo y otros factores cruciales en este tipo de sistemas.

El diseño de sistemas embebidos está afectado por un problema conocido como brecha de diseño. Este problema consiste en que la productividad de los diseñadores, aunque aumenta, lo hace a un ritmo menor que las rápidamente crecientes capacidades de integración de los dispositivos electrónicos. En la última actualización del *International Roadmap for Semiconductors (ITRS'07)*, se mostró por primera vez que esa brecha es debida, no sólo a una productividad insuficiente en el dominio hardware, sino también a una más relevante insuficiencia productiva en el dominio software.

El propio ITRS ha apuntado la solución: un salto cualitativo desde las metodologías de diseño de sistemas HW/SW clásicas, que se han erigido en una disciplina específica conocida como co-diseño HW/SW, hacia lo que se denomina diseño electrónico de nivel de sistema (ESL). El diseño ESL centra y hace partir las actividades de diseño desde la especificación de nivel de sistema. La especificación de nivel de sistema describe y documenta el sistema en un nivel más abstracto. Además, una metodología ESL debe permitir realizar diversas actividades del diseño, tales como la verificación y la exploración del espacio de diseño a partir de la especificación del sistema, y por tanto en un alto nivel de abstracción. Una metodología ESL debe habilitar también un flujo de implementación automático desde la especificación. Otro punto clave es el método de captura o lenguaje que se emplea en la realización de la especificación de sistema. Se precisa un lenguaje común, de sintaxis familiar, comprensible y lo suficientemente potente para los distintos dominios de diseño. En ese contexto, SystemC es un lenguaje estándar que se ha destacado como el candidato principal para la especificación de nivel de sistema de sistemas embebidos.

Sin embargo, existen aún algunas carencias que impiden una aplicación eficaz y eficiente de SystemC en un flujo de diseño ESL de sistemas embebidos. Entre ellas, hay que destacar la ausencia de una metodología de especificación sistemática que oriente al usuario en el uso de SystemC y que soporte heterogeneidad. El soporte de heterogeneidad en las metodologías de especificación de sistemas complejos es esencial, por cuanto es preciso la modelización y estudio del sistema completo. Sin embargo, la heterogeneidad puede contribuir igualmente a la magnitud y complejidad de la especificación ya que ha involucrado tradicionalmente el uso de diversos lenguajes, además de problemas semánticos no triviales.

Este trabajo cubre varias de esas carencias fundamentales. En primer lugar, provee una metodología de especificación de sistemas en SystemC denominada *HetSC*. Esta metodología ofrece un empleo sistemático y reglado de SystemC para asegurar que la especificación sea apta para la aplicación de otras actividades del diseño en y desde el nivel del sistema. De esta manera, la especificación del sistema se podrá reutilizar para realizar análisis de rendimiento, para la verificación a través de simulación y, en último término, para realizar una implementación HW/SW. Además, la metodología HetSC soporta heterogeneidad. Para ello, aporta reglas y facilidades de especificación SystemC adicionales para que la especificación se ciña a los supuestos de un Modelo de Computación (MoC) de entre varios soportados. De esta forma, se puede garantizar más fácilmente propiedades como el determinismo, lo que, de otro modo sería difícil, debido a la complejidad en tamaño y niveles de concurrencia y jerarquía de la especificación. La metodología *HetSC* soporta también la inclusión en la especificación de partes descritas bajo MoCs diferentes. Esto se hace dentro del lenguaje SystemC, por medio de canales y procesos frontera. *HetSC* aporta una librería metodológica denominada librería *HetSC*, que contiene facilidades de especificación que complementan las facilidades estándar de SystemC. Otra característica distintiva de *HetSC* es que el soporte de los distintos MoCs se realiza sobre el mismo núcleo de simulación de eventos discretos de SystemC. Todas estas características hacen de *HetSC* una contribución novedosa en el campo de especificación de nivel de sistema en SystemC.

En este trabajo se ha desarrollado también *SWGen*, una metodología de generación automática de software embebido desde código SystemC bajo la metodología *HetSC*, y por tanto, desde una especificación de nivel de sistema. Como se ha explicado, esto es crucial para que una metodología ESL provea la ganancia de productividad necesaria para paliar la brecha de diseño. La metodología *SWGen* se basa en la sustitución de las librerías SystemC y *HetSC*, por la librería *SWGen*. La idea es que la implementación SystemC de las primitivas de especificación, que hace que ésta sea ejecutable, se sustituye por una implementación software eficiente provista por la librería *SWGen* y que, entre otras cosas, no incluye ni el núcleo de simulación SystemC, ni el código de chequeo propio del nivel de sistema. *SWGen* es una contribución novedosa en el campo de generación de software desde el nivel de sistema por diversas características distintivas, tales como el tomar SystemC como lenguaje de entrada, estar basada en una librería metodológica y estar orientada a una plataforma HW/SW que incluye entre sus componentes básicos un sistema operativo embebido.

Finalmente, en este trabajo se han realizado una serie de experimentos que han permitido realizar una validación de las metodologías. En el nivel de especificación, se han realizado varios ejemplos, desde ejemplos sencillos que se incluyen en la librería *HetSC*, hasta un EFR Vocoder del estándar GSM. Con estos ejemplos se han mostrado los beneficios de la metodología de especificación en cuanto a simplicidad, reutilización de código y velocidad de simulación. La librería *SWGen* se ha implementado para las API-C (del RTOS *eCos*) y *POSIX* (válida para *GX-Linux*, una distribución de *Linux Embebido*, y para *eCos* también). Asimismo, se ha probado para diversas plataformas objetivo, basadas en procesadores de arquitectura *ARM* y *OpenRISC,* y en entornos de desarrollo diferentes. En todo caso, estas metodologías se integraron de forma natural y rápida en el entorno de desarrollo.

# Thesis Summary

Embedded Systems have a great importante in the electronic system market. These systems are often as ubiquous as transparent to the final user, since they are usually incrusted in bigger systems, where they perform a specific function. However, in some cases, i.e. mobile phones, they constitute a final product, with multiple I/O capabilities. Often, current embedded systems are hardware/software (HW/SW) systems. This enables a specific trade-off among crucial factors in embedded systems, such as cost, efficiency, speed, power consumption, etc.

A current issue on embedded system design is a problem known as *design gap*. This problem consists in that designer productivity grows at a lower rate than the quickly growing integration capabilities of electronic systems. The last update of the *International Roadmap for Semiconductors (ITRS'07)* showed for the first time that the design gap was due not only to an insufficient productivity in the hardware domain, but also to an insufficient productivity in the software domain, more relevant every day.

The ITRS itself addressed the solution: a qualitative evolution from the classic design methodologies for HW/SW systems, which have conform the specific discipline of HW/SW co-design, towards what is called Electronic System Level (ESL) design. An ESL design methodology focus the design on the system-level specification, which is taken as the starting point for the rest of the design activities. The system-level specification describes and documents the system in a more abstract level. Moreover, an ESL methodology must enable several design activities, such as verification and design space exploration (DSE) in such a high abstraction level, directly from the system-level specification. An ESL methodology must also enable an automatic implementation flow from the specification. A key point is the capture method or language employed in the generation of the system specification. A common language, with familiar syntax, understandeable and powerful enough for the different design domains is necessary. In this context, the SystemC standard language appears as the main candidate for system-level specification of embedded systems.

However, there are some deficiencies which prevent an effective and efficient application of SystemC in an ESL design flow of embedded systems. An important one, is the lack of a specification methodology which guides the user to perform a systematic usage of SystemC and which supports heterogeneity. Support of heterogeneity is essential in a complex embedded system specification methodology, since the modelization and study of the system as a whole is necessary. However, heterogeneity can also contribute to the size and complexity of the specification, since it has traditionally involved the usage of different languages, as well as non-straighforward semantical issues.

This works covers several of those fundamental deficiencies. Firstly, it provides a SystemC-based system specification methodology called *HetSC*. This methodology provides a systematic and ruled usage of SystemC in order to ensure that the specification is suitable for the application of other design activities performed in and from the system-level. In this way, the system specification can be reused for system-level performance analysis, for system-level verification by means of simulation and for HW/SW implementation activities from the system-level specification. The *HetSC* methodology supports heterogeneity. This means that it provides additional rules and SystemC facilities in order to get the specification suits the assumptions and rules of a Model of Computation (MoC) among those supported by *HetSC*. In this way, it is possible to obtain and guarantee certain properties, such as determinism, something otherwise difficult due to the complexity of the specification in terms of size, concurrency and hierarchy levels. *HetSC* gives the possibility to build specifications with parts under different MoCs. This is done by means of SystemC specification facilities too, specifically, border processes and channels. The methodological library *HetSC* contains the specification facilities which complement the standard facilities provided by the *SystemC* library. Another distinctive aspect of SystemC is that MoCs are supported over the discrete event simulation kernel of SystemC, without relying on specific solvers. All this features make *HetSC* a novel contribution in the field of system-level specification in SystemC.

In addition, the *SWGen* methodology has been developed. *SWGen* is a methodology for automatic generation of embedded software from SystemC code described under the HetSC methodology. Therefore, *SWGen* enables a direct software implementation flow from a system-level specification, which, as has been explained, is crucial in order to get an ESL methodology yielding a productivity gain able to reduce the design gap. The *SWGen* methodology is based on the substitution of the SystemC and *HetSC* libraries by the *SWGen* library. The idea is that the SystemC implementation of the specification primitives, which enables the specification to be executable, is substituted by an efficient software implementation provided by the *SWGen* library. *SWGen* implementation includes neither the SystemC simulation kernel, nor the system-level checking code. *SWGen* provides a novel contribution to the field of software generation from the system-level thanks to several distinguishing features, such as taking SystemC as front-end language, being based on a methodological library and being targeted to a HW/SW platform which includes an embedded operative system among its basic components.

Finally, in this work, a set of experiments have made possible a validation of the methodologies. In the specification level, several examples have been developed, from simple examples included in the *HetSC* library to an EFR Vocoder fulfilling the GSM standard. With these examples, benefits of the specification methodology, like code simplicity, simulation speed and code reuse possibilities, have been shown. The *SWGen* library has been ported for the *eCos* C-API and for a POSIX API, which was applied to *eCos* and *GX*-Linux, an Embedded Linux distribution. Moreover, *SWGen* has been ported to different target platforms based on ARM and OpenRISC architectures, and in different development environments. In any case, these methodologies were smoothly and quickly integrated in the development environment.

# Índice:

# Capítulo 1

# *Introducción*

*El crecimiento de la complejidad de los sistemas electrónicos e informáticos ha impulsado una mayor abstracción en las descripciones de los circuitos y de los programas. También una evolución en las herramientas de diseño que han de tratar con estas descripciones. Esto ha permitido un incremento en la productividad de los diseñadores de hardware y de los programadores. Sin embargo, esta productividad no es suficiente aún. La creciente complejidad de los sistemas embebidos requiere la sinergia de las técnicas de diseño de hardware y de software. Se precisan flujos de diseño que permitan una generación rápida y eficaz de los sistemas de altas prestaciones y gran flexibilidad funcional que requiere el mercado. La respuesta son los flujos de diseño de nivel de sistema o ESL. En estos flujos, la primera y más importante actividad es la especificación. En torno a ella se habilitan toda una serie de de actividades del diseño (verificación funcional, estima del rendimiento, etc) que antes se relegaban a etapas tardías del diseño. El Grupo de Ingeniería Microelectrónica (GIM) de la Universidad de Cantabria (UC), con una experiencia importante en el campo de los lenguajes y metodologías de diseño electrónico, desarrolla actualmente una parte importante de su labor investigadora en el diseño ESL y el empleo de nuevos lenguajes, como SystemC, en él. Este trabajo resume los principales resultados en dos líneas de investigación incluidos en ese campo: la especificación de sistemas embebidos heterogéneos en SystemC y la generación automática de software embebido a partir de esa especificación.*

## 1.1 Necesidades del Diseño de Sistemas Embebidos

Un sistema embebido es un dispositivo electrónico diseñado para computar una funcionalidad determinada en un sistema o producto mayor en el que se incrusta. Una característica básica de los sistemas embebidos es que están sujetos a estrictas restricciones de costo, consumo y, en general, de prestaciones. Tradicionalmente, el núcleo del cómputo de estos sistemas ha sido el microprocesador. Esto ha permitido que el costo de estos sistemas sea muy reducido. Por ejemplo, en **[RS08] [FA08]** se pueden encontrar microcontroladores de 32 bits con precios en torno a 10€ la unidad. Así, a pesar de su propósito específico, estos sistemas presentan una gran flexibilidad. En tanto que la arquitectura hardware permanece fija, el software empotrado en estos sistemas puede modificarse fácilmente. Esto permite nuevas funcionalidades del producto en un plazo breve (basta cargar una nueva versión del software), e incluso nuevos modelos de negocio (se ofrecen nuevas versiones como un servicio de soporte).


**Figura 1-1. Los sistemas embebidos son mucho más ubicuos que los ordenadores.**

Hoy en día, los sistemas embebidos están tan presentes y son tan importantes, como transparentes para la mayoría de la gente. A menudo pasan inadvertidos, dado que no interactúan directamente con el humano y muchos presentan un tamaño reducido. Sin embargo, se encuentran en teléfonos móviles, ascensores, coches, aviones, máquinas expendedoras, y pronto en nuestras ropas y el cuerpo, en un entorno de *inteligencia ambiental* **[AaRo03]**. Prueba de esta ubicuidad es que el 98% de los microprocesadores se emplean hoy día en sistemas embebidos, es decir**,** existen aproximadamente 3 sistemas embebidos por cada habitante del planeta, y que más del 90% del hardware computa en sistemas embebidos, lo que excede en mucho el que se dedica a ordenadores personales o servidores **[HKMB05]**.

El impacto económico de estos sistemas es consonante con su ubicuidad. El volumen de mercado de los sistemas embebidos es aproximadamente 100 veces superior al del computador de sobremesa. En Europa, el costo que involucran estos sistemas respecto al producto final se encuentra entre el 40% y el 60% dependiendo del sector (aviónica, automoción, industrial, medico, etc) **[HKMB05]**. Además, este impacto es creciente. Se prevé un volumen de 71 mil millones de euros para 2009 y una penetración de mercado de 40 mil millones de estos dispositivos en 2020 **[HKMB05]**. En **[KRIS05]** se predice un crecimiento del mercado superior al 16% para el software embebido, siendo ligeramente inferior (en torno al 14%) para los sistemas embebidos, superando los 82 mil millones de dólares de 2004 a 2009.

Hoy día, la alta capacidad de integración ha permitido que las prestaciones, a la vez que la complejidad, de los sistemas embebidos sea cada vez mayor. Esto ha

impulsado una evolución en su arquitectura. Los sistemas embebidos se convierten en sistemas HW/SW ya que es posible implementar como hardware específico una buena parte de la funcionalidad. Así, la *arquitectura*[1] o *arquitectura HW* de estos sistemas dispone e interconecta componentes de hardware específico junto con los componentes hardware necesarios para la ejecución del software embebido, tales como microprocesadores, periféricos, buses, etc. Esto provee una flexibilidad adicional en el diseño de estos sistemas, que permite ajustar el compromiso entre costo, prestaciones y flexibilidad. Por ejemplo, un reproductor de señal de video de satélite (DVB-S) puede implementar en hardware de forma óptima en consumo y velocidad la decodificación MPEG, en tanto que el resto de la funcionalidad del sistema (control de canales, configuración, actualización del *firmware*, etc) se realiza en software.

Un salto cualitativo determinado por la creciente capacidad de integración ha sido la concepción e implementación del sistema embebido como un único chip, es decir, un sistema en chip o SoC **[CCHM99]**, en lugar de como una placa impresa. Es decir, es posible integrar todos los componentes de la arquitectura en la misma pastilla de silicio. La capacidad de integración puede ser tal que varios de esos componentes sean núcleos procesadores, lo que posibilita el desarrollo de MPSoCs **[JeWo05]**. La capacidad de integración de muchos nodos de cómputo, traslada gran parte de la complejidad de esas arquitecturas a la red de interconexión, de modo que se proponen distintas arquitecturas de redes de interconexión en el chip (NoCs) **[CrJT03]**.

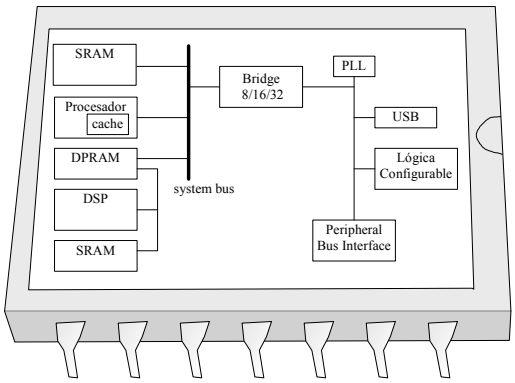



**Figura 1-2. Ejemplo de arquitectura típica de SoC (inspirada en la Fig. 3.3 de [CCHM99]).**

El crecimiento en la complejidad del hardware disponible en el sistema empotrado también afecta al software embebido. El SoC es menos accesible y se necesita asegurar la confiabilidad de ese software y proporcionar a las herramientas de desarrollo accesos para depuración. Asimismo, los MPSoCs permiten un paralelismo real que requiere un desarrollo software capaz de aprovechar esos recursos de ejecución.

En resumen, las plataformas de implementación de los sistemas embebidos son cada vez más complejas y más heterogéneas. Sin embargo, esta evolución no ha sido acompañada de una evolución igual de rápida en las metodologías de diseño de estos sistemas. El número de transistores disponibles crece más rápido que los que se pueden diseñar. Este es un fenómeno conocido como *brecha de diseño*. Desde sus ediciones

[1] El término *arquitectura* se reserva en este trabajo para la *arquitectura HW* del sistema, que se diferenciará posteriormente de la *arquitectura SW* del sistema.

más recientes **[ITRS01] [ITRS03]**, hasta las más recientes **[ITRS07]**, la hoja de ruta tecnológica internacional para los semiconductores, o *ITRS*, ha identificado el costo de diseño como la mayor amenaza para la continuación del itinerario propuesto.

Esta brecha de diseño se ha reflejado comúnmente en términos de hardware. Sin embargo, en **[ITRS07]**, se detalla por primera vez que esta brecha de diseño se produce en el diseño tanto del hardware como del software embebido. Aún más, los datos recalcan la importancia de la *brecha de diseño software*: la capacidad productiva del software embebido se duplica cada 5 años, en tanto que la demanda de líneas de código para aprovechar las capacidades tecnológicas se duplica cada 10 meses. Sin embargo, la inversión en tecnología de procesos domina todavía la inversión en tecnología de diseño. De este modo, en todas las versiones del ITRS hasta la actual se refleja como un reto primordial la mejora de la productividad del diseño en general y del diseño software en particular.

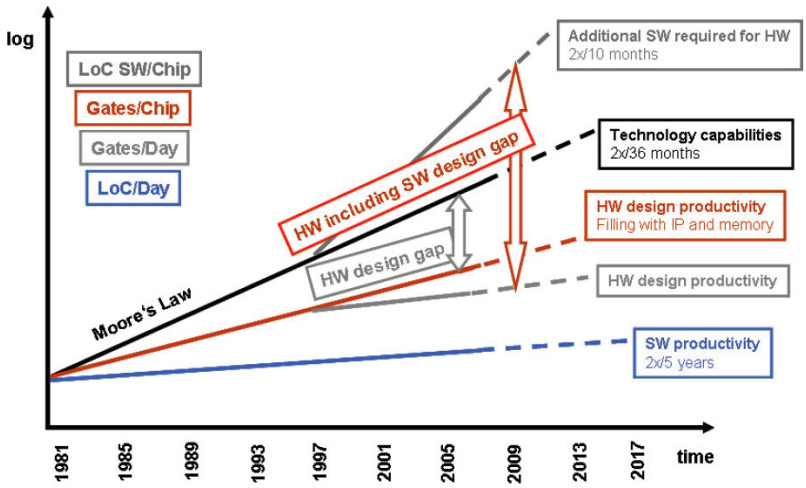



**Figura 1-3. Brecha de diseño HW/SW en el tiempo (tomado de [ITRS07].**

El diseño de sistemas embebidos se erigió como una disciplina distinguible del diseño electrónico o del desarrollo de software en tanto estos dos flujos, tradicionalmente separados, se comienzan a integrar. Así, se impulsó la propuesta de flujos de codiseño HW/SW. Se proponen flujos en "Y", en los que los flujos de diseño HW y SW arrancan por separado, para confluir posteriormente en una parte característica y compleja del codiseño HW/SW: la generación de interfaces HW/SW. Esta fase exige la puesta en común de los equipos de diseño HW y SW. Este modelo no es lo suficientemente productivo porque usualmente precisa secuenciar estos flujos, comenzando por el diseño de la plataforma HW antes. Ese diseño requiere un esfuerzo importante (exploración de arquitectura, descripción de la arquitectura propuesta en un lenguaje de descripción de hardware o HDL, etc) que retarda la labor del equipo software y aumenta, por tanto, el tiempo a mercado. Para paliar este problema, se proponen modelos de la plataforma hardware más abstractos, como los modelos de nivel de transacción **[Ghe05]**.

Los flujos en "Y invertida" elevan el nivel de abstracción en las etapas iniciales del diseño. En estos flujos, se introduce una etapa inicial y común consistente en la especificación del sistema. En algunos casos, esta especificación se reduce a un grupo

de documentos cuya semántica y requisitos son ambiguos. Esto dificulta la cooperación en el grupo de diseño, así como la compatibilidad de las implementaciones cuando se reutiliza la especificación. En algunos casos, se usa un modelo funcional, bien usando un lenguaje de programación de alto nivel (por ejemplo, C) o bien un lenguaje de modelado (Matlab, SDL, UML, etc), lo que contribuye a restar ambigüedades.

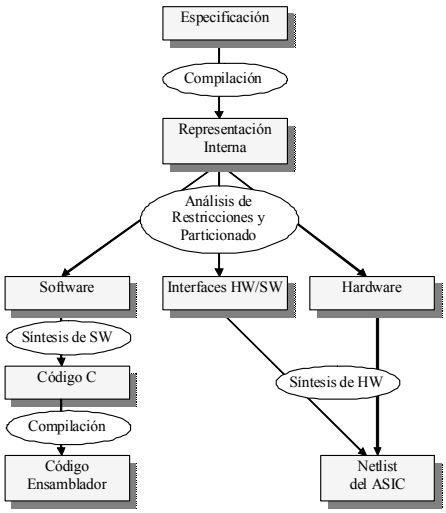



**Figura 1-4. Flujo de Codiseño HW/SW en "Y" invertida.**

En estos flujos las especificaciones se ciñen a ser un punto de arranque del proceso de diseño. Se requiere aún el desarrollo de las interfaces HW/SW y traslaciones manuales: la partición HW a lenguaje HDL y la partición SW a un lenguaje de programación. Además, el estudio del impacto de diversos aspectos acerca de la funcionalidad y el rendimiento, ambos vitales en el desarrollo de sistemas embebidos, se realiza tarde en las etapas de diseño. Así, los errores de especificación se detectan tarde, lo cual amplifica su costo y el tiempo a mercado.

Una contribución importante han sido los flujos de diseño basados en plataforma HW/SW **[San02]**. Es decir, se fija una gran parte de la arquitectura de la plataforma, así como un software mínimo, como controladores o, incluso, un sistema operativo, que se considerará también una parte íntegra de la plataforma. Esto posibilita un flujo iterativo previo a la implementación consistente en la especificación, exploración y refinado **[GaVN94]**. Esto hace más rápida y pausible la exploración del espacio de diseño (DSE) y el flujo de implementación HW/SW.

El ITRS, desde sus versiones iniciales **[ITRS01],** apuntó la necesidad de mejora de estos flujos hacia una metodología de *Diseño Electrónico de Nivel de Sistema* (ESL) **[KMNR00]**. En versiones posteriores, esta evolución se ha confirmado. En esta metodología, la especificación del sistema sigue siendo la entrada del proceso de diseño. Sin embargo, la especificación se convierte en una fase crucial del diseño. Como se muestra en la Figura 1-5, ahora, la mayoría de actividades de diseño se centran en torno a la especificación. El diseñador realiza esas actividades disponiendo de un marco integrado donde las herramientas de co-diseño, diseño lógico, layout, etc, cooperan en torno a la especificación.

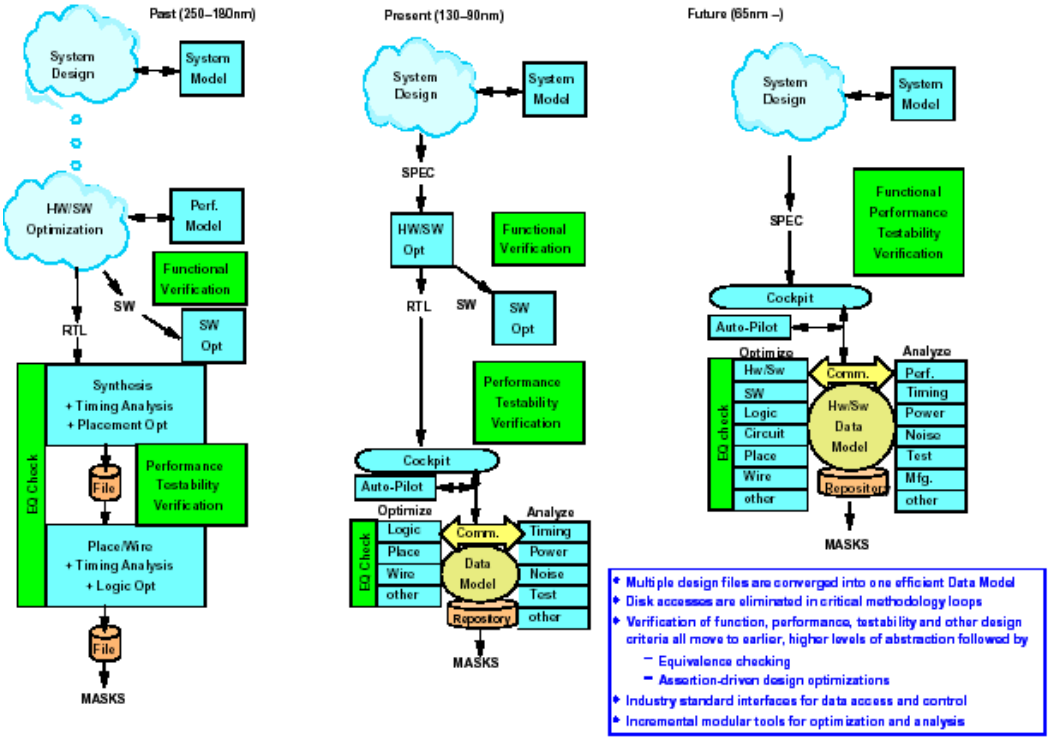



**Figura 1-5. Evolución de la metodología de diseño según el ITRS'05.**

Para que esta evolución del flujo de diseño predicha por el ITRS se haga realmente efectiva es preciso que la metodología propuesta cumpla una serie de requisitos. En primer lugar, debe estar basada en un lenguaje unificado para la especificación de sistemas HW/SW. La carencia de este lenguaje fue detectada como uno de los puntos más importantes en el entorpecimiento de la labor de los diseñadores de SoCs **[CCHM99]**. Los HDLs, tales como VHDL o Verilog, o los lenguajes de programación, tales como C, ADA, Java, etc no presentan por sí solos características suficientes para describir el sistema completo. Esto es debido a que las distintas funciones del sistema embebido deben ser ejecutadas en distintos recursos de la plataforma, tales como FPGAs, HW específico, procesadores de propósito general, etc, lo que requiere, a su vez, descripciones apropiadas para cada dominio de diseño. Es decir, la heterogeneidad en la plataforma lleva a la heterogeneidad en la especificación.

La heterogeneidad en la especificación se ha traducido usualmente en el empleo de varios lenguajes, ya que cada uno maneja los distintos aspectos de la especificación (sincronización, tipo de datos, etc) con un nivel de detalle determinado, y en la necesidad de cosimulación. En cambio, en una metodología ESL como la propuesta, la heterogeneidad de los sistemas embebidos requiere que un lenguaje de especificación unificado sea capaz de incorporar los distintos paradigmas de especificación desarrollados en los múltiples dominios de diseño. De otro modo, se dificultaría la aceptación del lenguaje por parte de aquellos grupos que usan lenguajes y metodologías adaptadas a su dominio de diseño. Asimismo, se debe proveer una metodología de uso de ese lenguaje apta para las actividades de diseño tales como verificación, exploración del espacio de diseño, etc. En cualquier caso, la especificación debe presentar una semántica de ejecución definida que evite ambigüedades en su interpretación.

La carencia de un lenguaje unificado ha afectado a una posible contribución sinérgica de distintos grupos de investigación para la construcción de una metodología de diseño en torno a la especificación. Asimismo, se precisan metodologías que solucionen de forma automática y eficiente las distintas actividades de diseño a partir de la especificación de nivel de sistema, tales como el análisis de rendimiento, exploración del espacio de diseño (DSE), decisión de la partición, verificación e implementación HW/SW. Respecto a la implementación, ya se ha destacado la necesidad de mejora de la productividad en la generación de software, que en el caso de una metodología ESL, debe arrancar desde la especificación de sistema. Por otro lado, se precisará incorporar diversos conceptos de los flujos de diseño previamente propuestos.

## 1.2 El Lenguaje SystemC

En este contexto el lenguaje SystemC **[Swan01] [GLMS02] [MuRR03] [BlDo04]** ha surgido como uno de los candidatos más firmes como lenguaje de especificación unificado para el diseño de nivel de sistema. Este lenguaje cuenta con decenas de miles de usuarios registrados, ha sido adoptado en el flujo de diseño de algunas empresas de dimensión internacional, y varias empresas de desarrollo de herramientas de diseño le han dado ya algún tipo de soporte.

El lenguaje SystemC extiende el lenguaje C++ proveyendo soporte para nuevos tipos (precisión finita, punto fijo y tipos *bit*), jerarquía modular (a través de nuevas clases, como el módulo y el puerto) y concurrencia (a través de distintos tipos de procesos). Todos estos elementos dan un soporte básico para el modelado de las partes SW, HW y sus interfaces. Además, el manual de referencia o LRM de SystemC se ha convertido en un estándar del IEEE **[IEEE05]**. El LRM estándar provee la estabilidad y certeza semántica que precisa el desarrollo de una metodología de diseño ESL.

SystemC presenta, además, una serie de ventajas adicionales:

- *Fácil adopción*. En un nivel sintáctico, SystemC provee básicamente elementos C/C++ para envolver funcionalidad C en procesos concurrentes. Ambos lenguajes, C y C++, son ya familiares para muchos desarrolladores de software embebido (según **[Edw98]** C y C++ se encuentran entre los tres lenguajes más usados en el desarrollo de software embebido). Tampoco son ajenos a muchos diseñadores de HW, que usan modelos de oro en C de los que derivan una descripción HW.
- La *Iniciativa Abierta de SystemC* (OSCI) aglutina empresas importantes del sector: de software, de diseño electrónico, tecnológicas, institutos de investigación y universidades. Existen diversos niveles de colaboración y contribución al lenguaje. La *OSCI* provee la librería SystemC, una librería de clases que soporta el lenguaje y contiene un kernel de simulación. Así, la compilación contra esta librería permite generar especificaciones ejecutables que cumplen estrictamente la semántica de simulación descrita en **[IEEE05]**. La OSCI también provee documentación básica como la *Guía de Usuario* **[SCUG02]** y la *Especificación Funcional* **[SCFS01]**, foros de discusión de usuarios, etc.
- *Ser de fuente abierta.* La librería SystemC así como otras librerías estándar OSCI se ponen a disposición de los usuarios en forma de código abierto.

- *Fácil extensibilidad.* Dado que SystemC es una librería C++, SystemC preserva todas las posibilidades de extensión del lenguaje C++ (herencia y polimorfismo). Además, el lenguaje añade una serie de facilidades tales como las clases canal (*sc_channel*) y canal primitivo (*sc_prim_channel*), métodos *request_update()*, *update()*, primitivas básicas de sincronización, como el evento y funciones *de* retrollamada, que permiten un control más preciso de las acciones a realizar antes, durante y después de cada fase de la ejecución SystemC (elaboración, simulación, etc). De esta forma, es posible crear nuevas facilidades de especificación con un gran control sobre su semántica de ejecución sin necesidad de modificar el kernel de simulación y, lo que es más importante, el LRM estándar.
- *Arquitectura del lenguaje escalable.* En relación al punto anterior, el lenguaje se articula de forma que se construye desde un núcleo estándar, compacto y eficiente. A partir de ahí, se pueden aportar al lenguaje más facilidades y características en forma de librerías. Estas pueden ser provistas bien por la OSCI o bien por terceros. Estas contribuciones pueden también ser estandarizadas.

De esta forma, SystemC presenta una disposición de capas característica tal y como se representa en la Figura 1-6.

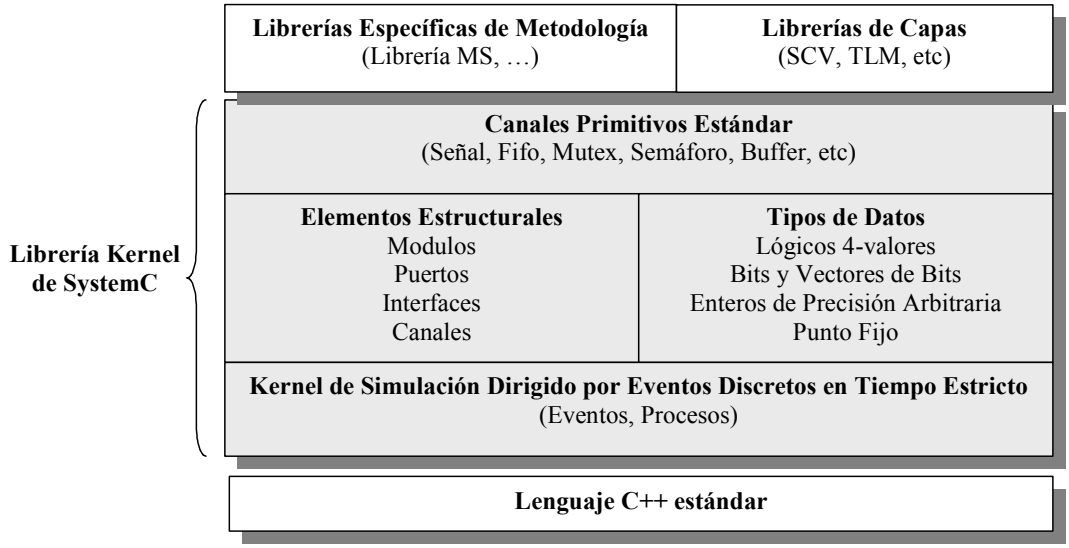



**Figura 1-6. Disposición en Capas y Elementos de la Librería kernel de SystemC.**

De este modo, en torno al lenguaje y la librería SystemC es posible construir toda una serie de metodologías. Éstas pueden extender el lenguaje y/o proveer herramientas que lo tomen como una entrada válida. De hecho, la OSCI provee una serie de librerías (Figura 1-7) enfocadas a distintas actividades de diseño. Entre estas librerías se cuentan la librería de verificación de SystemC o SCV **[SCV03]**, que provee una serie de facilidades para la mejora de los modelos de entorno y, por tanto, de la verificación del sistema por simulación. La librería de Modelado de Nivel de Transacción o TLM **[RSPF05]**, provee una serie de interfaces estándar que permiten la interoperabilidad y reutilización de componentes en modelos de plataforma realizados en el nivel de transacción. La librería de comunicaciones maestro/esclavo o MS **[MSCL02]** provee facilidades (como puertos maestro y puertos esclavo) que permiten la especificación

bajo un modelo de computación que permite encadenar la ejecución de funcionalidades en distintos módulos mediante llamadas a procedimiento remoto (RPC).

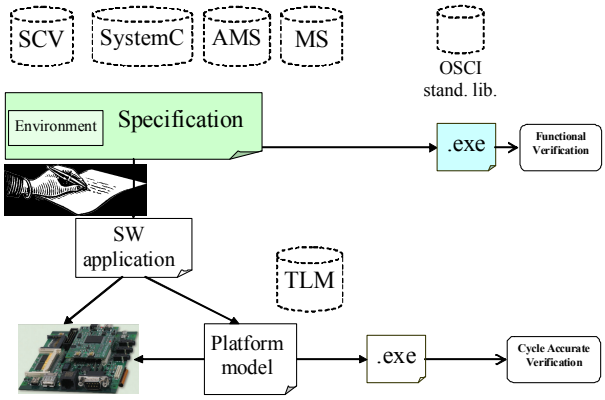



**Figura 1-7. La OSCI provee varias librerías metodológicas.**

## 1.3 Metodologías y Lenguajes de Diseño en el GIM

El Grupo de Ingeniería Microelectrónica (GIM) **[GIM08]** de la Universidad de Cantabria (UC) **[UC08]**, en el que se enmarca este trabajo de tesis, ha estudiado la evolución del estado del arte del diseño electrónico y del codiseño HW/SW **[Rodr00]**. También ha contribuido a dicha evolución mediante una labor investigadora en el empleo de lenguajes de descripción de hardware tales como VHDL **[ViSa93]**, cuando éstos empezaban a tomar un papel importante en el diseño electrónico.

Durante la década de los 90, el grupo ha realizado aportaciones en el ámbito de la tecnología de diseño electrónico en el tema de síntesis de alto nivel **[Sanc91] [Tabu95] [Fern98]**. Ese nivel, permite al diseñador hardware especificar circuitos mediante un código de nivel algorítmico. En esa misma década, el GIM inició también una actividad en el campo del codiseño hardware/software, que le ha dotado de experiencia en los campos de especificación de nivel de sistema y de desarrollo de software embebido. Inicialmente, el GIM provee soluciones en la adaptación del flujo de diseño HW, incluyendo la síntesis de alto nivel, para su integración en una metodología de co-diseño HW/SW. En **[Fern98]**, se propone una fase de exploración del espacio de diseño y se implementan algoritmos de síntesis adecuados a esa exploración.

Posteriormente, se propone y usa el lenguaje ADA **[ADA97]** como lenguaje de especificación. Una de las ventajas es que este lenguaje es de alto nivel, soportando orientación a objetos y concurrencia. Además, guarda una gran similitud sintáctica con VHDL. Este hecho facilitaría la introducción del lenguaje ADA o una variación de este lenguaje (a modo de sub o superconjunto) como lenguaje de especificación unificado en la comunidad de diseño hardware. Así, en **[Lope98]**, como un paso previo, se mejora la conexión del flujo de diseño SW y HW, habilitando la cosimulación de la parte SW del sistema, descrita en ADA**,** con la partición HW, descrita en VHDL. En **[Her00]**, uno de los últimos resultados del grupo en esta línea, se especifica en ADA un sistema AAL/ATM. Sin embargo, uno de los problemas principales que presenta ADA es que su compilador es complejo y de difícil portabilidad a las numerosas plataformas de implementación existentes. Esto dificulta la aplicación de ADA a un flujo de

implementación de software. También dificulta la extensión del lenguaje, y por tanto, el desarrollo de las herramientas ESL que tomen ADA como entrada.

En este contexto, a finales de los 90, aparece SystemC en sus primeras versiones como un lenguaje más flexible y extensible. El GIM empezó a centrar gran parte de sus contribuciones al diseño de sistemas embebidos HW/SW en torno a este lenguaje. En **[FHSV02]** se realiza una primera contribución para el desarrollo de una metodología de diseño de sistemas embebidos asequible a la pequeña y mediana empresa. Posteriormente, el GIM adquirió experiencia en la aplicación del lenguaje SystemC en un flujo de desarrollo HW. Se desarrolló un núcleo procesador *OpenRISC* y una plataforma HW, cuya característica fundamental era su alto nivel de configurabilidad **[BCHP03] [BPCH04]**. Además se desarrollo un paquete de desarrollo software para esa plataforma (ensamblador, compilador, etc), donde destacaba el soporte de *eCos* **[Mas03],** un sistema operativo configurable. En definitiva, se desarrolló una plataforma HW/SW altamente configurable. Otras aportaciones importantes de ese trabajo se realizaron en un nivel metodológico. Por ejemplo, se demostró la posibilidad de reutilización en un nivel industrial de código fuente abierto tanto del software de desarrollo como de la descripción hardware de la plataforma. Algunas aportaciones estuvieron muy relacionadas con el lenguaje SystemC. Se aplicaron técnicas avanzadas de verificación, reutilizando bancos de pruebas funcionales, aplicando técnicas de pseudo-aleatorización constreñida de las entradas y usando transactores, lo que requería el uso de la librería SCV.

Actualmente, el GIM sigue realizando contribuciones que se articulan en torno a una metodología ESL de diseño de sistemas embebidos HW/SW y que se basa en el lenguaje SystemC. La metodología de diseño se resume en la Figura 1-8.

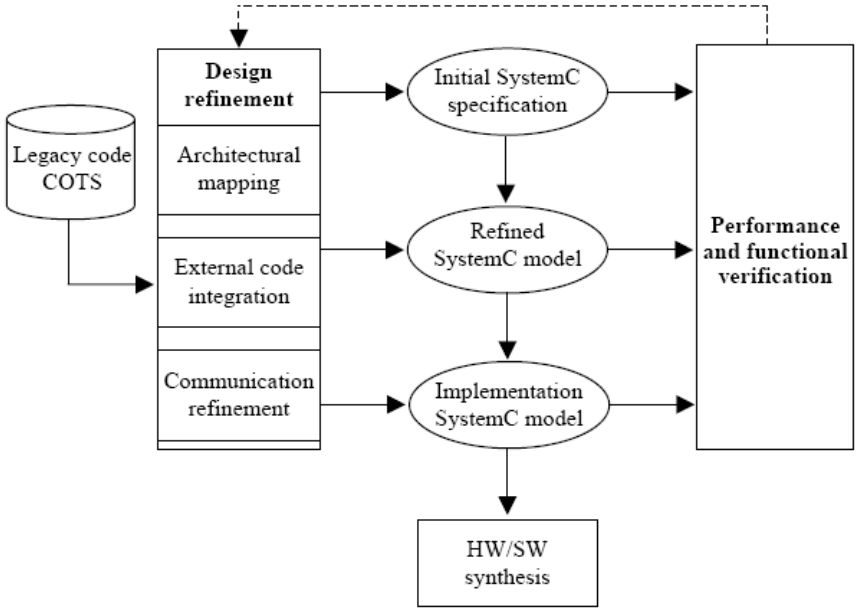



**Figura 1-8. Metodología de diseño de Fuente Única del GIM/TEISA/UC.**

Se trata de un marco de diseño ESL de fuente única. La metodología gira en torno a una especificación de sistema que se realiza usando el lenguaje SystemC. La especificación admite código desarrollado por el propio diseñador y la integración de

otros bloques de propiedad intelectual, que pueden requerir el desarrollo de los correspondientes bloques adaptadores o *envoltorio*. Permite la realización de una verificación de nivel de sistema, basada en simulación, que puede utilizar los servicios de la librería SCV u otras extensiones de la librería SystemC, como en **[HeVB06]**. El cumplimiento de los requerimientos de rendimiento determina la partición HW/SW. Esto se estudia en una etapa de exploración del espacio de diseño, que se realiza en el nivel de sistema. Es decir, se emplea la especificación, junto con la información de mapeo a la plataforma HW/SW (partición HW/SW) para estimar rápidamente el rendimiento y cotejarlo con los requerimientos iniciales. Para esta estimación el GIM desarrolló la librería *Perfidy* **[HPSV04]**, que estimaba el rendimiento temporal desde la especificación en SystemC al menos un orden de magnitud más rápido que un simulador de instrucciones (ISS). Posteriormente, la librería *PERFidiX* **[PASV06] [PAVE06]**, usando un motor de simulación SystemC, conseguía realizar estimaciones de mayor precisión, cuando el código SystemC se ha refinado en un código C o C++ con llamadas POSIX **[POSIX04]** a un OS embebido. En posteriores trabajos, se han desarrollado técnicas que permiten la estimación del consumo del software **[CPVM07]**.

La metodología admite una fase de refinamiento de la especificación, sin dejar de usar SystemC, para la adecuación de sus partes a uno o más dominios de diseño (analógico, digital, software, etc). El refinamiento es actualmente manual. Tanto la especificación inicial como el producto del refinamiento pueden ser heterogéneos, es decir, con partes o subsistemas con primitivas, construcciones, restricciones y semántica propias de distintos modelos de computación (MoC). Terminada la fase de refinamiento, tiene lugar el flujo de implementación o síntesis HW/SW. Este conlleva la síntesis del hardware, la generación de software y la generación de las interfaces HW/SW, que se incrusta en las dos primeras.

## 1.4 Líneas de Investigación y Objetivos Generales de la Tesis

Dentro del marco de diseño de fuente única presentado en el apartado anterior, el presente trabajo de tesis ha desarrollado dos líneas de investigación, resaltadas en la Figura 1-9. Consisten en el desarrollo de:

- Una **Metodología de Especificación Heterogénea de Nivel de Sistema en SystemC** que permite escribir una especificación bajo distintos modelos de computación y sirve como entrada a la metodología de diseño de fuente única.
- Una **Metodología de Generación Automática de Software** capaz de generar, desde la especificación en SystemC, sin necesidad de refinamiento manual, el software embebido, incluyendo el manejo de controladores de entrada/salida y la comunicación con el hardware específico de la plataforma. La metodología considera los elementos actualmente disponibles en la plataforma HW/SW.

Estas dos líneas de trabajo cubren dos requerimientos esenciales para la mejora de la productividad del diseño señalada en el ITRS: una metodología de especificación unificada que abarque múltiples dominios de diseño como núcleo de la metodología de diseño y una metodología eficaz y más productiva de generación de software embebido.

El resultado de este trabajo ha sido la metodología de especificación *HetSC* **[HSC08]** y la de generación de software embebido *SWGen* **[SWG08]**. Estas

metodologías llevan asociadas sendas librerías, denominadas librería *HetSC* y librería *SWGen* respectivamente. La librería *HetSC* extiende y mejora el soporte de la librería SystemC para especificación heterogénea. La librería *SWGen* permite la generación automática de software embebido desde el código de especificación SystemC. Estas metodologías y sus librerías serán explicadas en los capítulos siguientes.

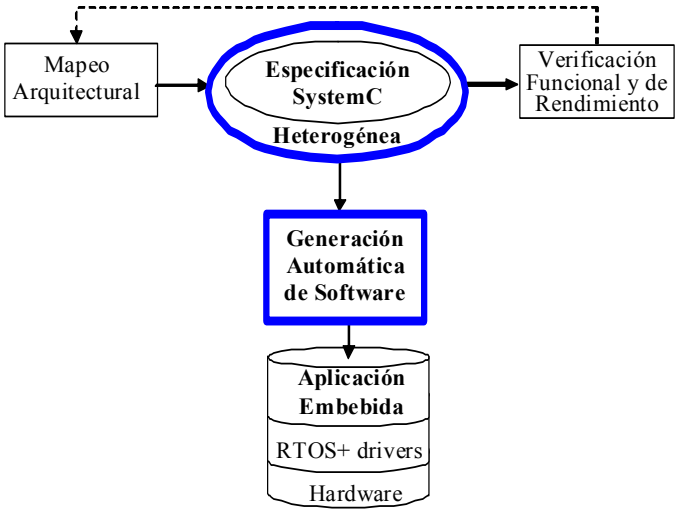



**Figura 1-9. Resalte de las líneas de investigación de esta tesis.**

Las metodologías *HetSC* y *SWGen* se integran en el flujo metodológico de fuente única propuesto por el GIM. En la Figura 1-10 se presenta el marco integrado basado en librerías metodológicas que soporta el flujo de diseño ESL propuesto. Además de las librerías *HetSC* y *SWGen*, otras librerías del GIM (*Perfidy, PERFidiX*) y OSCI (*MS, SystemC-AMS, TLM*) soportan o complementan las actividades de diseño realizadas en torno a la especificación del sistema.

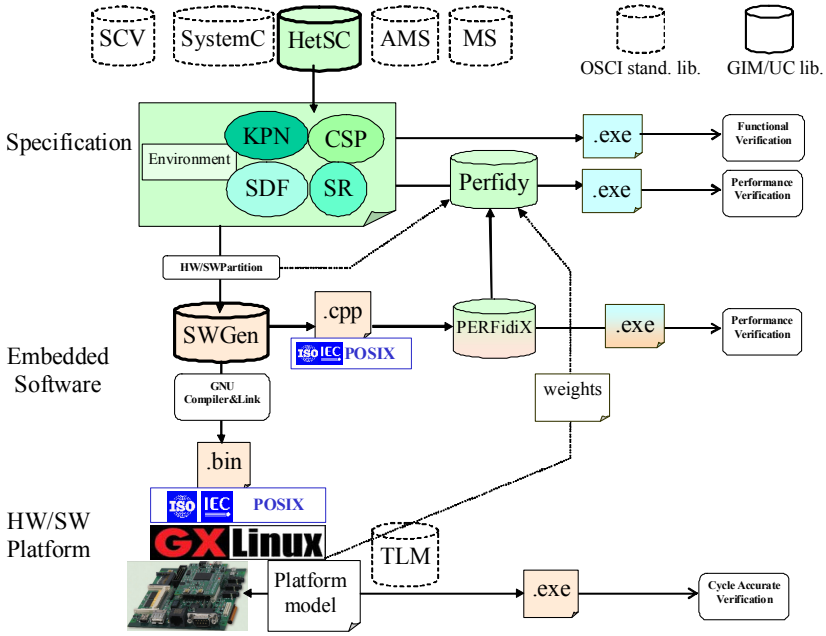



**Figura 1-10. Librerías *HetSC* y *SWGen* integradas con otras librerías en un flujo de diseño ESL.**

# Capítulo 2

# Metodología de Especificación Heterogénea *HetSC*

*En este capítulo se presenta la metodología HetSC, una metodología para la especificación de sistemas embebidos heterogéneos basada en el lenguaje SystemC. Esta metodología establece cómo usar el lenguaje SystemC para describir el sistema embebido y otorga a las construcciones descritas una semántica de ejecución abstracta, coherente con la semántica de simulación establecida por el LRM de SystemC. La simplicidad y ortogonalidad en la estructura de estas construcciones permitirán en el siguiente capítulo asociarles también una semántica de implementación (software). Es decir, la metodología HetSC se orienta al diseño. Esto significa que una especificación HetSC no es únicamente un modelo, sino también el punto de arranque de la verificación, análisis, e implementación del sistema. HetSC es además una metodología de especificación heterogénea ya que soporta que una especificación presente partes descritas bajo Modelos de Computación (MoCs) diferentes. Esta es una capacidad necesaria para que una metodología de especificación permita la descripción completa de un sistema embebido actual, cada vez más complejo y heterogéneo. La metodología HetSC provee una serie de guías y reglas de especificación y una serie de facilidades de especificación que complementan a las provistas por la librería de SystemC. Estas facilidades se incluyen en una librería, denominada también HetSC. Gracias a las guías y reglas de especificación, el diseñador sabe cómo usar y como no usar las facilidades provistas por las librerías SystemC y HetSC. Además, HetSC puede cooperar con otras librerías de especificación, como SystemC-AMS y TLM, y de implementación, tales como SWGen, explicada en el capítulo siguiente.*

## 2.1 Introducción

Como se ha visto el ITRS lleva señalando desde 2001 la necesidad de un cambio en las metodologías de diseño y de especificación. En este capítulo se aborda la metodología de especificación HetSC. Una metodología de especificación de sistemas embebidos HW/SW basada en el lenguaje SystemC.

En este contexto, el término *especificación* incluye al de *modelado*. El término de especificación añade al de modelado el matiz de "*modelo para diseño*". Comúnmente, el modelado está asociado al objetivo de conseguir capturar la funcionalidad para propósitos de análisis y verificación. La especificación se debe entender aquí como un modelo que permite abordar todas las actividades de diseño, no sólo las de simulación, sino también aquellas relacionadas con el flujo de implementación. Es decir, la especificación tiene una semántica de implementación, además de una semántica de ejecución. Por tanto, una primera motivación de esta línea de trabajo es la de proveer una metodología de especificación que sirva para modelar y diseñar desde un alto nivel de abstracción sistemas embebidos.

Otro factor que hay que tener en cuenta en la evolución de los sistemas embebidos es su tendencia hacia una concurrencia masiva. Estos sistemas no solamente están compuestos de un número creciente de componentes complejos, sino que muchos de estos son elementos concurrentes entre sí. Desde el nivel de especificación de sistema, esta concurrencia se puede estudiar independientemente de la implementación final que se de a cada componente. Cada componente puede ser implementado mediante bloques hardware (naturalmente concurrentes) o bien como hilos o procesos en software (corriendo en uno o varios procesadores). La necesidad de que una metodología de especificación soporte concurrencia se debe por un lado a la necesidad de mejora del rendimiento temporal. Por otro lado, es también cierto que hay algoritmos que son especificados de forma más natural como procesos concurrentes. Sin embargo, la programación concurrente presenta nuevos problemas y retos en el desarrollo de SW, especialmente en los sistemas embebidos, donde en muchas aplicaciones, la eficiencia, el determinismo, la protección frente al interbloqueo, etc son críticos. En **[Lee06]**, se mantiene que el desarrollo de software embebido basado en el desarrollo de un creciente número de hilos dejará de ser válido como metodología de ingeniería software. Un sistema debería ser concebido como una arquitectura de agentes cooperantes fácil de entender. Las características de estos agentes y de la interacción entre ellos deberían entenderse bajo un MoC determinado. Esta idea es extensible al sistema completo.

Los MoCs han sido propuestos y usados durante largo tiempo para el control de la concurrencia. Uno de los más extendidos es el MoC de Eventos Discretos (o MoC DE), capaz de simular eficientemente millones de puertas. Los diseños hardware de nivel de transferencia de registros (RTL), están basados en el MoC de Tiempo Discreto (DT). El éxito de este modelo de computación en el diseño de hardware motivó su extensión para el diseño de software fiable con el MoC Síncrono Reactivo (SR) **[BeBe91] [BCEH03] [Berry00]**. Otros modelos clásicos propuestos para el diseño de SW embebido son, los modelos de flujos de datos **[LePa95]**, y algunos casos particulares como el MoC de Flujo de Datos Síncrono (SDF) **[LeMe87]** y el MoC de Redes de Procesos de Kahn (KPN) **[Kahn74]**, el MoC de Procesos Secuenciales Concurrentes (CSP) **[Hoa78]**, las

redes de Petri (Petri-Net) **[Petr62]** y Redes de Petri de Libre Elección (FCPN) **[DeEs95][SLWS99],** etc.

Se pueden encontrar más modelos de computación en la bibliografía, entre ellos variantes y precedentes de los mencionados anteriormente. Muchos de estos modelos están relacionados y comparten supuestos básicos, en tanto que difieren en detalles. Cada modelo de computación establece unas reglas que aportan al diseñador una serie de propiedades y ventajas. Dado que cada campo de aplicación usualmente tiene una combinación distinta de requerimientos, es natural el hecho de que existan MoCs diferentes, cada cual adaptado mejor al diseño en un campo de aplicación determinado. Según **[Lee06]**, una de las ventajas más importantes que provee el uso de MoCs apropiados es el desarrollo del sistema desde especificaciones comprensibles, deterministas y que se puedan construir mediante mecanismos de composición. El no determinismo se introduciría sólo cuando es necesario.

Asimismo, es interesante el soporte de varios MoCs en una metodología de especificación. A este tipo de heterogeneidad se la denomina aquí *heterogeneidad de especificación* o *de nivel de sistema*. En **[Jan04]** se consideran las siguientes ventajas en la adopción de MoCs en una metodología de especificación:

***Desarrollo de la Especificación más rápido.*** Un MoC concreto puede proveer primitivas y reglas de especificación más adecuadas para la descripción de la solución a un determinado problema. Por ejemplo, un algoritmo de control puede ser más fácilmente implementable bajo un MoC CSP, en tanto que un algoritmo intenso en datos puede ser más fácilmente describible usando el MoC SDF.

***Velocidad de Simulación Óptima.*** El incremento en la complejidad del sistema conlleva tiempos de simulación crecientes y, en ocasiones, inaceptables. La selección de un MoC apropiado optimiza los tiempos de simulación porque los recursos del simulador solo tratan con la información importante.

***Propiedades Útiles.*** La especificación bajo las reglas de un MoC particular garantiza el cumplimiento de una serie de propiedades, entre las que puede estar el determinismo, protección frente a interbloqueo, etc que son ventajosas, e incluso necesarias, en muchas aplicaciones y modelos de sistema.

***Flujo de Implementación viable y eficiente***. El soporte de heterogeneidad facilita el flujo de implementación. Como se explico en al capítulo anterior, la creciente complejidad de integración permite la evolución de las plataformas y sus arquitecturas. Para que estas presenten un nivel de flexibilidad y eficiencia adecuado a esa creciente capacidad de integración se exige un mayor grado de *heterogeneidad de nivel de plataforma*. La plataforma deja de ser un procesador más un hardware específico. Ahora pueden aparecer varios tipos de elementos procesadores de SW (microprocesadores para embebidos, DSPs, coprocesadores, etc) y elementos HW de distinta naturaleza (FPGAs, FPGAs dinámicamente reconfigurables, ASICs, HW analógico parametrizable, etc) **[Resa05]**. De este modo, cada MoC puede proveer construcciones y facilidades cuya sintaxis y semántica de implementación haga que una misma funcionalidad sea más fácilmente identificable y manejable por las herramientas de síntesis. Por ejemplo, la *señal hardware* es una forma natural de describir comunicación entre bloques HW. Sin embargo, la implementación de la señal hardware en software

requiere un esfuerzo de interpretación que no es trivial. Un ejemplo complementario sería una comunicación a través de fifos, que puede encontrar una fácil interpretación en software, por ejemplo, como colas de mensajes, en tanto que, en HW, requeriría la definición de señales de protocolo, definir como se implementa la memoria interna, etc.

En **[RHV07]** , se distinguieron los dos tipos de heterogeneidad que debe soportar una metodología de especificación: *heterogeneidad horizontal* y *vertical*. (Figura 2-1).

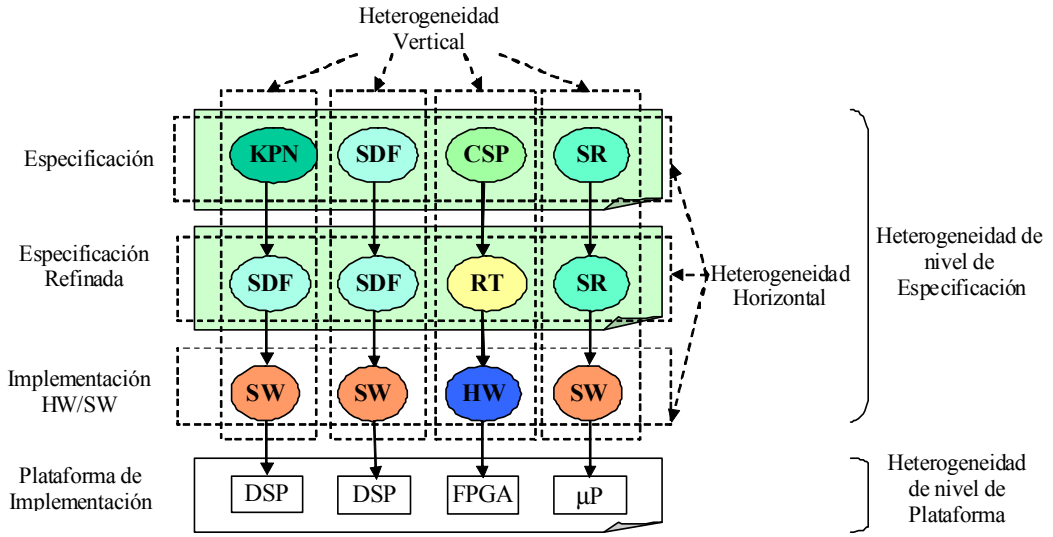



**Figura 2-1. Heterogeneidad Horizontal y Vertical.**

*Heterogeneidad Horizontal* se refiere a la capacidad de la metodología de especificación para permitir que ésta:

- presente varias partes donde cada una esté bajo un MoC específico.
- permita una integración suave, sin costuras, de esas partes que le de una semántica de ejecución (o simulación) coherente al conjunto.

*Heterogeneidad Vertical* se denomina a la capacidad de la metodología para soportar la evolución del mismo componente de la especificación a través de varios MoCs. Como se ha comentado, la especificación del sistema se está convirtiendo en el foco de la metodología de diseño completa. Las diferentes herramientas operan en cada componente y lo trasforman en un objeto que debe poder volverse a integrar en la especificación y dar un resultado coherente y simulable. El refinado sucesivo desde el modelo inicial hasta la implementación requiere la capacidad de tratar la transformación de MoCs. La metodología de especificación tiene que ser capaz de soportar estas transformaciones, que también implican heterogeneidad dado que el mismo componente tiene que evolucionar a través de diferentes MoCs.

La heterogeneidad es útil independientemente de la metodología de diseño. En la Figura 2-1, se aplica a un flujo de codiseño HW/SW en "Y" invertida. La especificación original es refinada hasta un punto adecuado en el que el modelo admite una partición HW/SW y es posible la asignación de cada parte refinada a los recursos SW o HW mediante un flujo de implementación eficiente. Por ejemplo, las partes refinadas a un MoC SDF admiten la aplicación de flujos de implementación SW eficientes. Las partes

refinadas a un MoC CS (síncrono de reloj), se puede corresponder con un estilo de descripción hardware bien de comportamiento o bien de transferencia de registros (RTL), del que varias herramientas de síntesis son capaces de generar un fichero de configuración para una FPGA o realizar una síntesis para una implementación a medida. En la Figura 2-2 se representa la heterogeneidad horizontal en un flujo de diseño en "Y". En estos flujos, se maneja un modelo funcional (modelo independiente de plataforma o PIM) y un modelo de la arquitectura (modelo dependiente de plataforma o PDM). La heterogeneidad de nivel de especificación y la de nivel de plataforma impulsan la heterogeneidad tanto en los modelos PIM como en los PDM.

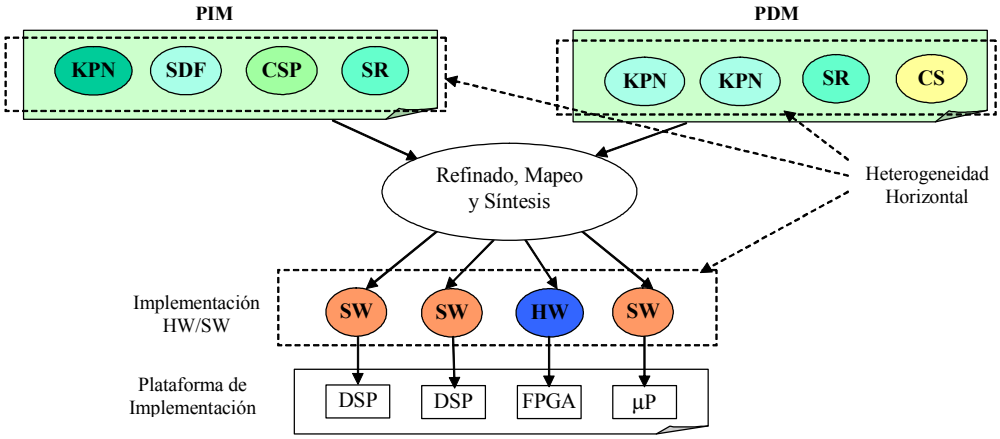



**Figura 2-2. Heterogeneidad Horizontal en un flujo de diseño en "Y".**

Hay dos alternativas fundamentales para introducir heterogeneidad en una metodología de especificación. Por un lado, es posible usar varios lenguajes. En este caso, la especificación de cada componente obtiene todas las ventajas que se derivan del uso del lenguaje más apropiado. Sin embargo, la integración de varios lenguajes es difícil y requiere solucionar la conexión entre ellos. Por otro lado, es posible usar un solo lenguaje. En ese caso, se exige al lenguaje muchas más características y la elección puede dar lugar a que dicho lenguaje no sea el más apropiado para especificar todos los componentes del sistema. Sin embargo, la integración es más simple.

Como se ha visto, SystemC es una propuesta de gran aceptación como lenguaje unificado de especificación de nivel de sistema. Tiene características que permiten el soporte de varios MoCs. Además, el núcleo del lenguaje provee algunas facilidades de extensión (como canales primitivos, eventos, retrollamadas en las distintas fases de simulación, etc), junto con las capacidades de abstracción de C++.

Este trabajo propone una metodología de especificación basada en el lenguaje SystemC. Esta metodología de especificación soporta heterogeneidad horizontal, ya que permite la especificación de partes bajo diferentes MoCs y la conexión entre ellas. También soporta heterogeneidad vertical, ya que provee el marco en el que se pueden aplicar las transformaciones de cada parte de la especificación. No obstante, las transformaciones del diseño están fuera del objetivo de este trabajo. La metodología de especificación es independiente y, por tanto aplicable, a diferentes flujos de diseño y distintos tipos de modelos. Asimismo, el desarrollo de las reglas y facilidades de

especificación se basa en trabajos que cuentan con un desarrollo formal. En cualquier caso, este trabajo se centra en el desarrollo de la metodología de especificación, más que en su completa formalización, que es objeto de trabajos posteriores.

Esta metodología va más allá del estándar TLM **[Ghe05]**, la propuesta más reciente para modelado de nivel de sistema en SystemC. Aunque la librería TLM ha elevado el nivel de abstracción en SystemC, no constituye una metodología de especificación heterogénea y carece de la semántica requerida por cada MoC y su integración. En cambio, la metodología que se presenta en este trabajo, establece una serie de reglas, una sintaxis, con una correspondiente representación gráfica, y una semántica para la construcción de una metodología de especificación heterogénea en SystemC. Además, la metodología provee las facilidades adicionales para cubrir las carencias sintácticas y semánticas de SystemC a este respecto. La metodología pretende el soporte de cualquier MoC, dado que este pueda simularse sobre el MoC DE de tiempo estricto. Este MoC está definido por la semántica de SystemC en el manual de referencia **[IEEE05]** y encuentra implementación en el núcleo de simulación incrustado en la distribución de la librería provista por la OSCI y en otros simuladores comerciales.

## *2.2 Estado del Arte*

Se pueden distinguir tres tipos de trabajos relacionados con la especificación heterogénea:

- Los que han descrito y sentado las bases formales de un modelo de computación.
- Los que han propuesto metamodelos para el entendimiento, análisis, comparación y clasificación de MoCs
- Los que han propuesto marcos y metodologías de especificación capaces de integrar diferentes MoCs.

En las tres subsecciones siguientes se repasarán estos trabajos.

### 2.2.1 Modelos de Computación

En esta sección se hace un breve repaso de los modelos de computación existentes más importantes. Un factor común en la mayoría de esos MoCs es que tratan de responder a problemas típicos de especificación concurrente, tales como el indeterminismo, el interbloqueo parcial o total, la inanición, la continuidad, etc. Asimismo, también tratan de proveer los elementos necesarios para dotar a la metodología de especificación de la capacidad expresiva suficiente. Los MoCs resuelven, por tanto, problemas prácticos y son comúnmente empleados, aunque en muchos casos de forma inconsciente por ingenieros de sistema, programadores, diseñadores de hardware, etc.

El carácter práctico de los MoCs no se entiende sin su base formal. Uno de los trabajos más importantes es el de Kahn **[Kahn74]**. Una red de Kahn puede verse como un conjunto de máquinas de Turing conectadas por cintas unidireccionales infinitas. Una de las mayores contribuciones es la eliminación de la noción de estado del sistema. Con esa visión, Kahn es capaz de proponer una estructura de sistema que, junto con un conjunto de restricciones y una semántica de comunicación entre los nodos de

computación basada en colas FIFO infinitas con accesos de lectura bloqueantes, le permiten demostrar formalmente una serie de propiedades del sistema, entre las que destaca el determinismo. El determinismo es una condición a menudo requerida y útil en los sistemas embebidos. En general, consiste en que una "*misma entrada*" al sistema produce siempre una "*misma salida*". Específicamente, el determinismo en una red de procesos de Kahn (KPN), consiste en que la historia de las unidades de datos transferidas en los canales de comunicación no depende del orden de ejecución de los procesos **[Park95]**.

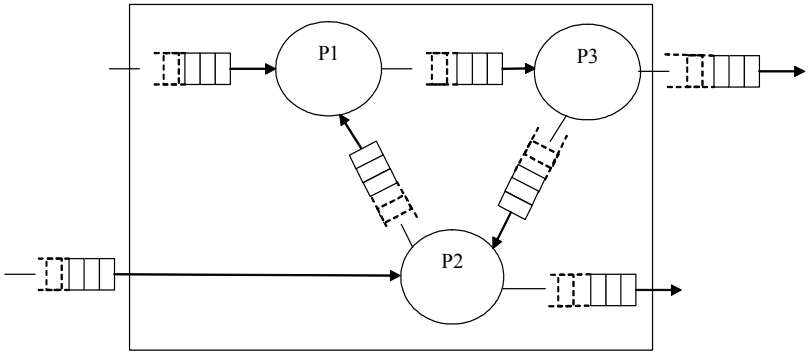



**Figura 2-3. Red de Procesos de Kahn o KPN.**

Un modelo KPN puede presentar situaciones de interbloqueo. Estas vienen dadas por una dependencia circular entre dos o más procesos, donde cada uno se encuentra en un estado en el que precisa leer un dato que, a su vez, sólo puede ser proporcionado por los otros. Por ejemplo, en la Figura 2-4 se presenta una red de Kahn formada por dos procesos y dos comunicaciones, ch1 y ch2, del tipo FIFO bloqueante referido anteriormente. Asumiendo que en el arranque las fifos están vacías, los procesos P1 y P2 se hayan bloqueados en un acceso de lectura. Es una situación de interbloqueo, en la que P2 espera a que P1, el único proceso que puede escribir datos en ch1, lo haga. Sin embargo, P1 a su vez está esperando a que P2 escriba datos en el canal ch2.

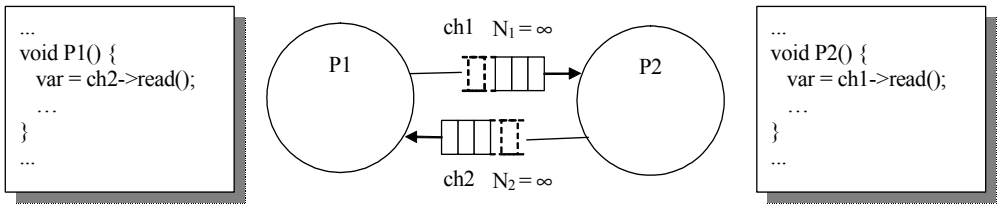



**Figura 2-4. Red de Kahn con interbloqueo por condiciones de lectura.**

Sin embargo, el MoC KPN elimina un número potencialmente alto de condiciones de interbloqueo, frente un modelo de estructura similar basado en colas finitas, ya que todas las situaciones de bloqueo en escritura, debidas a una situación de cola llena, se eliminan. Por eso, las redes de Kahn se encuadran en **[Jan04]** en un tipo de redes que denomina redes *débilmente acopladas*. Este nombre refleja la menor dependencia entre los procesos para la continuación de su ejecución.

Por ejemplo, si se codifica el cómputo de los procesos P1 y P2 como en la Figura 2-5, dependiendo del valor de N se puede tener una situación de interbloqueo. Asumamos que $N_1 = N_2 = N$. Un análisis del sistema de la Figura 2-5 permite deducir que entrará en una situación de interbloqueo si M>2N+1, por ejemplo, si el tamaño de

las fifos es N=2 y M=6. En esta especificación del sistema ese interbloqueo es debido a que ambos procesos llegan a llenar las fifos ch1 y ch2 y ambos se bloquean en escritura, a la espera de que el otro proceso haga al menos una lectura. A este tipo de interbloqueo se lo denomina *interbloqueo artificial* **[Park95]** o *sobresincronización* **[Jan04]**.

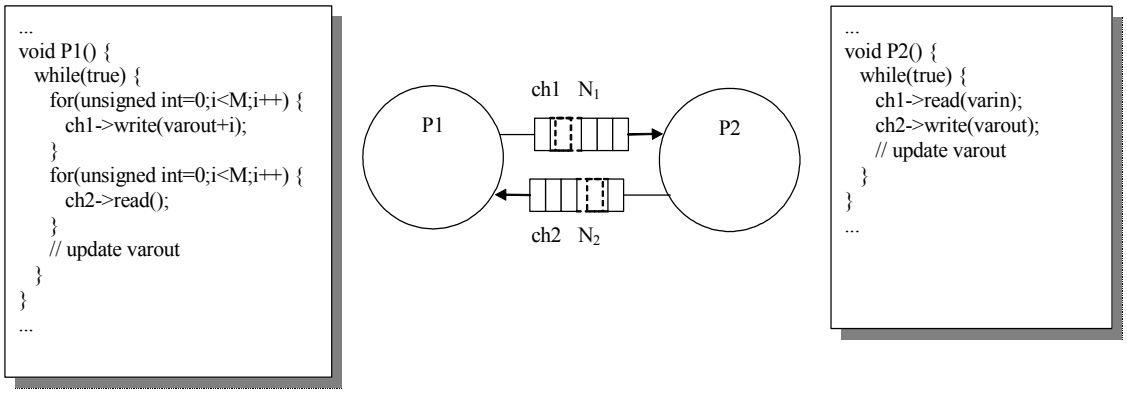



**Figura 2-5. Interbloqueo por condiciones de escritura dependiendo del tamaño de los canales fifo.**

Existen alternativas para evitar esta situación de interbloqueo. Una consiste en intercalar los accesos de escritura y lectura en P1. De esa forma se tiene una solución mínima en exigencias de tamaño de fifo. Una solución independiente de la secuencia interna de accesos consiste en asumir $M \leq 2N+1$. Una tercera solución es la dada por Kahn, asumir $N=\infty$. Entonces, se elimina la condición de interbloqueo para cualquier M. Es decir, una red de Kahn con la estructura de la Figura 2-5 no presentaría interbloqueo en el caso de $M>2N+1$. La red de Kahn sólo presenta la posibilidad de interbloqueo debido a las condiciones de bloqueo en espera a lectura de datos. Este tipo de interbloqueo se denomina *interbloqueo real* **[Park95]** o *sincronización bloqueante* **[Jan04]**. La principal desventaja de la red de Kahn es que es un modelo ideal, por tanto, no implementable físicamente debido a la longitud infinita de las fifos.

Otro modelo de relevancia es el MoC CSP (Procesos Secuenciales Comunicantes) **[Hoa78]**. El modelo CSP define de nuevo una red de procesos que, en este caso, se comunican por medio de un conjunto amplio y versátil de comandos de comunicación entre procesos. Todos esos comandos desarrollan los comandos de guarda de *Dijkstra* **[Dijk75]**. La mayoría de comandos se construyen sobre una semántica de sincronización básica, denominada *rendezvous*. La semántica de sincronización *rendezvous* exige la llegada de todos los procesos involucrados antes de que cada uno de estos continúe su ejecución. Esta condición de bloqueo independientemente de la consideración de datos hace que se reconozca a este tipo de redes como redes *fuertemente acopladas* **[Jan04]**. Esta semántica es independiente del hecho de que realmente haya una transferencia de datos en el canal. Por ello, se puede considerar el MoC CSP como un MoC no orientado a datos. Sin embargo, también soporta la transferencia de datos. Por ejemplo, se puede usar un *rendezvous* unidireccional para garantizar una dependencia de datos. Una especificación así puede ser también determinista, pero el acoplo entre procesos es más fuerte que en un modelo de estructura similar bajo el MoC KPN o una variante basada en colas de tamaño limitado.

En **[Hoa78]** se sostiene además que la composición de procesos secuenciales comunicantes es un método de estructuración fundamental. Es decir, propone que los

procesos y los mecanismos de comunicación entre procesos deberían ser primitivas propias del lenguaje de programación. Uno de los ejemplos que más fielmente llevan a la práctica esta idea es el lenguaje ADA **[ADA97]**. Dicho lenguaje permite la declaración de tareas concurrentes que incluyen primitivas de aceptación y de llamada y que determinan los puntos del código de la tarea donde se da la sincronización tipo *rendezvous*. El *rendezvous* de ADA es algo más genérico que en **[Hoa78]**, ya que soporta, transferencia bidireccional de datos. Además, esa transferencia se describe dentro de un *cuerpo protegido*, que admite la especificación de una secuencia de sentencias ejecutada atómicamente en la sincronización *rendezvous*.

El trabajo de **[Hoa78]** se centra en la descripción formal de un lenguaje de programación concurrente basado en el *rendezvous* y los comandos de guarda. Sin embargo, no se sugieren pruebas ni métodos para asistir el desarrollo y asegurar la corrección de los programas desarrollados. De esta forma, el amplio conjunto de primitivas de **[Hoa78]**, así como de facilidades provistas por ADA, permiten al especificador incurrir fácilmente en construcciones indeterministas o afectadas por otros problemas típicos de la concurrencia. Por ejemplo, ADA soporta la primitiva *select*, que permite que una tarea se desbloquee por sincronizarse con una entre varias posibles tareas. Sin mecanismos adicionales, esta sentencia introduce indeterminismo.

Un grupo de MoCs importante es el de Flujo de Datos (MoC DF). Un flujo de datos (DF) admite una representación gráfica o grafo DF (DFG), que consta de nodos de cómputo y arcos, como únicos elementos que comunican los nodos. Aunque una red KPN admite una representación similar, los nodos y los arcos tienen restricciones semánticas y estructurales específicas que distinguen el MoC DF del MoC KPN.

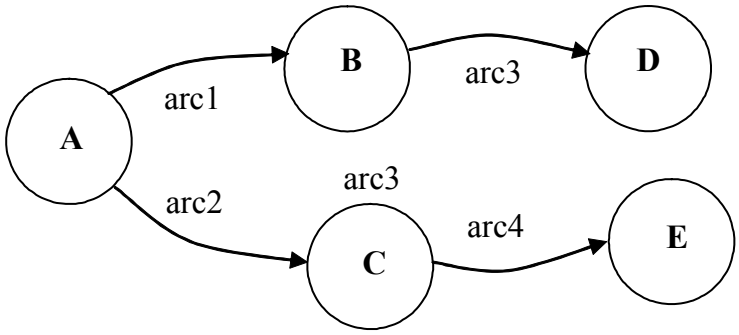



**Figura 2-6. Grafo de flujo de datos (DFG).**

En primer lugar, los *nodos* representan un cuanto de computación. Un cuanto de computación es un cómputo de datos que relaciona un conjunto finito de datos, proporcionado por los arcos de entrada al nodo, y el estado interno del nodo (si lo tuviere), con un conjunto finito de datos de salida, que son volcados en los arcos de salida. En cada arco de salida, cada unidad de datos transferida se obtiene según una relación funcional que toma como entrada las unidades de datos leídas en los arcos de entrada y el estado del nodo. Una restricción fundamental del cómputo, semejante a las que aparecen en las redes de procesos CSP, KPN y KPN con fifos limitadas, es que un nodo no comparte su ámbito de variables con otros nodos. Los nodos tampoco pueden acceder a variables propias del ámbito de otros nodos. Es decir, los nodos no utilizan un mecanismo de comunicación por variable compartida.

Se considera también que el cuanto de computación realiza un cómputo atómico. Esto es, no existe ningún tipo de bloqueo entre el inicio (disparo) y el final del cómputo del nodo, es decir, durante la evaluación del nodo. Los nodos no representan necesariamente hilos o procesos. Esta es una diferencia básica en la interpretación de los nodos DF respecto a los procesos KPN y CSP. En los flujos de datos, el hilo se define como una repetición homogénea de disparos de uno o más nodos **[LePa95]**.

Un grafo como el de la Figura 2-6 deja implícita también la consideración de dos tipos de nodos. Por un lado, nodos autónomos o activos, que no presentan arcos de entrada. Por otro lado, nodos dependientes o reactivos, que presentan al menos un arco de entrada. En algunos trabajos, se evita tal disquisición obligando a representar todos los nodos del DFG con arcos de entrada. En ese caso, los nodos autónomos tienen que representar su generación de datos o su disparo mediante arcos cerrados, que salen y retornan al nodo.

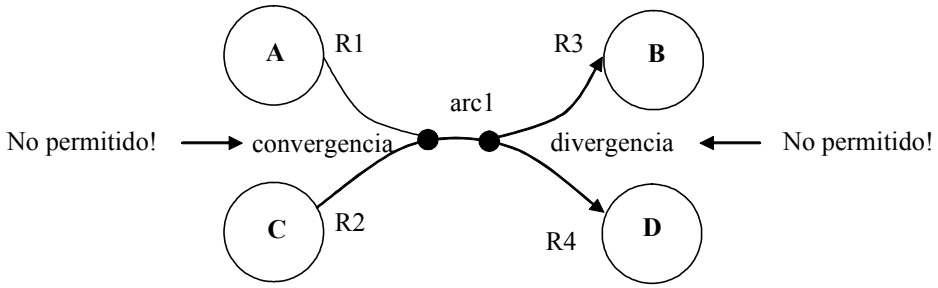



**Figura 2-7. La convergencia y divergencia de los arcos está prohibida en los flujos de datos.**

Los *arcos* representan la comunicación entre nodos de computación. En **[LeMe87]**, esa comunicación se reduce a una "*relación"* entre el nodo productor y el consumidor. El arco impone una semántica de comunicación unidireccional en la transferencia de datos, desde la salida del nodo escritor a la entrada del nodo lector. Además, impone restricciones como las de que un arco se origina desde un único nodo y llega exclusivamente a un sólo nodo. Es decir, no se permite la divergencia de un arco ni la convergencia de arcos (Figura 2-7). En cambio, sí es posible que llegue más de un arco de entrada al nodo y, así mismo, que un nodo escriba datos a más de un arco de salida. En resumen, los arcos restringen el número de escritores y lectores a 1. Esta restricción también se aplicaba en las comunicaciones FIFO en el MoC KPN.

En un flujo de datos existe además una semántica adicional que afecta a la semántica de comunicación tipo arco. Es la siguiente:

- Las tasa de escritura del arco. Se trata del número de unidades de datos escritos por el nodo escritor en el arco en cada una de sus evaluaciones.
- Las tasa de consumo del arco. Se trata del número de unidades de datos leídas del arco en cada evaluación o disparo de su nodo lector.
- El número de unidades de datos en el arco antes del primer disparo de cualquiera de los nodos, lector o escritor.

Además, el flujo de datos cuenta con unas reglas de disparo. Estas reglas establecen cuando se evalúa cada nodo. Pueden existir múltiples combinaciones de

reglas de disparo, aunque normalmente siempre dirigidas por datos **[LeMe87]**. Las reglas de disparo caracterizan fuertemente el tipo de MoC DF.

La primera ventaja que se deriva de las restricciones propias de los flujos de datos es que la división en nodos hace más fácil el mapeo de la especificación a lenguajes funcionales **[LeMe87][LePa95]**, tales como Haskell o C. Esto facilita y hace más eficiente la generación de software. Incluso si se considera un estado interno del nodo, este puede implementarse con facilidad en lenguajes como C.

En un caso general, las tasas de escritura y lectura pueden dinámicas. Esto es, pueden variar durante la ejecución, para cada nuevo disparo. Este tipo de modelos son flujos de datos dinámicos (DDF). En estos casos las ecuaciones de balance son variantes y la planificación estática de los disparos de los nodos no es posible. Las ecuaciones de balance buscan una situación estable que permita un procesado de datos sostenido, dadas unas tasas de entrada y de salida, con un aumento acotado del almacenamiento de unidades de datos en el sistema. El que esas ecuaciones sean variantes no impide necesariamente encontrar esa situación estable. Pueden haber casos en los que la necesidad de almacenamiento esté acotada, por ejemplo, si las variaciones de las tasas son finitas o periódicas, y conocidas.

Dentro de los MoC DDF, las Redes de Petri **[Petr62]** son un grupo importante, con un estudio extenso, donde la variación de esas tasas es, en general, dependiente de datos. Una red de Petri consta de lugares, transiciones, y arcos dirigidos. Matemáticamente, una red de Petri sirve para modelar sistemas distribuidos como un grafo bipartito dirigido. El grafo es bipartito en la medida en que hay dos tipos de nodos, los lugares y las transiciones. Se definen dos tipos de lugares, de entrada y de salida. Los arcos, y por tanto el grafo, son dirigidos, ya que van desde los lugares de entrada hacia las transiciones y desde las transiciones a los lugares de salida.

Los lugares pueden contener cualquier número de unidades de datos. Una *marca* es la distribución de unidades de datos en todos los lugares de la red. Las *transiciones* representan el cómputo que actúa sobre las unidades de datos de uno o más lugares de entrada. Se habla del disparo de la transición, que sólo se da si hay unidades de datos en todos los lugares de entrada asociados a la transición. Cuando la transición se dispara, consume las unidades de datos de sus lugares de entrada, realiza un procesado y emplaza un número específico de unidades de datos en cada lugar de salida. Ese cómputo es atómico. El número de unidades de datos consumidas en cada arco por una transición puede venir dado por un peso o bien se puede dibujar más de un arco asociado al mismo par lugar-transición.

Las redes de Petri presentan algunas particularidades importantes dentro del esquema DDF. El estado de la red de Petri se define en función de la marca. Es decir, las redes de Petri se fijan en el análisis del número de datos de entrada y salida de los nodos de cómputo (que en ese caso son las transiciones). No se centra en el estado de las variables del cómputo que conforman las transiciones, sino en cómo esos cómputos producen y consumen unidades de datos. De este modo, este MoC tiene propiedades importantes como la posibilidad de analizar si se puede alcanzar una marca (*reachability*), la acotación del número de unidades de datos en el sistema (*boundeness*), que se da siempre que cualquier estado alcanzable no supere un número finito de

unidades de datos y el análisis del nivel vida de la red de Petri (*liveness*) que se define en función de la vida de sus transiciones. La vida de una transición da una idea de cuantas veces se ejecuta en las secuencias de disparo de la red. Por otro lado, las redes de Petri son no deterministas ya que permiten la habilitación de múltiples transiciones al mismo tiempo, pudiendo dispararse cualquiera de ellas. Asimismo, no requieren que ninguna de las transiciones habilitadas se dispare. Se pueden disparar entre tiempo 0 e infinito (es decir, nunca).

Un caso particular de MoC DF son los Flujos de Datos Estáticos (Static DF), que mantienen constantes las tasas producción y consumo. A su vez, un caso particular de estos son los Grafos de Computación (CG) **[KaMi66]**. El MoC GC impone una serie de restricciones a los arcos y las reglas de disparo adicionales a las del MoC DF. Las restricciones respecto a los arcos, denominados ramas en **[KaMi66]**, afectan tanto a la semántica de computación como a la de comunicación. El MoC GC asocia a cada arco cuatro números, (Ai, Ui, Wi, Ti), que permanecen invariantes durante la ejecución. Los primeros tres números se han mencionado anteriormente. $A_i$, se refiere al número de unidades de datos presente en el arco antes del primer disparo de su nodo escritor o consumidor. $U_i$ y $W_i$ son las tasas de escritura y de lectura del arco. El último de los números, $T_i$, se refiere a la tasa de disparo. Se trata del número de unidades de datos que tiene que haber en el arco para que el nodo lector se pueda disparar. Se exige que $Ti \geq Wi$. En **[KaMi66]** se asume un acceso secuencial de lectura y de escritura de tipo fifo. De ese modo, el nodo, cuando se dispara, lee de un arco de entrada i las primeras Wi unidades de datos, es decir, las $W_i$ unidades de datos más antiguas en el arco. El modelo de **[KaMi66]** provee las siguientes reglas de disparo:

R.D.1. Un nodo i no se disparará mientras al menos uno de sus arcos de entrada tenga menos de $T_i$ unidades de datos. Equivalentemente, el nodo se puede disparar cuando todos sus arcos de entrada tienen al menos Ti unidades de datos.

R.D.2. La ejecución termina solo cuando no hay nodos que se puedan disparar.

R.D.3 Se define una "ejecución propia" como aquella en que todo nodo que tenga al menos Ti unidades de datos en cada arco de entrada se disparará tras un número finito de disparos de otros nodos.

Se asume además que no hay simultaneidad de disparos. Es decir, son reglas de secuencialización de la especificación concurrente que permiten la simulación e implementación del modelo en un elemento procesador. La R.D.1 establece las reglas de disparo de los nodos. Junto con la regla de especificación $Ti \geq Wi$, garantiza que no se de la situación de que no haya suficientes unidades de datos para consumir de cada arco de entrada durante la evaluación del nodo. Esto ratifica el carácter de "dirigido por datos" del MoC CG. La R.D.2 establece el criterio de terminación de la simulación, en tanto que la R.D.3 impide la inanición de alguno de los nodos. Nótese que, si bien las reglas de disparo de este MoC no fijan un tipo de planificador concreto, la regla R.D.3 ya establece una condición específica sobre el mismo.

Esas tres reglas de disparo son coherentes con un grafo que considere que todos los nodos presentan arcos de entrada. Este es el caso del trabajo original de **[KaMi66]**, tal y como se muestra en la Figura 2-8. En ella, el nodo que provee los datos de entrada

tiene un arco cerrado. Ese arco cumple $U_i$=0 y $A_i$=N. De ese modo, se garantiza que el nodo A puede consumir N unidades de datos, leídos de $W_o$ en $W_o$ unidades de datos.

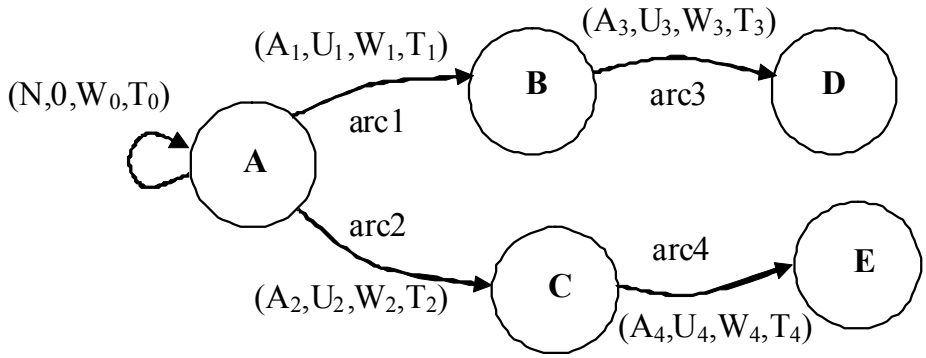



**Figura 2-8. Representación original de un grafo de computación.**

Alternativamente, puede ser interesante la disquisición entre nodo autónomo y reactivo. En ese caso, el grafo de la Figura 2-8 se representaría como en la Figura 2-9. Es posible adaptar las reglas del MoC GC a la representación de la Figura 2-9 y mostrarlo como un caso particular de MoC DF asumiendo que existen nodos autónomos y reactivos. Los nodos autónomos serían aquellos que no tienen ningún arco de entrada (como A). El resto (nodos B, C, D y E) serían nodos reactivos.

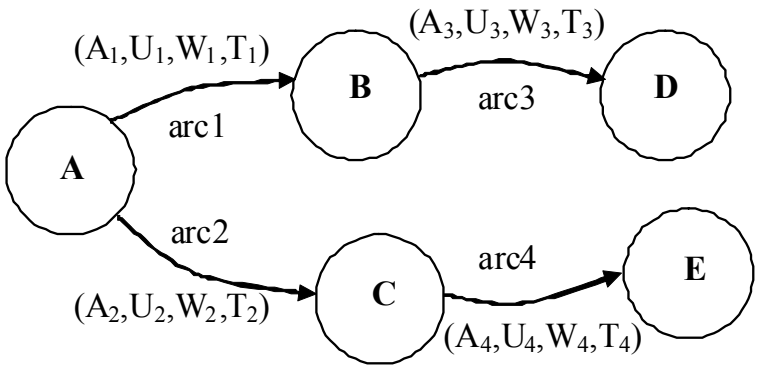



**Figura 2-9. Grafo de Computación con nodos autónomos y reactivos.**

En este caso, el enunciado de las reglas de disparo anteriores queda como sigue:

R.D.1. Un nodo dependiente i no se disparará mientras todos sus arcos de entrada no tengan al menos $T_i$ unidades de datos. Cualquier nodo autónomo que tenga datos que escribir podrá dispararse.

R.D.2. La ejecución termina solo cuando no hay nodos que se puedan disparar, sean estos dependientes o autónomos.

R.D.3 Se define una "ejecución propia" como aquella en que todo nodo que se pueda disparar lo hará tras un número finito de disparos de otros nodos.

De esta forma la terminación de la ejecución de la especificación depende ahora de que los nodos autónomos no generen indefinidamente datos. En caso contrario, la ejecución seguirá indefinidamente. Asimismo, la R.D. 3 elimina el problema de inanición (nodos que no se ejecuten nunca) ya que impide, por ejemplo, que el nodo A se dispare y escriba indefinidamente, si dar lugar al disparo del resto de los nodos.

Si además se quiere conseguir una acumulación mínima de unidades de datos en el grafo se puede añadir la siguiente regla de disparo:

R.D.4. Un nodo no autónomo que se pueda disparar lo hará antes que cualquiera de los nodos autónomos.

Nótese de nuevo que las reglas la R.D.3 (y la R.D.4 en su caso) se refieren al planificador.

Con esta serie de reglas, los GC proveen una serie de ventajas adicionales a las propias de los MoC DF. Además de garantizar el determinismo, las reglas de disparo anteriormente expuestas garantizan que la ejecución es finita (*bounded*). Además, el MoC GC da criterios para decidir la terminación (*termination*), la acotación (*boundedness*), y la posibilidad de planificación estática de un grafo **[KaMi66]**.

Los flujos de datos síncronos o MoC SDF **[LeMe87]** son un caso particular de flujos de datos que ha tenido un fuerte impacto en campos de aplicación importantes, tales como el procesado digital de señal (DSP). El MoC SDF es un caso particular de MoC GC en el que la tasa de consumo y la de disparo de cada nodo son iguales para cada arco del grafo (∀i Wi=Ti). Por lo tanto, los parámetros que acompañan al grafo asociado al MoC SDF o SDFG (Figura 2-10) son las tasas de consumo (o disparo) y producción de los arcos. El número de unidades de datos inicial es usualmente 0 por defecto y no se representa en el SDFG, aunque se puede asignar un valor distinto.

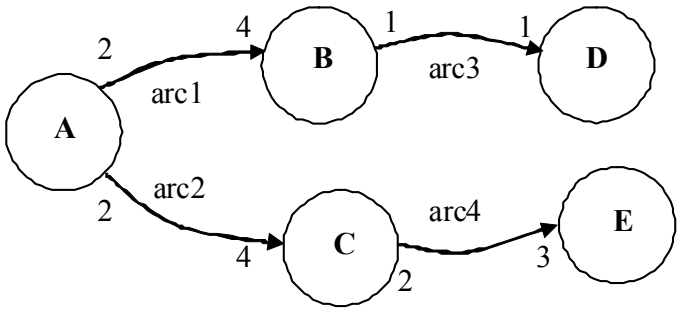



**Figura 2-10. Representación de un Grafo de un Flujo de Datos síncrono (SDFG).**

En algunos casos, por ejemplo, cuando existen lazos de realimentación, es preciso proporcionar un número de unidades de datos iniciales en el arco que cierra el lazo para que continúe o arranque la ejecución. Por ejemplo, en el SDFG de la Figura 2-11, es preciso inicializar el arco *arc4* con dos unidades de datos para garantizar la continuidad de la ejecución.

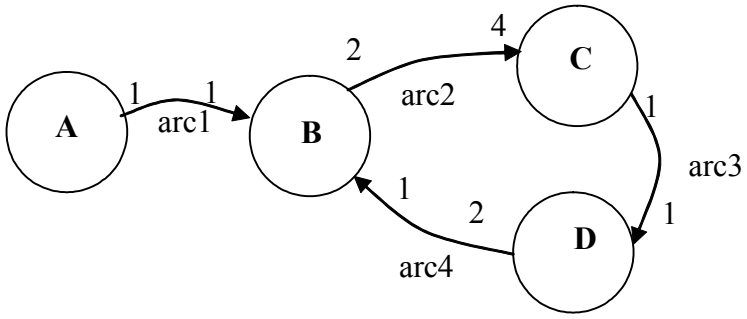



**Figura 2-11. Grafo SDF con realimentación.**

El MoC SDF permite analizar las ecuaciones de balance y obtener y decidir la planificabilidad estática de la especificación, con lo cual se puede ganar en tiempo de simulación y/o ejecución del sistema. El resto de restricciones y propiedades del MoC SDF son heredadas de las comentadas para el MoC CG. A su vez, dentro del MoC SDF se pueden encontrar casos particulares. Por ejemplo, un MoC SDF Homogéneo o MoC HSDF es un MoC SDF en el que se fuerza a que todas las tasas de producción y consumo sean 1.

Estos últimos ejemplos ilustran que puede establecerse una clasificación de MoCs, jerárquica en tanto algunos MoCs heredan las restricciones de otros y les van añadiendo otras adicionales. De esa forma, se obtienen nuevas propiedades, normalmente a costa de un estilo de especificación menos flexible. Algunos niveles de condiciones suelen entenderse en la bibliografía más propios del flujo implementación que del de especificación. Este suele ser el caso de las condiciones impuestas a la planificación. Este es el caso, por ejemplo, de trabajos que encuentran planificaciones eficientes de grafos SDF para reducir la necesidad de tamaño de almacenamiento dedicado a la transferencia de datos entre nodos.

Los MoCs descritos hasta aquí (KPN, CSP, Flujos de Datos, Redes de Petri, CG, SDF, etc) pertenecen a un tipo de MoCs denominados atemporales. Sin entrar en más detalle ahora, el único nivel de información temporal que manejan estos MoCs es el orden parcial entre los eventos de la especificación. Este orden parcial es forzado por la semántica de las facilidades empleadas y la topología de la especificación.

Otro grupo importante de MoCs son los sincronizo. Estos son MoCs que manejan un nivel más detallado de información temporal, tal como la ranura temporal, en el caso del MoC Síncrono Reactivo (o SR), o como el ciclo de reloj, en el caso del MoC Síncrono de Reloj (o CS) **[Jan04]**. De este modo, estos MoCs son capaces de garantizar propiedades como el determinismo por medio de condiciones temporales, en tanto que relajan otras restricciones presentes en los modelos atemporales. Por ejemplo, en los modelos sincronizo se suele permitir que existan varios procesos lectores de un canal de comunicación. Esto es debido a que en los canales de comunicación utilizados en los modelos síncronos el dato interno puede persistir entre diversos accesos de lectura (acceso no destructivo), durante el ciclo de reloj o la ranura temporal, dependiendo del tipo de MoC síncrono.

El MoC Síncrono Reactivo o MoC SR **[BeBe91]** es un MoC síncrono que ha tenido aplicación en varias metodologías de especificación. Entre ellas, el lenguaje Esterel **[BoSi91][Berry00][BCEH03]**, el dominio SR de Ptolemy **[Edwa97]**, y la implementación original de ForSyDe **[SanA03]**. El MoC SR parte de la hipótesis de *sincronía perfecta*, que establece que el cómputo y la comunicación consumen un tiempo cero. Como consecuencia, se da una reacción instantánea en la que los datos de salida aparecen al mismo tiempo que los datos de entrada, es decir, son síncronos.

El manejo del tiempo en un MoC SR es de mayor nivel de detalle que en los MoC atemporales, ya que la condición de sincronía perfecta se añade al orden parcial entre eventos. De este modo, en el MoC SR, toda la actividad del sistema se concentra en puntos específicos del eje temporal denominados *ranuras temporales* o, simplemente, *ranuras*. Todos los eventos situados en la misma ranura temporal son síncronos entre sí.

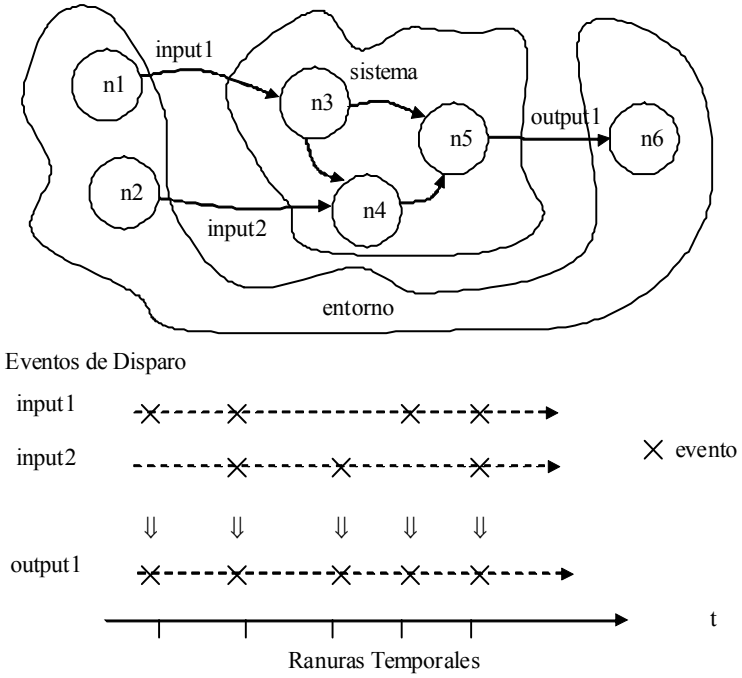



**Figura 2-12. En un MoC SR la actividad del sistema se concentra en ranuras temporales.**

No se imponen más restricciones ni detalle de información temporal a la etiqueta que representa una ranura más que la de que el conjunto de ranuras sea un conjunto totalmente ordenado, tal como el de los números naturales **[Jan04]**. No se requiere una distancia regular entre las etiquetas temporales de cada ranura. Ni siquiera se requiere una interpretación de tiempo físico para esa etiqueta temporal. El MoC SR obliga a realizar una distinción entre los nodos autónomos que no necesitan ser disparados y que generan los eventos o datos (usualmente, parte del entorno) y las entidades reactivas que se disparan y computan a consecuencia de recibir esos eventos o datos (usualmente, representando el sistema reactivo). La hipótesis de sincronía perfecta exige que la funcionalidad reactiva no implique retardos ni accesos bloqueantes que puedan provocar directa o indirectamente un retardo.

Una especificación Esterel tiene las mismas construcciones que un lenguaje imperativo, como C, pero incluyendo soporte de concurrencia, expulsión y un modelo de tiempo síncrono. Aparecen facilidades de especificación como la *señal* y la *variable*. La primera es un mecanismo de transferencia de eventos entre procesos, en tanto que la variable es el elemento empleado para la transferencia de datos entre procesos. La señal Esterel tiene una semántica distinta de la señal en HDLs como VHDL o de SystemC (*sc_signal*). En estos dos últimos casos, la señal es un canal de comunicación que acarrea tanto transferencia de datos como de eventos entre procesos.

Algunos problemas son específicos del modelo temporal del MoC SR. Por ejemplo, en Ptolemy y ForSyDe, la sincronía de señales se interpreta de forma que en la comunicación entre nodos se transfiere siempre una unidad de datos por cada ranura temporal. Por esta razón, estos modelos introducen el evento vacío ($\perp$), cuyo empleo provoca trasiegos de datos de tipo "no evento" y que supone una sobrecarga en la simulación de estos modelos. Por otro lado, la implicación de que los datos de salida y

los de entrada se encuentren en la misma ranura temporal, a pesar de existir una relación de causalidad entre ellos, trae consigo paradojas cuando hay situaciones de reconvergencia y lazos de realimentación. En Ptolemy o ForSyDe, por ejemplo, el problema de la realimentación se resuelve rompiendo el lazo por medio de facilidades de especificación que introducen explícitamente retardos equivalentes a una ranura temporal y mediante el operador de realimentación, que permite establecer explícitamente la necesidad de resolver una ecuación de punto fijo. Por lo tanto, el manejo del nivel de detalle de modelo síncrono aparece explícitamente reflejado en la especificación.

El MoC síncrono de reloj (CS) se puede entender como un caso particular del MoC SR. En una especificación CS, cada nodo de cómputo es concurrente con el resto y se dispara como respuesta a un evento global y común, generado por el *reloj del sistema*. El reloj del sistema puede ser un proceso o una facilidad de especificación primitiva. Normalmente no transfiere datos, sólo los eventos generales de sincronización. Por tanto, las *ranuras* de una especificación CS son las provocadas por el reloj y se denominan usualmente *ciclos*. Esta particularización simplifica significativamente las condiciones temporales requeridas para la transformación de un modelo CS en una implementación física equivalente. En esto se basan las herramientas de síntesis lógica para implementar HW digital síncrono.

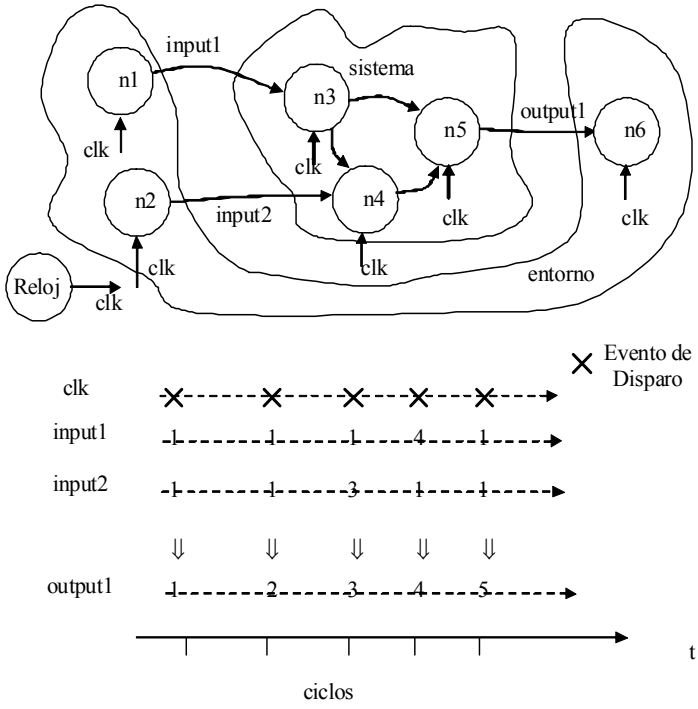



**Figura 2-13. En el MoC CS el reloj se encarga de disparar los procesos.**

En el MoC SR los canales de comunicación sincronizan los procesos y transfieren datos. En el MoC CS es también común el empleo del mismo tipo de facilidad de comunicación para transferir los eventos de sincronización del reloj y para la transferencia de datos entre procesos. Por ejemplo, en VHDL se emplea la *señal* tanto para transferir los eventos de reloj como para transferir valores entre procesos. No

obstante, también se separa el conjunto de instancias de tipo *señal* VHDL que se dedica a la comunicación de procesos (que se dedican a transferir datos), del conjunto de instancias de tipo *señal* VHDL que distribuye el reloj (y que se dedica a transferir eventos de disparo).

Una característica típica del canal *señal* en modelos CS es la persistencia del valor asociado durante el ciclo. De ese modo, la semántica de escritura/lectura es más una semántica de actualización/muestro que una semántica de envío/consumo tal y como ocurre en los canales empleados en los modelos atemporales. Además, como se aprecia en la Figura 2-13, todas las *señales* de la especificación presentan un valor en cada ciclo. Las reglas de diseño de hardware digital sintetizable constituyen un ejemplo del MoC CS sobre lenguajes como VHDL, basados en un MoC DE de tiempo estricto **[ViSa93]**.

## 2.2.2 Metamodelos

Un metamodelo provee un marco más abstracto para la descripción y análisis de un sistema. Más aún, un metamodelo provee un marco general que permite el entendimiento, la comparación y clasificación de MoCs entre la variedad existente.

**e** (evento): par valor-etiqueta

**s** (señal): conjunto de eventos

**s-tuple**: N señales

$\mathbf{S^N}$: conjunto de todas las s-tuplas posibles

**P** (process): conjunto de s-tuples, subconjunto de $S^N$

**Behavior**: s-tupla que satisface el proceso ($s \in P$)

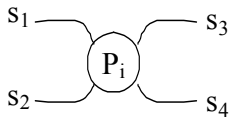



**Figura 2-14. En el metamodelo LS los procesos son las relaciones posibles entre señales.**

Se han propuesto varios metamodelos. En **[LeSV98]** se define un marco general (que se denominará metamodelo LS en adelante) en el que una especificación concurrente es abstraída como un conjunto de *procesos* conectados mediante *señales*. Las señales y los procesos del metamodelo LS tienen un definición, además de distinta de las manejadas en Esterel, VHDL o SystemC, más abstracta y general, aunque exenta de ambigüedades. En el metamodelo LS, una señal se describe como una colección de eventos, siendo un evento un par etiqueta-valor. Los procesos se definen como una relación entre señales. Es decir, un proceso no es más que una relación que define el conjunto de posibles comportamientos. Un comportamiento posible es una tupla de N señales, en las que se establecen las posibles señales de salida dadas unas señales de entrada. Por lo tanto, una forma equivalente de definir el proceso es como el conjunto de N-tuplas válido. Este conjunto es, en general, un subconjunto de entre todas las posibles N-tuplas ($S^N$). El metamodelo LS soporta también composición de procesos.

El metamodelo ForSyDe (Diseño Formal de Sistemas) **[Jan04]** es la evolución de la semántica formal originada en **[SanA03]**. Este metamodelo esta basado también en procesos, eventos y señales. Sin embargo, aunque similar en abstracción y generalidad al metamodelo LS, existen diferencias. En primer lugar, en el metamodelo ForSyDe, un evento sólo tiene asociado un valor (a diferencia del evento del metamodelo LS, que

tiene también una etiqueta asociada). Además, en el metamodelo ForSyDe, una señal es una *secuencia* de eventos (en el metamodelo LS la señal es un *conjunto* de eventos). En el metamodelo LS el orden de los eventos de una señal viene determinado por las etiquetas temporales de los eventos. En cambio, en ForSyDe, aunque el evento no tiene asociado una etiqueta (sólo un valor), la señal implica una ordenación entre los eventos que la forman, y por tanto, el evento tiene una etiqueta temporal implícita que viene dada por el orden de dicho evento en la señal. Es decir, en la señal ForSyDe $s_1=\{e_1,e_2\}$, el evento $e_1$ precede al evento $e_2$. En ForSyDe se definen tres tipos de eventos: atemporales ($e$), síncronos ($\bar{e}$) y temporales ($\hat{e}$). Un evento síncrono puede tomar, además de un valor, el pseudo-valor "⊥". Un evento temporal es un evento síncrono al que se le otorga una noción de tiempo físico. En consecuencia, en ForSyDe se definen tres tipos de señales, en función del tipo de evento del que están compuestas: señales atemporales ($s$), síncronas ($\bar{s}$) y temporales ($\hat{s}$).

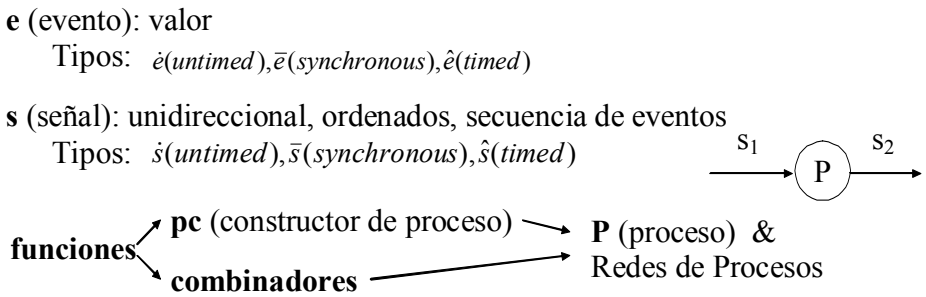



**Figura 2-15. En ForSyDe los procesos se generan mediante constructores.**

Estas diferencias en conceptos básicos implican diferencias en cómo cada metamodelo maneja el tiempo, el cómputo y la comunicación.

En cuanto al modelo del cómputo, en ForSyDe se modela el proceso desde el punto de vista introspectivo, es decir, cómo se construye el proceso, en tanto que en el metamodelo LS se modela desde un punto de vista externo, a partir de la relación de las señales de entrada y salida del proceso. En ForSyDe, el proceso es considerado como una máquina de estados finita que consume y produce eventos. Como se ha visto, esos eventos son unidades de datos que pueden llevar asociadas un tipo y una noción temporal. Hay restricciones en la forma en la que se construyen los procesos. Se provee un conjunto limitado de constructores de proceso y tres combinadores. Los constructores de proceso son plantillas parametrizables que instancian procesos a partir de funciones y señales. Los combinadores son operadores que toman procesos o redes de procesos y los componen resultando en redes de procesos más complejas.

En cuanto a la forma de modelar el tiempo, en el metamodelo LS, las señales portan una información temporal explícita, dado que las etiquetas de los eventos establecen las relaciones temporales de los eventos. En cambio, en ForSyDe, como se ha explicado, los eventos portan tal información de forma implícita y en función de los tipos de eventos transferidos. Otra parte de la información temporal radica en el tipo de constructor de proceso y, por tanto, cómo los procesos consumen y generan los eventos. Los procesos manejan un concepto del avance del tiempo ya que su actividad se divide en ciclos de evaluación. Cada evaluación del proceso implica un mapeo de subsecuencias de las señales de entradas a subsecuencias de las señales de salida.

En cuanto al modelo de la comunicación, el metamodelo LS apenas refleja la semántica de comunicación existente entre procesos. En el metamodelo ForSyDe, la semántica de comunicación se fija toda vez que se fija el constructor de proceso y el modelo de tiempo manejado. Asimismo, en ForSyDe la señal representa un flujo unidireccional de eventos, lo cual no es requerido en el metamodelo LS.

Además de ForSyDe, **[Jan04]** presenta una herramienta para la clasificación y comparación de MoCs denominada *metamodelo de Rugby*. En ese metamodelo, los MoCs son clasificados y estudiados atendiendo al nivel de abstracción o detalle manejado en cuatro *dominios*. En ese contexto, un *dominio* es un aspecto importante de la especificación que puede ser analizado por separado del resto. El metamodelo de Rugby distingue cuatro dominios: *Datos*, *Computación*, *Comunicación* y *Tiempo*. Los dominios se representan como ejes confluyendo en los extremos más abstractos (especificación) y más concretos (implementación). El resultado (Figura 2-16) es un diagrama con forma de balón de Rugby del cual el metamodelo toma su nombre.

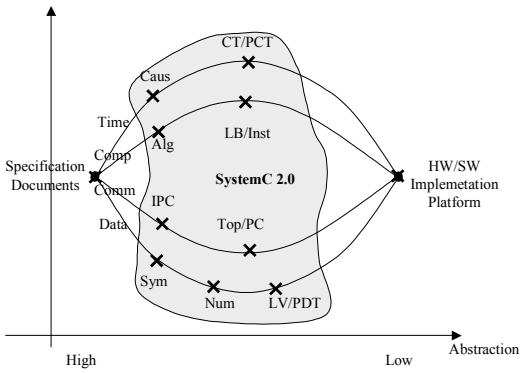



**Figura 2-16. Representación de las coordenadas de Rugby de SystemC.**

La idea es que un MoC soporta en cada uno de estos dominios uno o varios niveles de abstracción. Así, es posible clasificar un MoC determinando las coordenadas por cada uno de los ejes del diagrama. Por ejemplo, los lenguajes de descripción de hardware (HDLs) permiten usar en la especificación desde tipos de datos *bit* hasta tipos más abstractos, como enteros. Por lo tanto, la coordenada de los HDLs en el dominio de *Datos* del metamodelo de Rugby tiene cierta dimensión, en lugar de ser puntual.

Si bien el metamodelo de Rugby no considera todos los posibles aspectos distintivos entre MoCs, al menos considera los más importantes y permite una clasificación gráfica y compacta de estos. De esta forma, con este metamodelo se puede estudiar y representar, por ejemplo, cual es el nivel de detalle que es capaz de soportar el lenguaje SystemC en cada dominio. Ese ejercicio se hizo en **[CHSV05]**, dando como resultado la Figura 2-16, que muestra cómo el lenguaje SystemC soporta varios niveles de abstracción en cada dominio. En **[Jan04]**, se propone también una taxonomía de modelos de computación derivada del nivel de abstracción manejado en el dominio *Tiempo* del metamodelo de Rugby. Esta clasificación distingue entre MoCs *atemporales*, *síncronos* y *temporales*.

Los metamodelos proveen una base formal a la especificación heterogénea. Sin embargo, un metamodelo no es una metodología ni una herramienta de especificación. En la siguiente sección se repasan diversos marcos de especificación heterogénea.

## 2.2.3 Marcos de Especificación Heterogénea

### *2.2.3.1 Entornos de cosimulación.*

La necesidad de mejora productiva y la existencia de diversos dominios de diseño (programación de software, diseño de hardware digital, analógico, etc), ha provocado una evolución paralela de distintas metodologías de especificación. En cada dominio de diseño se han desarrollado lenguajes y metodologías de especificación que presentan primitivas, sintaxis, semántica y entornos de simulación adaptados, eficientes y característicos. Así, este desarrollo ha sido tradicionalmente separado, aunque compartiendo una tendencia hacia la abstracción. Por ejemplo, mientras los compiladores y lenguajes de programación han ido evolucionando para soportar objetos, herencia, polimorfismo, etc, la evolución en el dominio de diseño hardware ha hecho posible que éste se describa de forma más abstracta mediante lenguajes de descripción de hardware (HDL). Sin embargo, el desarrollo de los sistemas embebidos y el co-diseño HW/SW ha acrecentado la necesidad de acoplar las distintas metodologías de especificación.

De este modo, los diversos entornos de cosimulación que han ido apareciendo en el mercado pueden ser considerados como un primer precedente de entornos de especificación heterogénea. Un primer ejemplo lo proporcionan las herramientas que soportan la cosimulación de varios lenguajes HDL (por ejemplo, VHDL y Verilog). En este caso, estos entornos resuelven principalmente problemas de conexión sintáctica., ya que semánticamente, los elementos en uno y otro lenguaje son muy cercanos. Sin embargo, SystemC permite la introducción de modelos más abstractos, y una cosimulación, por ejemplo, VHDL-SystemC, puede involucrar la conexión de MoCs distintos y, por tanto, añadir problemas semánticos a la conexión de modelos.

Casos que pueden ser considerados como ejemplos básicos de especificación heterogénea son aquellos en los que se cosimula un programa descrito mediante un lenguaje de programación de alto nivel (como C, ADA, Pascal, etc) junto con una descripción HDL (como VHDL o Verilog) **[Gup02]**. Muchas herramientas de simulación de HDLs proveen una API de programador. Esta API es habitualmente C (API-C). Por ejemplo, en el caso del simulador *VSS* de *Synopsys*, esa API se denomina CLI (*C Language Interface*) **[CLI00]**. Aún más, en el lenguaje Verilog, la interfaz de acceso al simulador HDL está estandarizada en el PLI (Programming Language Interface) de Verilog (IEEE 1364) **[Shu02]**. En todos estos trabajos, se provee un medio para conectar la ejecución de un programa C con la simulación HDL. Sin embargo, estas facilidades no proveen ni un marco o lenguaje común de especificación ni una máquina de simulación común. Aún más, la semántica y la metodología de conexión de MoCs queda abierta. Es decir, el usuario tiene que preocuparse de describir la sintaxis y la semántica de la conexión. Esta flexibilidad no es siempre deseable. Por un lado implica un trabajo de descripción de la interfaz. Por otro lado, la implementación de una semántica de conexión correcta puede exceder el tiempo y/o conocimientos con los que cuenta el especificador. Por otro lado, una semántica de conexión adecuada evita repercusiones indeseadas en la semántica global del sistema.

Por tanto, estas soluciones no se pueden considerar entornos de especificación que integren la heterogeneidad vertical y horizontal de forma global y sistemática, tal y como exige una metodología de especificación actual.

### *2.2.3.2 Ptolemy II y CAL*

Un avance cualitativo importante hacia un marco unificado para la especificación heterogénea lo representa Ptolemy II **[BLLN07]**. Ptolemy II es un marco para el modelado heterogéneo basado en componentes. Ptolemy II provee un entorno de captura gráfica denominado *Vergil* y toma Java como lenguaje de implementación.

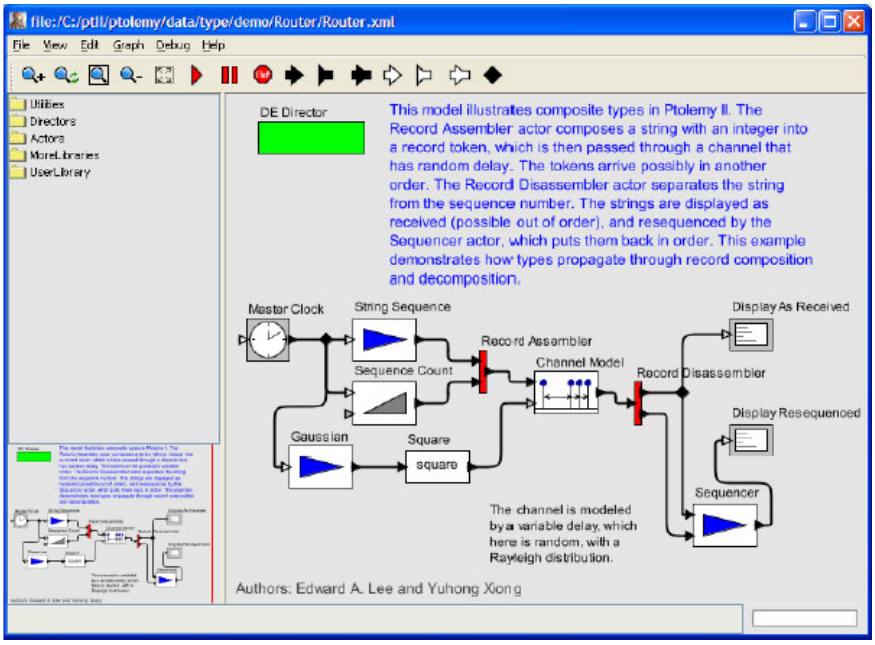



**Figura 2-17. Ejemplo de Modelo Ptolemy Visualizado en Vergil (tomado de [BLLN07]).**

En Ptolemy II, la especificación está formada por componentes denominados *actores*. Un *actor* puede ser compuesto o primitivo. En el primer caso el actor se compone de varios actores. En el caso de ser primitivo, el actor contiene una funcionalidad específica descrita en Java. Esa funcionalidad se ejecuta en diferentes etapas de simulación bien definidas, separadas y controladas por una clase denominada *director*. La funcionalidad del actor en esas etapas se corresponde con la de una serie de métodos internos del actor (*initialize*, *prefire, fire, postfire, etc*) e invocados por la clase directora que constituyen una interfaz fija entre director y actor. Por tanto, la semántica del componente depende tanto de la funcionalidad de cada actor, dividida en esos métodos, como de la propia clase directora. Ésta es la que fundamentalmente define el MoC al que está sometido el componente, que, en Ptolemy, se denomina *dominio*. En términos generales, el director establece las reglas de ejecución para la estructura interna del actor. Cuando ese actor es primitivo, el director asociado controla el orden de ejecución de sus métodos *initialize*, *prefire*, etc. Cuando el actor es compuesto, el director controla qué actores (y, por tanto, qué directores hijo) disparar en cada momento. Por tanto, en Ptolemy hay una asociación MoC-componente (o *dominio–actor*) reflejada en la asociación *director-actor*. También hay una heterogeneidad jerárquica, esto es, actores bajo MoCs distintos pueden componer un nuevo actor bajo

otro MoC distinto, dando lugar a una composición de MoCs jerárquica. En Ptolemy aparece el concepto de componentes *polimórficos de dominio*, según el cual, la misma descripción de un actor puede reutilizarse en dominios distintos.

La comunicación entre actores se da por medio de entidades de comunicación genéricas denominadas *puertos*. Esto introduce una clara separación entre computación y comunicación. Los *puertos* de Ptolemy sirven para indicar una conexión entre actores que permite la transferencia de *unidades de datos* entre los mismos. Sin embargo, estos *puertos* no tienen un contenido semántico o un control sobre la simulación significativo. Son los directores los que marcan los accesos de escritura y lectura en cada dominio. Eso permite simplificar las interfaces de MoCs, para las que no se precisa una infraestructura específica adicional.

Hay cuestiones que considerar antes de adoptar Ptolemy como metodología de especificación heterogénea. En primer lugar, Ptolemy está diseñado fundamentalmente como un entorno de modelado y simulación, restando consideración a facilitar el soporte de una metodología de diseño completa. A pesar de ello, existen algunos trabajos de generación de software embebido **[PHLB95]**, mencionados más adelante.

Por otro lado, ciertos conceptos avanzados, como el *polimórficos de dominio*, pese a que permiten la reutilización de la descripción del componente, pueden ser peligrosos a nivel metodológico. El usuario no debe olvidar que necesita considerar la semántica de la clase directora además de la de los actores para hacer una correcta interpretación semántica de la especificación.

Además, la definición y normalización en Ptolemy de una estructura de ejecución del actor primitivo dividida en los métodos *fire*, *prefire, etc*, si bien facilita conceptos avanzados como el *polimorfismo de dominio* y la *heterogeneidad jerárquica*, hace que el uso del lenguaje de implementación (*Java* en este caso) sea más complicado, al fragmentar y dispersar la funcionalidad del componente. Esto es comprensible en tanto que Ptolemy se oriente a la captura gráfica, dejando al lenguaje de implementación un rol secundario. Sin embargo, en una metodología de especificación, el lenguaje es tan importante o más que su correspondencia con una serie de primitivas gráficas. El lenguaje obliga usualmente a una definición concreta de la semántica de cada primitiva y elemento constructivo a nivel de manual de referencia. El empleo de las primitivas gráficas, si bien cómodo, puede llevar a un empleo intuitivo y ambiguo, y por tanto, insuficiente de la semántica del componente y de la especificación.

Prueba de que la especificación Java de Ptolemy es más compleja y redundante de lo que sería deseable es el desarrollo del lenguaje CAL (Caltrop *Actor oriented Language)* **[CAL08]**, un lenguaje surgido del proyecto *Caltrop*, a su vez parte del proyecto *Ptolemy*. CAL es un lenguaje textual para describir actores propios del dominio o MoC de flujos de datos. El empleo de CAL hace la escritura de estos actores más accesible y compacta, reduciendo la probabilidad de errores de especificación respecto al empleo directo de *Java* bajo la interfaz definida por *Ptolemy*. Una ventaja adicional es la de que sería un lenguaje adecuado para la extracción de información para el análisis del modelo. En **[Wer02]** se presenta un generador de código capaz de extraer código Java de Ptolemy a partir de CAL. Sin embargo, como se ha comentado, CAL se limita al MoC de flujo de datos. Además, CAL tampoco se encarga de definir una

semántica de ejecución precisa, dejando ciertos puntos abiertos a la interpretación que realice el generador de código o la máquina de simulación correspondiente **[CAL08]**. Por ejemplo, CAL no define cómo resolver la ambigüedad que se produce cuando la ejecución de dos o más acciones es posible dada la coincidencia de dos o más condiciones de disparo. CAL tampoco fija como parte de sus objetivos la definición de tipos y la semántica de sus operadores asociados, ni tampoco recomienda su uso para definir funciones y procedimientos.

Por tanto, aunque CAL permite una especificación abstracta y compacta, no se orienta al soporte ni de heterogeneidad horizontal ni vertical en tanto que se ciñe a un MoC específico y no se ocupa de los posibles pasos de refinamiento que pueden involucrar mayor detalle en el manejo de tipos, semántica de ejecución, etc.

#### *2.2.3.3 ForSyDe*

ForSyDe se constituye como una metodología **[SanA03]** que permite la producción de una especificación del sistema desde un alto nivel de abstracción. La característica más remarcable de ForSyDe es su soporte formal, que va desde aspectos básicos de metamodelado (introducidos en la sección 2.2.2), hasta el soporte de un flujo de implementación a través de la sucesiva aplicación de métodos formales de refinamiento del diseño **[Jan04]**. Estos refinamientos generalmente conllevan la transformación de MoCs. Por lo tanto, este es un ejemplo de metodología que ha desarrollado la heterogeneidad vertical. La metodología ForSyDe ha sido implementada en la librería estándar ForSyDe **[SanB03]**, que está basada en el lenguaje funcional Haskell. Un problema de esta metodología viene dado con el empleo de este lenguaje, que no tiene un gran impacto en la comunidad de diseñadores de sistemas embebidos o de programadores. El empleo de lenguajes de especificación, bien basados en HDLs, bien en lenguajes de alto nivel con un grado de penetración importante, exigiría un menor esfuerzo de aprendizaje y de ampliación, adaptación o mejora de las herramientas existentes.

#### *2.2.3.4 Metropolis*

*Metropolis* **[DDMP07] [BWHL03]** es un entorno de diseño que da soporte para especificación, simulación, verificación y síntesis, fundamentalmente de software. Una ventaja de *Metropolis* es que provee un marco formal y riguroso. Ese marco permite la conexión o incorporación al mismo de herramientas de verificación y síntesis. Por otro lado, en general, cualquier herramienta o *back-end* requeriría un analizador sintáctico.

Al igual que Ptolemy, *Metropolis* se basa en Java, pero con una aproximación diferente. *Metropolis* soporta heterogeneidad en dos niveles. El nivel inferior consiste en una infraestructura de clases con una semántica bien definida y lo suficientemente generales como para soportar modelos de computación existentes y otros nuevos, que constituyen un segundo nivel o capa. Por esta razón, esa infraestructura define en realidad un metamodelo, del cual derivar posteriormente modelos concretos. Los elementos básicos de la infraestructura son los *procesos*, *puertos*, *interfaces* y *media*:

**Procesos**: elementos de computación atómicos, definidos como secuencias de eventos y mapeados a hilos.

**Puertos**: Único medio de comunicación entre procesos.

**Interfaces**: Conjunto de métodos de comunicación.

**Media**: primitivas de comunicación entre procesos. Se conectan a los procesos a través de los puertos[2].

Hasta aquí, estos elementos tienen similitudes con SystemC. En cambio, otros elementos son específicos de *Metropolis*, como los *quantity manager* y *state media*. Los **quantity manager** fuerzan restricciones acerca de si los procesos deben ser planificados o no. Los quantity manager se comunican entre ellos mediante **state media**. *Metropolis* permite la mezcla de partes ejecutivas con partes declarativas. Entre estas últimas, es distintivo de *Metropolis* la posibilidad de especificar de forma abstracta, mediante un lenguaje lógico, requerimientos no funcionales y declarativos.

Sobre este marco general, *Metropolis* es capaz de soportar diferentes MoCs, principalmente a través de las distintas semánticas de comunicación de los *media*. Esta es una diferencia importante frente a Ptolemy, en donde gran parte de la semántica radica en la clase directora y las conexiones entre actores juegan un papel más secundario. En **[Lee06]** se afirma que este planteamiento le otorga a *Metropolis* más expresividad que a Ptolemy, aunque requiere más disciplina por parte del diseñador para evitar la generación de modelos incomprensibles. Para el soporte de especificaciones heterogéneas, *Metropolis* incluye en su infraestructura un proceso adaptador, denominado *adapter* o *wrapper*. El usuario debe describir explícitamente este proceso, que determina las condiciones de disparo del *proceso funcional* y cómo se conecta con el *media* que, a su vez, lo conecta al resto de procesos del sistema. Los *quantity manager* pueden desempeñar también un papel en la conexión de MoCs en tanto que dan cierto control sobre la semántica de ejecución de los componentes de la especificación.

#### *2.2.3.5 Lenguajes de especificación basados en C*

Las aproximaciones presentadas en las secciones anteriores se basan en *Haskell* y en *Java*. Una primera traba en el éxito de estos lenguajes para la especificación es que al ser interpretados o ejecutados sobre una máquina virtual, la velocidad de simulación conseguida es limitada. No obstante, existen ya compiladores como *gcj*, el compilador de *Java* de GNU, que atenúan este problema. Otra cuestión es, como se ha comentado anteriormente, que en algunas metodologías basadas en estos lenguajes, éstos tienen un rol más secundario, como lenguajes de implementación más que de captura. A eso se añade la menor familiaridad que la comunidad de desarrolladores de sistemas embebidos tiene con esos lenguajes frente a lenguajes como C o C++, lo que dificulta una adopción rápida y poco costosa, y la falta de metodologías en su empleo para la especificación de sistemas embebidos.

Esta situación ha favorecido la aparición de diferentes propuestas en torno al lenguaje C. Éste es un lenguaje más familiar para los desarrolladores de sistemas embebidos, e incluso para algunos desarrolladores de hardware que cotejan sus modelos

[2] El concepto de *media* guarda cierta similitud con el de *canal* de SystemC

VHDL o Verilog con modelos iniciales escritos en C o C++. Existen propuestas como *Embedded-C++* **[Pla97]** que define un subconjunto de C++, de forma que haga más portable y eficiente el uso de este lenguaje en el desarrollo de software embebido. Por tanto, esta propuesta se centra en el dominio software. *Handel-C* **[Pag96]** toma un subconjunto de C y lo extiende para soportar concurrencia y tipos de datos propios del dominio hardware, de forma que las descripciones, además de compilables, se puedan implementar en FPGAs o en ASICs. Una especificación Handel-C se ciñe al MoC CSP **[Hoa97]**. Por tanto, no es un lenguaje orientado al soporte de heterogeneidad. *SpecC* **[GZD00]** es un lenguaje de especificación de sistema basado en C, que soporta también la entrada de una metodología de diseño electrónico de nivel de sistema (ESL). De hecho, la separación entre el cómputo y la comunicación a través de canales, una noción fundamental de SystemC, se ha tomado de este lenguaje **[GLM02]**. Por otro lado SpecC difiere un tanto de SystemC en cuanto a que no se constituye como una librería de C++, sino que es una extensión que precisa de un compilador, que transforma la especificación SpecC en código C++ compilable, y un simulador específico. Existe un compilador de referencia de SpecC (SCRC) de fuente abierta y libre distribución y un manual de referencia **[SPC08]**.

### 2.2.4 Requisitos de una Metodología de Especificación Heterogénea

Esta sección resume las principales características que debe proporcionar una metodología de especificación heterogénea. Esto permitirá posteriormente analizar las limitaciones de las aproximaciones que ya han sido presentadas, motivar el uso del lenguaje SystemC y analizar hasta que punto éste lenguaje necesita ser complementado y adaptado para tal propósito.

Una metodología de especificación heterogénea debería cubrir una serie de necesidades básicas. En este trabajo se han identificado las siguientes:

1. Debe estar basada en un lenguaje suficientemente expresivo, compacto y con una semántica precisa, entendida sin ambigüedades por la comunidad de diseño.
2. El lenguaje debe ser adecuado para el soporte de especificación heterogénea. Es decir, el lenguaje debe proveer (o, al menos, ser lo suficientemente flexible para soportar una extensión eficiente) las facilidades de especificación que requieren los MoCs involucrados en los distintos dominios de diseño. Así, el lenguaje será más fácilmente admitido por las diversas comunidades de diseño.
3. Unas guías y reglas de modelado que permitan al usuario saber cómo usar las facilidades del lenguaje para especificar bajo un MoC y para integrar coherentemente en la misma especificación MoCs diferentes.
4. Medios para obligar a la especificación a cumplir las reglas del MoC o, en su caso, para la detección de su no cumplimiento.
5. Un marco formal que de fiabilidad y seguridad a la metodología de especificación.

Los dos primeros puntos se refieren directamente a la motivación que impulsa, en este trabajo, a optar por el lenguaje SystemC como alternativa a las metodologías presentadas en las secciones anteriores. El diseño del SoC exige el desarrollo de

especificaciones heterogéneas que permitan modelar, analizar e implementar todo un sistema antes de llegar a la implementación. Los trabajos presentados en las secciones anteriores (Ptolemy, *Metropolis*, Implementación de ForSyDe en Haskell) proveen un marco de especificación heterogéneo. Sin embargo, están basados en lenguajes que o bien juegan un papel secundario y son usados de una forma complicada (como Java en Ptolemy) o bien tienen alcance limitado en la comunidad de diseñadores (como Haskell en ForSyDe). La carencia de un lenguaje de especificación unificado fue identificada como uno de los principales obstáculos en el diseño de SoCs **[Gepp00]**. En ese contexto, SystemC, por diversas razones (ver sección 1.2) se ha convertido en uno de los lenguajes más aceptados para la especificación de nivel de sistema. El lenguaje es ya un estándar **[IEEE05]**, tiene una sintaxis cercana a C/C++ y, como se justificará más adelante, es lo suficientemente general y flexible para soportar una metodología de especificación heterogénea. No obstante, el lenguaje presenta aún ciertas carencias que motivan la metodología propuesta en este capítulo.

La adopción de una metodología de especificación heterogénea va más allá de la selección de un lenguaje adecuado. Los puntos 2-5 evidencian la necesidad de la metodología propuesta en este trabajo. El punto 2, considera el grado de completitud del lenguaje que permita abordar cada tipo de modelo, pero también tiene que ver con cómo debe ser la arquitectura del lenguaje para que soporte múltiples MoCs. La filosofía de SystemC es la de mantener un núcleo compacto y estándar y permitir la construcción de cada metodología específica a su alrededor. El tercer punto incide en el aspecto metodológico que tiene un MoC y como se debe plasmar en el lenguaje. El punto 4 incide en la necesidad de unas reglas de especificación y de que el diseñador sepa, al menos, si las está cumpliendo. El quinto punto, significa que dado que la metodología se basa en modelos con una base formal, entonces, sólo es preciso, comprender y demostrar la validez de la forma en la que los MoCs soportados se mapean en SystemC. De esa forma, quedaría formalizada la metodología. Este es un trabajo teórico de mayor calado, en curso, y cuya completitud se escapa de los objetivos de este trabajo de tesis. Este trabajo se centra en los puntos 2, 3 y 4 y provee avances en el quinto, de forma que futuros trabajos prometen culminar ese objetivo.

Antes de pasar a describir la metodología desarrollada y cómo ésta resuelve esos puntos, en la sección siguiente se repasan las actuales capacidades para especificación heterogénea de SystemC y de otras librerías metodológicas asociadas.

## 2.2.5 SystemC Para Especificación Heterogénea

### *2.2.5.1 Limitaciones de SystemC para Especificación Heterogénea*

Desde sus primeras versiones, SystemC ha provisto elementos para el soporte del modelado de hardware en un nivel de transferencia de registros (RTL) y en un nivel de comportamiento **[SYN02] [OSCI05]**. La base fundamental de este soporte es el kernel de simulación de eventos discretos y la provisión de una primitiva de tipo señal (*sc_signal*), con una semántica de comunicación de procesos de actualización y muestreo típica del diseño de hardware. Esto ha facilitado utilización del mismo lenguaje de programación de alto nivel, como C/C++, con una descripción HDL. Sin embargo, en SystemC no está definida una metodología estándar que explique cómo

especificar bajo distintos MoCs y cómo integrarlos en la misma especificación, considerando las implicaciones semánticas que ello tiene sobre el sistema (soporte de heterogeneidad horizontal). Más aún, se precisa una definición de un punto de partida abstracto del que parta de la especificación de nivel de sistema, y de unos pasos de refinado automáticos y con una base formal (soporte de heterogeneidad vertical).

Desde la versión 2.0, el lenguaje ha provisto nuevas facilidades de especificación de mayor nivel de abstracción. Además de algunos cambios en el modelo del tiempo, SystemC generalizó la infraestructura de comunicación de procesos, introduciendo el concepto de canal. De este modo, la señal pasó a ser un tipo particular de canal primitivo (el canal *sc_signal*). Además, SystemC provee ahora un conjunto de primitivas de comunicación predefinidas y denominadas canales primitivos estándar. Entre esos canales se encuentran, además de la señal (*sc_signal*), el búfer (*sc_buffer*), el mutex (*sc_mutex*), el semáforo (*sc_semaphore*), la cola fifo (*sc_fifo*) y la cola de eventos (*sc_event_queue*). Esta ampliación de la infraestructura de comunicación, mejoró el soporte de SystemC para especificación heterogénea. Supóngase ahora, por ejemplo, que se pretende escribir una especificación siguiendo un modelo de red de procesos comunicados por medio de canales fifo bloqueantes **[LePa95]**. Una representación típica de esa red de procesos se da en la Figura 2-18.

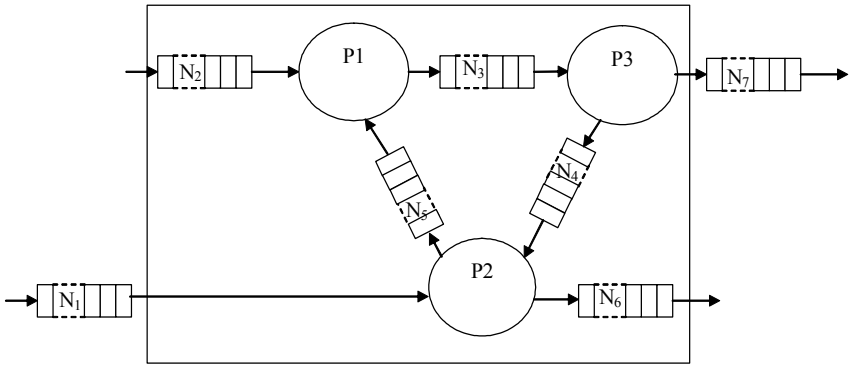



**Figura 2-18. Red de procesos comunicados con fifos finitas bloqueantes.**

En la Figura 2-18, se representan los procesos como círculos. Esos procesos se comunican mediante colas FIFO de *unidades de datos*. Estas colas o búferes tienen un tamaño de almacenamiento limitado y permiten una transferencia unidireccional de datos en la que tanto los accesos de lectura como los de escritura son bloqueantes. Las condiciones de bloqueo del proceso que accede al búfer son el estado de llenado o vaciado del búfer. Para deducir el comportamiento de esta especificación, además de la descripción de la conexión de los componentes primitivos (por ejemplo, a través de la representación gráfica de la Figura 2-18), es preciso también precisar aspectos de la semántica de relación y ejecución de esos elementos.

Si de esta descripción de una red de procesos, se quiere obtener una especificación equivalente y sin ambigüedades en SystemC por medio de los elementos primitivos del lenguaje, el primer problema es el cómo. Es decir, se precisan unas guías de especificación que informen al diseñador qué facilidades de especificación tiene disponible y cómo usarlas para plasmar el modelo bajo ese MoC. Este es uno de los aspectos más importantes tratados por la metodología propuesta en este trabajo.

Otro problema es que SystemC tampoco define una representación gráfica asociada a los elementos del lenguaje. Tal representación facilitaría el desarrollo y documentación de las especificaciones y serviría de base para las herramientas de captura gráfica. También daría cierta homogeneidad a la representación gráfica de una especificación heterogénea. La metodología *HetSC* también trata este punto. La Figura 2-31 muestra más adelante la representación gráfica introducida por la metodología *HetSC* para las primitivas básicas de SystemC. Esta representación se utilizará también en esta sección para mostrar otras carencias de SystemC para el modelado heterogéneo.

Un problema adicional aparece cuando el conjunto de primitivas de SystemC no es suficiente para soportar un MoC concreto. Por ejemplo, en la Figura 2-19, se representa una red de procesos comunicada a través de canales *rendez-vous*. Estos son canales que tienen una semántica de comunicación que no coincide con la de ninguno de los canales estándar de SystemC.

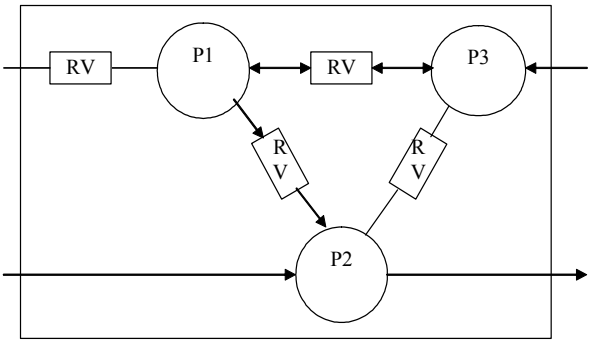



**Figura 2-19. Red de procesos comunicados por canales rendez-vous.**

Si se analizan otros MoCs, tales como KPN, SDF o Redes de Petri, se echarán igualmente de menos primitivas de comunicación en SystemC amoldadas a la sintaxis y semántica necesarias en esos casos. Por ejemplo, mediante canales *sc_fifo* y algunas restricciones adicionales en el empleo del código SystemC es posible construir grafos SDF. Sin embargo, el empleo de un canal *sc_fifo* para representar la semántica de un arco SDF es de muy bajo nivel, no conduce al usuario a respetar la estructura de tasas fijas de SDF. La metodología *HetSC* resuelve este problema proveyendo nuevas facilidades de especificación con una sintaxis y una semántica compacta y adecuada a los requisitos del MoC.

También se dan casos en los que, aunque SystemC provee facilidades de especificación con una semántica y la sintaxis adecuadas para un MoC, sus características pueden ser aún mejoradas para asegurar otros requisitos del MoC. Por ejemplo, el usuario se puede dar cuenta de que cada bola o proceso del ejemplo de la Figura 2-18 se puede describir como un proceso SC_THREAD de SystemC y que el búfer finito y de acceso bloqueante se puede implementar con un canal estándar *sc_fifo*. También puede darse cuenta de que el empleo de elementos de partición modular jerárquica (módulos y puertos), no afecta al modelo y se pueden emplear flexiblemente. De este modo, se puede escribir una especificación SystemC (mostrada en la Figura 2-20 haciendo uso de la representación gráfica de *HetSC)* que sólo emplea facilidades de la librería SystemC estándar. Para que sea posible afirmar que esa especificación SystemC representa un MoC de red de procesos es necesario además una equivalencia

semántica. Es decir, la semántica de simulación asumida en las primitivas de la Figura 2-18 debe mantenerse en la Figura 2-20.

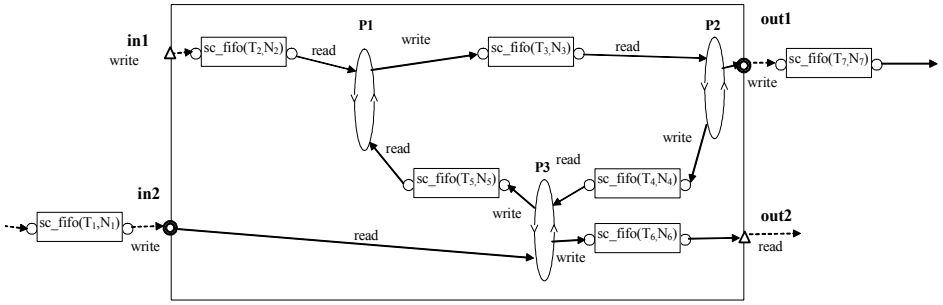



**Figura 2-20. Componente bajo MoC PN en SystemC.**

En este ejemplo, también se da esa equivalencia semántica. El canal *sc_fifo* de SystemC de la Figura 2-20 es un canal que permite en un extremo la escritura de unidades de datos bloqueante, en tanto que en el otro, la lectura bloqueante de esos unidades de datos. Esa fifo mantiene el orden y tiene un tamaño limitado y parametrizable. Pese a que internamente la implementación del canal *sc_fifo* fija en mayor detalle la situación de cada evento en el eje temporal, en el nivel de usuario no hay necesidad de manejar una información temporal más allá de que el orden de la secuencia de datos en la entrada se mantiene en la salida del canal. Todo esto, configura la semántica de una red de procesos de Kahn con colas limitadas, representada de forma abstracta en la Figura 2-18 y en términos de facilidades de especificación de SystemC en la Figura 2-20.

Sin embargo, quedan aún cuestiones abiertas acerca del grado de flexibilidad para codificar la especificación y si cualquier tipo de estructura es posible. La metodología de especificación debe resolver esas dudas mediante la adopción de la disciplina de un MoC. Por ejemplo, en la Figura 2-21, se presenta un modelo ligeramente modificado respecto al de la Figura 2-20 que presenta cuatro situaciones problemáticas en un MoC de redes de procesos.

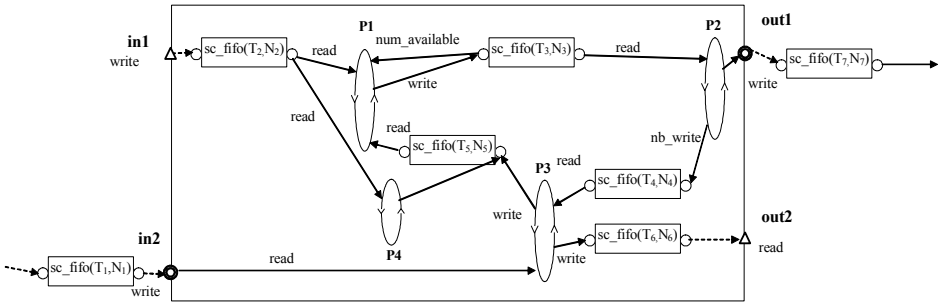



**Figura 2-21. Situaciones problemáticas en un MoC PN.**

En concreto, estas cuatro situaciones son los dos accesos de sólo lectura de los procesos P1 y P4 a la misma instancia de canal fifo; los dos accesos de escritura a la misma instancia de canal fifo por parte de los procesos P4 y P3; la lectura por parte del proceso P1 del número de unidades de datos de un canal y, finalmente, el acceso de escritura no bloqueante realizado por el proceso P2. Estas cuatro situaciones producen un indeterminismo potencial en la especificación y también pueden conducir a

situaciones de interbloqueo no esperadas. Estas situaciones son indeseadas en muchos casos e inaceptables en la mayoría de los sistemas de tiempo real. Las reglas de ciertos MoCs sirven para eliminar estas situaciones peligrosas.

Las reglas del MoC se deben traducir en una serie de reglas de especificación, que mejoran el uso del lenguaje, informando qué facilidades de especificación se deben usar y cómo se deben usar y componer con otras en la especificación. No se ha encontrado hasta la fecha una documentación de este tipo para el lenguaje SystemC. Por tanto, las guías y reglas de especificación son parte íntegra de la metodología *HetSC*.

Aunque se establezcan las reglas, una especificación SystemC como la de la Figura 2-21 compilará y ejecutará sin reportar ningún tipo de error al usuario. Tan importante como establecer unas reglas de especificación es fomentar y asegurar en lo posible su cumplimiento. Así, una metodología que tenga en cuenta los objetivos de un MoC y pretenda obtener sus beneficios debería, al menos, hacer saber al usuario que existe algún problema potencial con la especificación que ha descrito e incluso impedirle seguir más allá con la simulación mientras no garantice el cumplimiento de las reglas del MoC. La metodología *HetSC* provee soluciones en este sentido por medio de reglas que fuerzan el cumplimiento de otras y de código y facilidades de chequeo de reglas de MoCs.

Otra carencia en el soporte de MoCs en el lenguaje SystemC es respecto a la disponibilidad de informes o análisis relacionados con el MoC. Por ejemplo, sería interesante saber en qué punto se queda bloqueado cada proceso de una red cuando se da una situación de interbloqueo, o bien cuál ha sido el tamaño máximo de unidades de datos almacenados en una cola de una red de procesos de Kahn. Ambas son informaciones útiles para un posterior depurado y refinado del sistema respectivamente.

Finalmente, se puede llegar a la conclusión de que ciertos MoCs, como el MoC KPN, son imposibles de implementar en SystemC y, de hecho, en ningún lenguaje computable. La razón es que un modelo como KPN es ideal debido al tamaño de almacenamiento infinito de los canales, algo que ninguna plataforma de simulación o implementación puede ofrecer. No obstante, puede haber aproximaciones más cercanas a la semántica y sintaxis de canal infinito que la que proporciona el canal *sc_fifo* de la librería SystemC. La metodología *HetSC* también da soluciones en este sentido.

#### *2.2.5.2 Posibilidad de Extensión de SystemC*

Las limitaciones de SystemC puestas de manifiesto en la sección anterior no significan que SystemC sea un lenguaje inadecuado para una metodología de especificación heterogénea. Como se ha visto, SystemC ya proporciona un conjunto de primitivas de comunicación estándar que incluyen la señal, el mutex, etc, y que reflejan que se puede emplear el lenguaje para modelos de distinto nivel de abstracción. Sin embargo, en lugar de proporcionar un conjunto amplio y exhaustivo de primitivas, SystemC ofrece una arquitectura escalable del lenguaje. Esta escalabilidad se da gracias a las facilidades que el lenguaje provee para su extensión.

En la especificación funcional de SystemC **[SCFS01]**, se reconoce que SystemC puede soportar muchos MoCs y metodologías de diseño, pero también que las librerías y modelos provistos para el soporte de metodologías de diseño específicas (lo que

incluye a las metodologías de especificación) se consideran separadas del núcleo del lenguaje. Por esa razón, SystemC provee un núcleo genérico, pequeño y estándar.

Para la extensión de SystemC, existen facilidades como la interfaz, el canal primitivo y el evento, que permiten la adición de nuevas facilidades de especificación, especialmente en cuanto a nuevas primitivas de comunicación, es decir, nuevos canales. En los modelos concurrentes, gran parte de la peculiaridad de un MoC radica en la semántica de comunicación entre procesos. Este hecho se explotaba en *Metropolis*, donde la semántica de comunicación se concentra en primitivas específicas, los *media*. En SystemC, esto se hace en los canales. De esta forma, la creación y soporte de nuevos canales puede dar lugar al soporte de nuevos MoCs en SystemC. Estos canales pueden manejar además el dominio del tiempo de forma más abstracta. Además, las nuevas facilidades se pueden construir sobre el núcleo de SystemC y concentrar en una librería metodológica, sin que las extensiones supongan una extensión del núcleo del lenguaje.

De este modo, SystemC puede ser fácilmente extendido para que soporte una metodología de especificación heterogénea, que soporte la especificación bajo MoCs diferentes, tal y como se sugiere en **[GLM02]**. El MoC subyacente del lenguaje (o MoC base), viene determinado por las primitivas básicas de cómputo, sincronización y comunicación de SystemC (procesos, eventos y variables compartidas) y por su semántica de simulación. Así, el MoC base de SystemC es un MoC de eventos discretos (DE) de tiempo estricto. Sobre este MoC base, otros MoCs más abstractos o específicos pueden ser limpia y eficazmente soportados. Esto se representa en la Figura 2-22, donde el kernel se ha representado de forma simplificada respecto a la Figura 1-6.

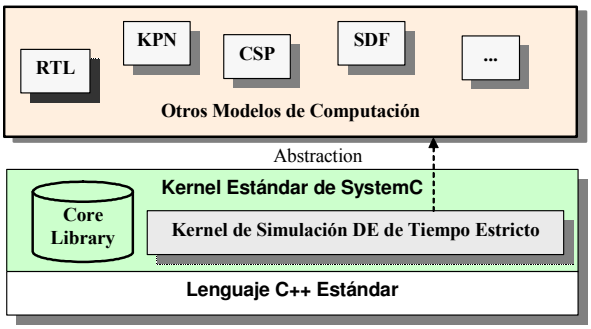



**Figura 2-22. SystemC tiene un MoC suficientemente general para soportar otros MoCs**

El modelo de computación abstracto puede incluir concurrencia, comunicación abstracta (en forma de canales) y un manejo más abstracto de la información temporal. Sin embargo, por debajo, el modelo de computación de la máquina de simulación de SystemC sigue siendo un modelo de Eventos Discretos (DE) de tiempo estricto. En este modelo, los procesos se sincronizan a través de eventos SystemC y transfieren datos a través de variable compartida.

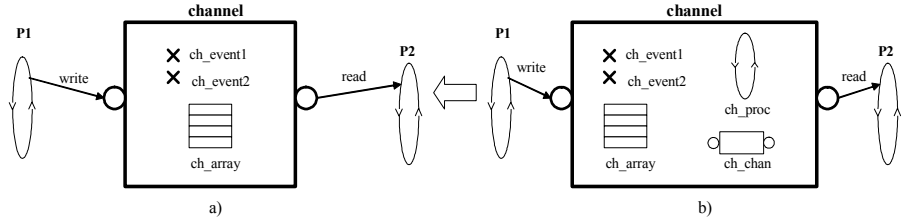


**Figura 2-23. La semántica de comunicación se abstrae como un canal en SystemC.**

Por ejemplo, en la Figura 2-23a se representa la implementación de cualquier canal primitivo SystemC. En general, puede contener solo eventos y variables. Incluso si se considera el canal más general de SystemC, es decir, un canal jerárquico (Figura 2-23b), que puede contener procesos, al final se puede generar un modelo semánticamente equivalente, que elimina los elementos jerárquicos intermedios, y que conste sólo de eventos, variables compartidas y procesos. La información temporal asociada al evento es concreta, una etiqueta temporal. Esta etiqueta es un par (t,δ), donde t es tiempo SystemC (*sc_time*), que representa tiempo físico, en tanto que δ es el ciclo delta de simulación dentro de un valor de tiempo SystemC. Desde fuera del canal, toda esta información no tiene porqué tener que ser tenida en cuenta. Hay una abstracción de esa información que refleja la abstracción de MoCs sobre el MoC DE de tiempo estricto.

La extensibilidad del lenguaje no se restringe a los canales. Por ejemplo, SystemC permite generar nuevos puertos que contengan las interfaces específicas utilizadas en cada modelo. Además, se puede introducir en ellos código de chequeo estático (ejecutado en tiempo de elaboración) para verificar reglas de conexión de módulos.

Pese a que en **[GLM02]** se menciona la posibilidad de extensión de SystemC para el soporte de diversos MoCs, esa posibilidad no se desarrolla en ese trabajo. Existen, sin embargo, algunos trabajos que mejoran la capacidad de abstracción y proveen nuevas primitivas con contenido semántico diferente al que proveen las primitivas estándar de SystemC. En las siguientes secciones se hace un repaso de esos trabajos y se analiza cómo contribuyen a la especificación heterogénea en SystemC.

#### *2.2.5.3 Librería TLM*

El modelado de nivel de transacción o TLM es una nueva aproximación (probada a nivel industrial) al modelado de plataforma HW que mejora la productividad en un entorno de codiseño HW/SW **[Ghe05]**. El modelado TLM eleva el nivel de abstracción en varios aspectos de la especificación de la plataforma reduciendo el tiempo de simulación y permitiendo el desarrollo y chequeo del SW embebido en etapas más tempranas, lo que permite reducir el tiempo a mercado.

SystemC ha sido el lenguaje precursor por excelencia del modelado TLM. Una de las librerías metodológicas basadas en SystemC más relevantes es la librería TLM **[RSPF05]**. Desde la primera versión (TLM 1.0) hasta el reciente estándar OSCI TLM 2.0 **[OSCI08]**, se ha presentado al conjunto de interfaces TLM, denominadas *interfaces núcleo,* como la parte más importante de la librería TLM. TLM 1.0 introdujo las interfaces unidireccionales bloqueantes (*tlm_blocking_get_if<T>, tlm_blocking_peek_if<T>* y *tlm_blocking_put_if<T>*) y no bloqueantes (*tlm_nonblocking_get_if<T>, tlm_nonblocking_peek_if<T>, tlm_nonblocking_put_if<T>)*, la bloqueante bidireccional (*tlm_transport_if<REQ,RSP>*) y las Maestro/Esclavo (*tlm_master_if<REQ,RSP>, tlm_slave_if<REQ,RSP>*). A éstas, TLM 2.0 ha añadido las interfaces *transport* bloqueante (*tlm_blocking_transport_if<TRANS>*) y *transport* no bloqueantes

(*tlm_fw_nonblocking_transport_if<TRANS>,tlm_bw_nonblocking_transport_if<TRANS>*), las interfaces de depurado y las de acceso directo a memoria (DMI).

TLM 1.0 aportó también algunas facilidades de especificación, como los canales *tlm_fifo*, *tlm_req_rsp_channel* y *tlm_transport_channel*, que enriquecen la capacidad de elección en semánticas de comunicación. TLM 2.0 ha definido además otro conjunto de elementos como los módulos *Iniciadores*, *Objetivo*, *Puente*, *Componentes Interconectores* y *los conectores iniciadores y objetivo*. TLM 2.0 también define una *carga útil genérica*, que soporta el modelado abstracto de buses mapeados en memoria, lo que junto a elementos como las interfaces DMI y a la propia documentación TLM, han ratificado al modelado de plataforma y el desarrollo de componentes y modelos TLM interoperables como objetivos principales de TLM. De hecho, en TLM 2.0 se define la *Interoperabilidad TLM* como la capacidad de conectar modelos directamente y que puedan intercambiar transacciones sin modificación de código.

Por lo tanto, atendiendo a estos objetivos, TLM es un trabajo distinto al que se presenta en este trabajo, esto es, el desarrollo de una metodología de especificación heterogénea en SystemC. El soporte de diversos MoCs y su conexión coherente es independiente de si el objetivo es desarrollar un modelo de aplicación, o de si, tal y como ocurre en los casos de uso TLM, se trata de un modelo del hardware de la plataforma. En ambos casos es preciso manejar aspectos funcionales, de concurrencia, de precisión en los tipos de datos, manejo del tiempo, etc y hay que tomar decisiones adaptadas al nivel de detalle requerido en cada parte del modelo.

Un modelo TLM supone ciertos niveles de detalle en el modelo. Recurriendo a los dominios de Rugby *Datos* y *Comunicación*, un aspecto distintivo de TLM es el tipo de datos transferido entre procesos: se da una transacción de una carga útil genérica. En el dominio temporal, TLM se ha enfocado en un rango intermedio entre los MoC atemporales y los MoCs síncronos de reloj, sin englobar ninguno de éstos. En **[OSCI08]**, se explicita que el modelado atemporal está fuera del ámbito de TLM 2.0. Se entiende que este *estilo* de modelado está cubierto con las primitivas aportadas por SystemC y TLM 1.0. Además, el objetivo de TLM es modelar un sistema basado en bus, cuyo modelado precisa alguna noción de tiempo estricto. Por otro lado, se pretende evitar los modelos síncronos de reloj, puesto que se pretende mejorar las prestaciones en velocidad de simulación de estos modelos. Más específicamente, en **[OSCI08]**, se definen dos *estilos de codificación* situados en ese rango intermedio: el *débilmente temporal* y el *aproximadamente temporal*. En ambos estilos se da una noción de tiempo estricto a las transacciones realizadas entre módulos. En el estilo débilmente temporal las transacciones se definen por dos puntos temporales, en tanto que en el estilo aproximadamente temporal por cuatro.

Esta mayor definición en cuanto a los posibles niveles de abstracción manejados, a los beneficios y contraprestaciones obtenidas y la disponibilidad de un conjunto más rico de facilidades de especificación hacen que TLM suponga también un avance en la capacidad de SystemC para el modelado heterogéneo. Por ejemplo, el canal *tlm_fifo* provee una semántica más general que el canal *sc_fifo* del núcleo de SystemC. Es posible declarar canales de tamaño infinito (o, de forma más precisa, indefinido y automáticamente redimensionable). De este modo, mediante instancias de canal

*tlm_fifo*, es posible realizar una especificación SystemC-TLM (Figura 2-24) más aproximada a los supuestos de cola infinita del MoC de redes de Kahn.

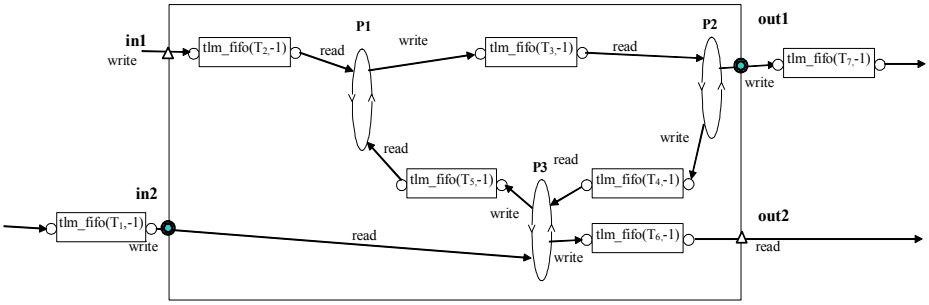



**Figura 2-24. Implementación de una red de Kahn con canales tlm_fifo.**

Sin embargo, TLM no aborda el aspecto metodológico y formal que se pretende en este trabajo. La propia documentación TLM habla de *estilos de codificación* en lugar de una metodología de especificación, y no explica como realizar una especificación bajo el MoC KPN. De hecho, como se ha comentado, un MoC atemporal, como KPN está fuera del ámbito de SystemC TLM 2.0.

Sin embargo, como se presenta y reseña es este trabajo, tanto en los MoC atemporales, como en los que aborda TLM es preciso restar ambigüedades semánticas, clarificar qué reglas de especificación seguir, y otorgar una base formal a la metodología de especificación. Es la diferencia entre una metodología de especificación y un estilo de codificación. Esta aseveración se puede sostener desde otros aspectos. Por ejemplo, en cuanto al dominio *Comunicación*, las interfaces TLM no establecen totalmente la semántica del canal que las implementa. De hecho, estrictamente hablando, en SystemC, una interfaz no implica semántica de comunicación alguna, sino que ésta la define el canal. No obstante, en TLM se asume que el uso de una interfaz núcleo supone ciertas implicaciones semánticas, lo que, en consecuencia, fija parcialmente la semántica del canal. Por ejemplo, si un canal implementa una interfaz unidireccional de escritura bloqueante (*tlm_blocking_put_if*), la semántica de acceso de escritura a dicho canal (a través del método *put*) determina que la transferencia de datos será en un sólo sentido, desde el proceso al canal, y que debe existir al menos una condición en la que el proceso que accede se bloquea. Sin embargo, el uso de la interfaz *tlm_blocking_put_if* no establece si hay una o más condiciones de bloqueo, cuál es o son la(s) condicion(es) en la(s) que se produce ese bloqueo, etc. Por tanto, puede existir más de una implementación de canal (en principio, infinitas) que cumplan la semántica requerida para dicha interfaz TLM.

En contraste, una metodología de especificación heterogénea exige una semántica de comunicación más precisa para la mayoría de los MoCs. Cuál es la condición de bloqueo no concretada por la interfaz TLM es precisamente un factor distintivo primordial entre algunos MoCs atemporales. Además de las condiciones de bloqueo, existen otras características que definen una semántica de comunicación y que no están especificadas por las interfaces TLM. Entre otras, se pueden enumerar el manejo de la información temporal en el canal, el tipo de dato transferido, la capacidad de almacenamiento del canal, etc. En definitiva, las interfaces TLM no proporcionan una semántica de comunicación que satisfaga el punto 2 de la sección 2.2.4.

Por otro lado, TLM define reglas de especificación y chequeadores de esas reglas. Por ejemplo, **[OSCI08]** incluye reglas que gobiernan cómo se mezclan y ordenan las llamadas *transport* bloqueantes y no bloqueantes al mismo módulo, o que dictan que un módulo iniciador puede reutilizar un *objeto transacción* desde una llamada a la siguiente y a través de llamadas a las interfaces *transport*, DMI y de depurado. Sin embargo, estas reglas y chequeadores responden a objetivos fundamentalmente de interoperabilidad, distintos a los que tienen los MoCs tratados en este trabajo. Por ejemplo, retomando el caso de la red de Kahn con facilidades TLM, el canal *tlm_fifo* implementa dos interfaces bloqueantes (*tlm_bloking_put_if<T>* y *tlm_blocking_get_if<T>*), pero también dos no bloqueantes (*tlm_bloking_put_if<T>* y *tlm_blocking_get_if<T>*). Los accesos no bloqueantes no están permitidos en un MoC KPN. Por tanto, para garantizar que un modelo cumple las reglas del MoC KPN sería necesario informar previamente al usuario que eso no es correcto. También sería conveniente chequear que esos métodos de acceso no son usados, o bien contar con un canal que no los implemente y, por tanto, impidan su uso por construcción. Existen más reglas que cumplir en el MoC KPN. Por ejemplo, no más de un proceso debe acceder como escritor a la misma instancia de canal FIFO. Sin embargo, una especificación como la de la Figura 2-25, que incluye accesos no bloqueantes desde más de un proceso escritor a una instancia de canal fifo TLM, compila y ejecuta sin dar un aviso o un error. Existen más reglas aún que debería cumplir la especificación SystemC para atenerse a un MoC KPN y que no observadas por la metodología y las facilidades TLM estándar. Por lo tanto, los puntos 3 y 4 de la sección 2.2.4 no se cumplen. Una conclusión parecida se puede obtener cuando se analizan otros MoCs, tales como SDF, CSP, etc.

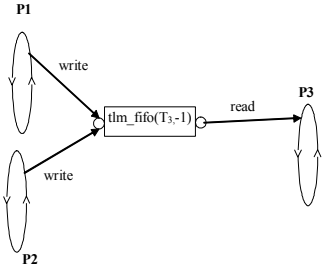



**Figura 2-25. Un canal tlm_fifo no comprueba el acceso de los procesos escritores.**

En resumen, se puede decir que, si bien TLM, con unos objetivos concretos (*casos de uso* de TLM 2.0), ha aportado de forma indirecta un avance en el soporte de heterogeneidad, no ofrece el soporte (facilidades de especificación, reglas de diseño, chequeadores) que requiere una metodología de especificación heterogénea. No obstante, TLM puede tanto guiar el avance como beneficiarse de los resultados de este trabajo. Por ejemplo, una de las claves de interoperabilidad en TLM es el soporte de *Interfaces y Conectores Combinados*, que soporten la conexión de componentes descritos en distintos niveles de abstracción. Sin duda, eso exigirá determinar qué tipo de modelo heterogéneo se está generando y qué reglas determinan la posibilidad de esa conexión y la validez y coherencia de los resultados ofrecidos por ese modelo.

### *2.2.5.4 Librería Master-Slave*

La librería Maestro-Esclavo o MS **[MSCL02]** provee unas reglas de modelado y una serie de primitivas específicas que permiten la construcción de una especificación consistente en una serie de cadenas de llamada a procedimiento remoto (cadenas RPC). De este modo, se puede entender esta contribución a SystemC como una librería que cubre un MoC basado en el paradigma de comunicación Maestro-Esclavo.

El cómputo se realiza en procesos *Maestros* y *Esclavo*. Un proceso maestro invoca y transfiere datos a un módulo esclavo a través de una RPC. Una peculiaridad de la librería MS es que el tipo de proceso (maestro o esclavo) se termina de definir en tanto se asocia a un tipo de puerto (maestro o esclavo), pese a que, en general, el puerto es un elemento que se utiliza para definir la estructura modular de la especificación. La RPC parte desde el puerto maestro, asociado a un proceso maestro, y se transfiere a un puerto esclavo, que tiene asociado un proceso esclavo. El proceso esclavo es básicamente una función que toma como parámetros los recibidos por el puerto esclavo y devuelve los parámetros de retorno por la RPC al proceso maestro. Usualmente, aunque no necesariamente, el proceso maestro y el proceso esclavo se encuentran en distintos módulos. En el caso de que se encontraran en el mismo, la comunicación precisaría ir por fuera del módulo. Las escrituras a un puerto maestro pueden realizarse desde un hilo SystemC, por ejemplo, de tipo SC_THREAD, o bien desde un proceso esclavo. Gracias a esto último, es posible encadenar llamadas RPC.

La característica principal de este modelo es que, al estar la funcionalidad del maestro y la del esclavo secuencializadas, no se exigen hilos distintos en el módulo maestro y en el esclavo. Por ejemplo, en el ejemplo de la Figura 2-26 existe un solo hilo de ejecución, cuyo contexto se reparte a lo largo de los módulos M1, M2 y M3. El proceso P1, en M1, es un proceso "independiente" o disparador, de cualquiera de los tipos de proceso de SystemC, en tanto que los procesos P2 y P3, en M2 y M3 respectivamente, son procesos esclavos, asociados a sus correspondientes puertos esclavos, y cuyo disparo se da con la llamada RPC (las dos bidireccionales en el ejemplo). Notar la representación gráfica propia de esta metodología.

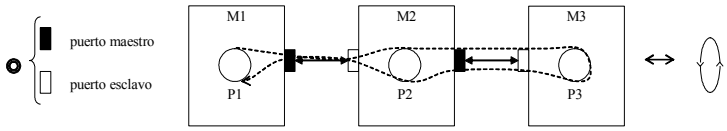



**Figura 2-26. Una cadena de llamadas RPC de la librería MS.**

Este estilo de modelado es muy eficiente en tiempo de simulación, característica esencial en el modelado de alto nivel de plataformas HW. De hecho, se puede entender que la librería MS es un primer antecedente de TLM. En la versión 2.0 de TLM se definen los niveles PV y PVT, es decir, de Vista del Programador (PV) y del Programador Temporal (PVT). En ellas se emplea una aproximación similar a la del modelo MS, pero añadiendo anotaciones temporales. Por otro lado, TLM está desarrollando su propia terminología y sintaxis estándar. Por ejemplo, los módulos *Maestro* y *Esclavo* son equiparables a los *Iniciador* y *Objetivo* de TLM 2.0.

La aproximación Maestro-Esclavo, bien realizada mediante la librería TLM-2.0, bien mediante la ya casi obsoleta librería MS, gana en velocidad de simulación, pero

puede resultar más compleja y menos natural en una especificación concurrente, especialmente si no se considera información temporal. Una razón es la no coincidencia entre la partición de módulos y de procesos, como se vio en la Figura 2-26, y también en la Figura 2-27. Por otro lado, puede conducir a indeterminismos cuando, por ejemplo, se realiza un modelo atemporal en el que varios maestros acceden a un bus.

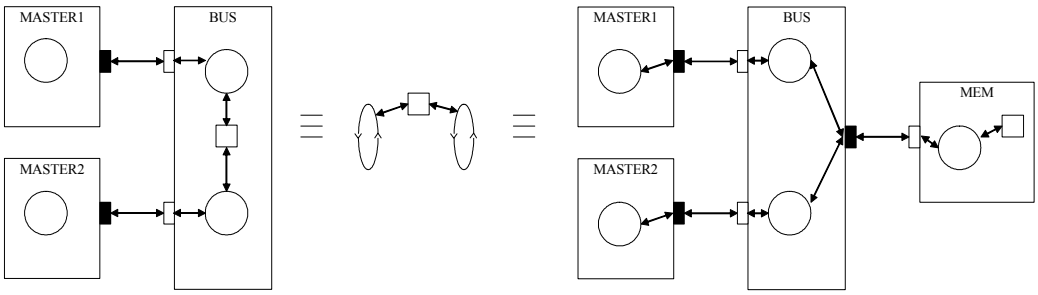



**Figura 2-27. Uso implícito de variable compartida en la especificación Maestro/Esclavo.**

Como se esquematiza en la Figura 2-27, en esos casos es fácil encontrar un punto de la especificación en el que se da una comunicación de procesos concurrentes por medio de variable compartida. En algunos casos, esto puede no ser relevante o incluso intencional, en cambio, en otros es necesario disponer de un MoC que garantice el determinismo y otras propiedades en la especificación.

#### *2.2.5.5 Extensiones de SystemC para MoC analógicos*

En las técnicas de especificación y simulación en el dominio del diseño analógico se está tendiendo hacia la abstracción y la especificación de nivel de sistema. Tradicionalmente, la parte analógica de los sistemas se ha modelado y simulado con herramientas como SPICE **[SPI08]** o Matlab **[HaLi01]**. Sin embargo, cuando se pretende modelar y simular un sistema con partes tanto digitales como analógicas, esto implica el uso de lenguajes diferentes, además de la conexión de los simuladores propios de cada entorno. Por esta razón, se han desarrollado extensiones de los HDLs clásicos, tales como VHDL-AMS **[VHA97]** y Verilog-AMS **[VEA00]**. Otra solución, consiste en la extensión de lenguajes de programación tales como C++, Java o UML.

La tendencia hacia el modelado basado en C++, ha impulsado la aparición de diferentes propuestas que extienden este lenguaje para el modelado analógico. En **[AlKa05][OBC06]** se puede encontrar un repaso detallado de estas. En **[AlKa05]**, se recalca que estas extensiones son en su mayoría específicas de aplicación y/o específica del nivel de abstracción. Como ejemplo del primer caso, en **[BHY01]** se propone una librería de módulos C++ para la simulación de convertidores analógico digitales (ADCs). Como ejemplo del segundo caso, en **[CCOB04]** se propone un marco en el que la implementación de los bloques analógicos se realiza mediante macromodelos analógicos. Estos son módulos compuestos de dos tipos de hilos, uno de cómputo y otro de activación. La irrupción de SystemC ha hecho que las propuestas se muevan hacia este lenguaje.

SystemC-A **[AlKa05][AlKa04]** propone una aproximación general para proveer extensiones analógicas en SystemC capaz de manejar un amplio rango de sistemas dinámicos no lineales. De este modo, SystemC-A es un superconjunto de SystemC que

provee soporte para varios niveles de abstracción, principalmente centrado en el modelado de más alto nivel. No obstante, también soporta descripciones de nivel de circuito. SystemC-A propone un solucionador analógico de propósito general acoplado con el núcleo de simulación de eventos discretos de SystemC por un algoritmo de sincronización pesimista (*lock-step*). Esta sincronización exige la modificación del núcleo de la librería SystemC.

SystemC-WMS **[SCW08] [OBC06]** mejora la aproximación en **[CCOB04]** de manera que habilita la especificación dentro del mismo modelo tanto de descripciones de bajo nivel como de alto nivel. La librería SystemC-WMS provee tanto componentes electrónicos básicos como la posibilidad de macromodelado de bloques analógicos. La propuesta básica es realizar la descripción de bloques analógicos por medio de las facilidades básicas de SystemC (módulos, canales, interfaces). De esta forma, las partes analógicas del sistema se modelan como módulos analógicos que se comunican intercambiando ondas de energía a través de interfaces de canales de ondas (*wavechannel*). De este modo, esta metodología permite la definición de un estándar de interfaz analógico. SystemC-WMS puede soportar tipos simples de ecuaciones diferenciales no lineales. En **[LCVA07]** se aplica SystemC-WMS para el estudio de diferentes técnicas de control de amplificadores conmutados, empleados en transmisores de RF polares. En sistemas como estos, es preciso usar modelos de bajo nivel de los convertidores *Buck* (mediante resistores, transistores MOSFET, etc), en tanto que las diferentes leyes de control se modelan mediante una descripción de alto nivel.

SystemC-AMS **[VGE04] [EGV05]** es la propuesta del grupo de trabajo para señal analógica y mixta de SystemC (*SystemC AMSWG*) **[AMSWG]**. Una característica importante de SystemC-AMS, al igual que otras librerías estándar y metodológicas de SystemC, es que está basada en una estructura de capas que se superpone a la librería núcleo de SystemC, sin exigir la modificación de ésta.

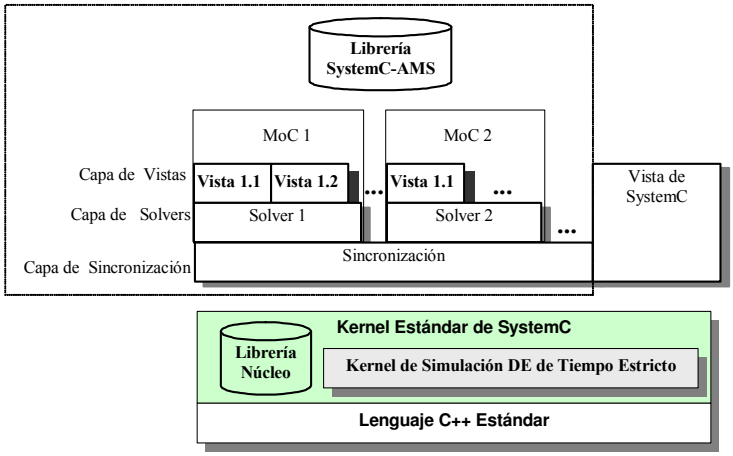



**Figura 2-28. Estructura General de SystemC-AMS.**

En la Figura 2-28 se muestra la estructura propuesta en **[EGVM02]**. Consta de tres capas:

- **Capa de Sincronización**. Es una capa común a los simuladores analógicos específicos (solucionadores), cada uno de ellos optimizado para un MoC

analógico concreto, que los sincroniza con el núcleo de simulación de SystemC. Dos restricciones básicas de esta capa simplifican la sincronización:

- La capa de sincronización solo incrementa tiempo. Dado que SystemC no guarda el estado de simulación, se evitan situaciones en las que es preciso una vuelta atrás (*backtracking*).
- Flujo de señal directo.

- **Capa de Solucionadores.** Es una capa en la que a cada MoC soportado se le provee un solucionador, es decir un código que resuelve las ecuaciones que rige el modelo. El solucionador provee la forma específica y óptima (en tiempo y recursos de la plataforma de simulación) de computar para ese MoC. Es capaz de procesar los datos de entrada (un sistema de ecuaciones provisto por la capa de vistas) y obtener los de salida con el nivel de detalle propio de ese MoC. Además cada solucionador es capaz de comunicarse con la capa de sincronización.
- **Capa de Vistas o de Descripción.** Es la capa que maneja el usuario y la que permite realizar las descripciones analógicas y conectarlas a otras descripciones SystemC. La capa de vistas genera el sistema de ecuaciones que se proporciona al solucionador correspondiente como entrada. Un solucionador tiene asociadas una o más vistas. En **[EGVM02]** se distingue al menos las dos siguientes:
  - **Interfaz de lista de nodos.** Permite dar una descripción más estructural del modelo analógico.
  - **Interfaz de Ecuaciones.** El usuario especifica por medio de ecuaciones diferenciales. Es una descripción más funcional del modelo analógico.

Como muestra la Figura 2-29, SystemC-AMS soporta actualmente varios MoCs: el Flujo de Datos Síncronos Temporal (T-SDF), que a su vez da soporte a otros como el MoCs de Redes Lineales o LN (*Linear Networks*) con varias vistas, una de ceros y polos (ZP) y otra de función de transferencia (TF), y las Redes Eléctricas Lineales (LEN), para la descripción de circuitos con componentes básicos como capacitores, resistencias, fuentes de tensión, de intensidad, etc.

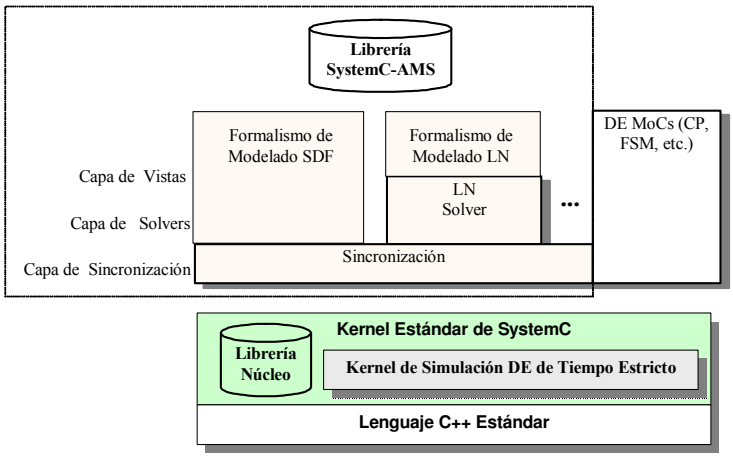



**Figura 2-29. Disposición de Capas Actual de la Librería SystemC-AMS.**

Para cada uno de estos MoCs, SystemC-AMS provee un solucionador específico. El más importante es el solucionador del MoC T-SDF. Este MoC es una variante del MoC SDF. El solucionador analiza los módulos AMS ligados a un *grupo* o *cluster*, a través de sus conexiones de entrada y salida. Por cada grupo genera un grafo SDF y obtiene su planificación estática (si es factible) y el periodo de grupo. De este modo, SystemC-AMS forzará un avance temporal de dicho periodo por cada planificación del grupo. El MoC T-SDF da soporte al resto de MoCs. Por ejemplo, el MoC LN cuenta con su propio solucionador que se activa y computa en cada periodo de grupo.

#### *2.2.5.6 Extensión del Kernel de SystemC: SystemC-H*

En **[PaSh04] [PaSh05]** se propone la extensión del kernel de SystemC para el soporte de especificación heterogénea y se presenta de una forma generalizada como realizar esa extensión. Se desarrollan una serie de nuevas facilidades (clases C++, macros, etc) que sustituyen a las provistas por el núcleo de SystemC para la descripción y simulación bajo un MoC específico, como el MoC SDF. Algunas de esas facilidades son públicas y proveen al especificador nuevas primitivas. Sin embargo, otras clases son transparentes al especificador y se usan para implementar nuevas máquinas de simulación adaptadas a su MoC correspondiente. De esta forma, esta aproximación propone un "kernel de de simulación heterogéneo" **[PaSh05],** que satisfaga las necesidades de velocidad de simulación de la especificación heterogénea. La idea es similar a la capa de solucionadores de SystemC-AMS.

En **[PaSh04] [PaSh05]** se provee un soporte específico para el MoC SDF. Para ese MoC se reportó una aceleración de la simulación de hasta un 75% con respecto a una especificación funcionalmente equivalente descrita bajo un MoC DE en SystemC. En **[PMS04]** se explora esta técnica para el soporte de otros MoCs, tales como el MoC CSP o el MoC de Máquinas de Estados Finitos o MoC FSM **[Jan04]**. En estos casos las ganancias en velocidad de simulación resultaron ser más limitadas. Por otro lado, las ganancias en velocidad de simulación reportadas en una especificación heterogénea DE-SDF se reducían al 13%, debido a la ley de Amdahl **[Amd67]**. Es decir, el impacto en las ganancia de velocidad simulación conseguidas en la parte SDF se minimizaba en el total debido a que el peso de cómputo antes de aplicar la optimización de esta técnica recaía fundamentalmente en la parte del MoC DE, que maneja un mayor nivel de detalle en el modelado del tiempo, tipos de datos, etc..

La disminución de la ganancia en la velocidad de simulación debido a la ley de Amdahl no es achacable a la técnica propuesta. Sin embargo, es preciso considerar la atenuación de esa ventaja en la especificación de sistemas heterogéneos y considerar la principal desventaja de la técnica propuesta. Ésta consiste en que la extensión de las capacidades de SystemC se hace a costa de modificar el núcleo estándar de SystemC y, además, hacerlo menos compacto. En efecto, las nuevas clases que soportan la sintaxis y la semántica de simulación específica del MoC soportado se incrustan en el núcleo. Esto refleja también que no se ha sacado provecho de las múltiples facilidades para la extensión de SystemC mencionadas en **[GLM02]**.

Es preciso considerar si en un entorno de especificación heterogénea la ganancia de simulación que puede aportar un núcleo extendido mediante nuevas máquinas de simulación o solucionadores compensa un mayor tamaño del nuevo núcleo, redundancia

en las primitivas de especificación, una complejidad adicional en la especificación, etc. Las extensiones analógicas están en mejores condiciones de justificar la extensión del núcleo de simulación y del lenguaje. El coste en tiempo de simulación de MoCs de tiempo continuo es lo suficientemente grande como para justificar la inclusión de nuevos solucionadores en el núcleo del lenguaje. Pese a ello, aproximaciones como SystemC-AMS lo hacen de forma desacoplada, mediante una librería metodológica que hace uso de las facilidades de extensión de SystemC. En cambio, MoCs más abstractos, pueden no justificar la inclusión de nuevos solucionadores y aún menos hacerlo extendiendo el núcleo de SystemC. Esta es otra de las ideas de fondo de la metodología de especificación que se presenta en este trabajo de tesis.

#### *2.2.5.7 SysteMoC*

SysteMoC **[FHT06]** es una metodología que se centra en proveer la capacidad de extracción y análisis del MoC empleado en la especificación SystemC. Esta capacidad se entiende como un requisito fundamental para el resto de las actividades de diseño. Para dotar la metodología de esa capacidad, la librería SysteMoC provee soporte para un MoC básico denominado *Funstate*. Las especificaciones escritas bajo este MoC expresan el comportamiento de la comunicación como máquinas de estados finitos (FSM). Esto posibilita la extracción y análisis del MoC empleado, tan solo analizando la FSM de comunicación junto con la topología de la especificación.

## 2.3 Introducción a la Metodología de Especificación HetSC

*HetSC* es una metodología de especificación de sistemas embebidos heterogéneos basada en el lenguaje SystemC. La metodología establece cómo especificar en SystemC las diferentes partes de un sistema empleando uno o varios MoCs. Cuando se emplea más de un MoC, *HetSC* también describe cómo realizar una conexión coherente de las diferentes partes a través de interfaces de MoCs.

### 2.3.1 Contribuciones

La metodología *HetSC* aporta ventajas en la especificación y modelado de sistemas embebidos hardware/software.

**Carácter Metodológico.** Propone una metodología de uso de las facilidades de especificación. El usuario no sólo necesita contar con un lenguaje de sintaxis y semántica bien definida, como SystemC, sino también saber cómo usarlo eficaz y eficientemente en cada dominio de diseño. De esta forma, la metodología provee:

- Una documentación con las reglas y guías de modelado con una base formal.
- Una librería metodológica (librería *HetSC*) que complementa las facilidades de especificación de SystemC.
- Elementos para asegurar y chequear las reglas de diseño.

**Unificación.** En *HetSC* se unifica el marco de especificación y el de simulación en un grado importante frente a otras metodologías. El lenguaje estándar SystemC constituye un marco común de especificación. Por otro lado, el kernel de simulación de eventos discretos y de tiempo estricto constituye un marco común de simulación.

**Semántica definida y estable.** La metodología se basa en un lenguaje normalizado (IEEE1666). Tanto los elementos del lenguaje SystemC como los nuevos elementos aportados en *HetSC* tienen una semántica definida que reduce la ambigüedad y, por tanto, mejoran la transferencia de los modelos.

**Costo de aprendizaje reducido.** La sintaxis empleada para especificar el cómputo concurrente es la del lenguaje C/C++. En **[Edw98]**, C y C++ se reportaban entre los tres lenguajes de programación más empleados en sistemas embebidos (con un 81% y un 39% respectivamente). En segundo lugar se encontraba el lenguaje ensamblador, con un 70%. Más recientemente, estos dos lenguajes ocupan las dos primeras posiciones (con porcentajes del 94% y 84%), en tanto que el uso del ensamblador cae notoriamente (33%) **[HKMB05]**. Por otro lado, la sintaxis de SystemC es también próxima a la de lenguajes de diseño hardware tales como Verilog.

**Estructura Eficiente.** La metodología propuesta y las herramientas que la soportan se integran de forma eficiente. La metodología utiliza, siempre que es posible, las facilidades de especificación provistas por la librería SystemC y otras librerías relacionadas. Cada nuevo elemento añadido por la librería *HetSC* resuelve alguna carencia en estas librerías (en alguno de los puntos reseñados en la sección 2.2.4).

**Modularidad.** La metodología no precisa la modificación de la librería estándar SystemC. Las facilidades de especificación y de chequeo adicionales se distribuyen

como una librería metodológica, la librería *HetSC,* y una documentación asociada, que complementan la librería SystemC y su documentación asociada.

**Compatibilidad y sinergia con estándares y metodologías relacionadas.** La metodología no precisa modificar otras librerías instaladas y basadas en SystemC. Aún más, la librería *HetSC* presenta compatibilidad TLM (sección 2.5.7) y puede interoperar con SystemC-AMS (sección 2.5.8).

**Orientación al diseño.** La metodología *HetSC* está diseñada para facilitar la aplicación de metodologías de perfilado e implementación. Específicamente, el GIM ha desarrollado metodologías de fuente única, las cuales, partiendo de la especificación *HetSC*, permiten realizar automáticamente la estimación de rendimiento o la implementación en forma de software embebido. Ésta última, es la metodología *SWGen*, explicada en el Capítulo 3 de esta tesis.

**Flexibilidad.** La metodología permite activar y desactivar ciertas reglas de especificación. De esta forma, el usuario experimentado puede, bajo su propio criterio y conveniencia, utilizar otros estilos de especificación no comprendidos entre los MoC actualmente soportados por *HetSC*.

**Extensibilidad.** La metodología y la librería metodológica se estructuran de forma que es factible y más sistemática la inclusión de nuevas facilidades de especificación y el soporte de nuevos MoCs.

**Soporte formal.** La metodología se inspiró inicialmente en un metamodelo **[LeSV98]** y su desarrollo se ha basado en varios trabajos en el campo de los modelos de computación. Las reglas, las facilidades de especificación y su semántica, así como la relación de beneficios obtenidos de cada modelo se han extraído de esos trabajos. La metodología *HetSC* surge de un estudio e interpretación de esos modelos y de una reflexión de cómo estos se pueden plasmar eficientemente en el lenguaje SystemC.

**Soporte de Heterogeneidad Eficiente.** Una contribución más teórica de este trabajo es la de hipotetizar que es posible implementar eficientemente sobre un núcleo de simulación basado en un MoC de eventos discretos y de tiempo estricto cualquier MoC que contenga el mismo o inferior nivel de detalle de información temporal, tales como los MoCs atemporales y síncronos. Esto se ha confirmado para el conjunto de los MoCs soportados. Más aún, se arguye cómo algunos de esos MoCs, como el KPN, que no permiten optimizaciones tales como la sustitución de planificación dinámica por estática, son soportados de una forma eficiente sobre el MoC DE de tiempo estricto. En el caso específico de SystemC, esto supone que no se precisa proveer máquinas de simulación específicas para su soporte. Por otro lado, la metodología se conecta con otras metodologías, como SystemC-AMS, que proveen solucionadores para MoCs analógicos, donde la extensión del marco de simulación es claramente beneficiosa.

### 2.3.2 Arquitectura de *HetSC*

En la Figura 2-30 se representa la arquitectura de *HetSC*. La metodología presenta dos niveles principales.

En un primer nivel, la *Metodología General de Especificación* de *HetSC* define una serie de reglas y guías de especificación básicas. Estas reglas permiten la

generación de una especificación concurrente y modular, más estructurada, sistemática y protegida contra errores de especificación que si el usuario hiciese un libre uso del lenguaje SystemC. Estas reglas además facilitan la aplicación de otras metodologías de nivel de sistema y de implementación. Por ejemplo, en **[PHSV04]**, se describió una librería de estimación de rendimiento de nivel de sistema basada en esta metodología de especificación. En **[CHPS03][HPS03]** se originó la metodología de generación de software, explicada en el Capítulo 3 de este trabajo. En **[RPHF04]** la metodología se uso como el punto de partida de una metodología de diseño de fuente única.

La metodología de especificación general provee también una representación gráfica de las facilidades de especificación básicas de SystemC. De esta forma, el usuario de la metodología puede documentar de forma gráfica y sin ambigüedades la especificación.

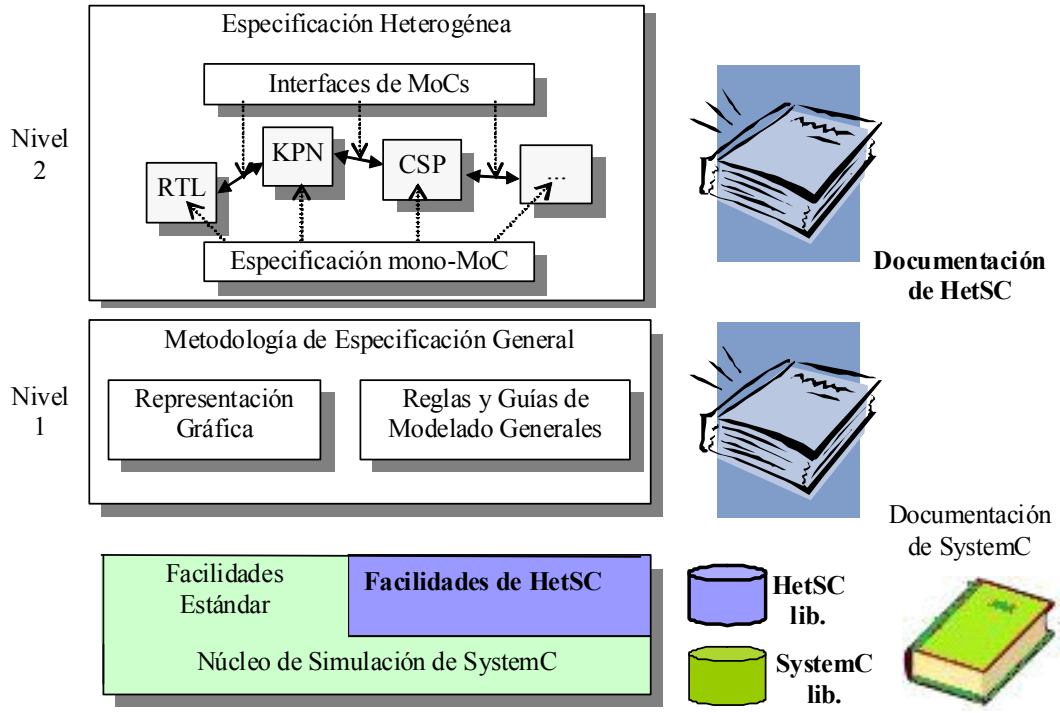



**Figura 2-30. Arquitectura de *HetSC*.**

En un segundo nivel, la metodología provee soporte para especificación heterogénea. En primer lugar, esto quiere decir que la metodología establece cómo se usa el lenguaje para cada estilo de especificación soportado, es decir, bajo cada MoC. Esta parte de la metodología se denomina *Especificación mono-MoC*. En segundo lugar, se habilita la construcción de especificaciones heterogéneas estableciendo cómo se pueden combinar partes bajo distintos MoCs en una misma especificación de sistema. Estas partes necesitan comunicarse y sincronizarse, lo que se resuelve mediante *Interfaces de MoCs*.

Un elemento característico de la metodología *HetSC* es la *librería HetSC*. La metodología de especificación muestra la forma en la que usar las facilidades de especificación proporcionadas por SystemC para desarrollar una especificación heterogénea. Por tanto, en su desarrollo, la metodología *HetSC* emplea, siempre que es

posible, las facilidades definidas en el LRM de SystemC y provistas por la librería SystemC. A pesar de ello, como se ha mostrado en la sección 2.2.5.1, SystemC carece de algunas facilidades necesarias para especificación heterogénea. La librería *HetSC* provee facilidades que suplen esas carencias. Con ello se consigue una metodología escalable que mantiene el núcleo de SystemC pequeño, general y estándar. De esta forma, la arquitectura metodológica de la Figura 2-30 lleva a la práctica la arquitectura representada en la Figura 2-22 y expuesta en **[GLM02]**.

En este trabajo se resume la metodología *HetSC*. El manual de especificación de *HetSC* **[HUM08]** y sus anexos, disponibles en **[HSC08]**, proveen información adicional acerca de la metodología.

### 2.3.3 Librería *HetSC*

Para la instalación y uso de la librería *HetSC* sólo se requiere la instalación previa de la librería SystemC, que no requiere modificación alguna. Tampoco se requiere la modificación de otras librerías basadas en SystemC, tales como TLM o SystemC-AMS, con las que, de hecho, la librería *HetSC* puede cooperar (secciones 2.5.7 y 2.5.8).

En la Tabla 1 se da una clasificación de las facilidades provistas por la librería *HetSC*, ejemplificándose cada caso con la enumeración de algunas de ellas.

| **Visibilidad** | **Facilidades** | **Ejemplos** |
|---|---|---|
| Visibles | Macros | *CH_MONITOR*, *SDF_NODE*,… |
| | Interfaces | *uc_rv_if*, *uc_rv_sync_if*,… |
| | Puertos | *uc_fifo_blocking_in*, *uc_fifo_blocking_out*,… |
| | Canales | *uc_fifo*, *uc_inf_fifo*, *uc_rv_sync*, *uc_rv*, *uc_rv_uni*, *uc_arc*, *uc_SR*, … |
| | Canales Frontera | *uc_inf_fifo_SR*, *uc_inf_fifo_signal*, *uc_inf_fifo_rv*, … |
| Transparentes | Chequeo de Reglas de MoCs y Facilidades de Reporte | *system_memory_monitor*, *uc_rpl*, … |

**Tabla 1. Facilidades provistas por la librería *HetSC*.**

Las facilidades de especificación visibles son las que el usuario maneja explícitamente en la especificación. Las facilidades visibles provistas por la librería *HetSC* son macros, interfaces, puertos y canales SystemC. Para distinguirlas de las provistas por la librería SystemC, se pueden denominar macros, interfaces, puertos y canales *HetSC*. No obstante, siguen siendo facilidades SystemC en la medida en que heredan clases SystemC (*sc_interface, sc_port, sc_prim_channel, sc_channel, etc*) y se implementan mediante facilidades SystemC (tales como el *sc_event*). Las facilidades HetSC suponen una extensión sintáctica y semántica sobre el núcleo del lenguaje. No obstante, la generación de un lenguaje nuevo no es el objetivo de *HetSC,* sino, utilizar y mejorar donde sea preciso SystemC para la especificación de sistemas embebidos heterogéneos. Actualmente, una gran parte de las facilidades provistas son nuevos

canales, que amplían las posibles semánticas de comunicación disponibles para comunicar procesos y dan el soporte requerido por los MoCs soportados.

Las facilidades transparentes al usuario son, fundamentalmente, chequeadores de reglas de MoCs a cargo de la verificación del cumplimiento de las reglas de construcción impuestas por el MoC. Su implementación tiene la forma bien de clases (que pueden ser clases públicas C++, pero que el usuario no necesita ni manejar explícitamente ni conocer para realizar la especificación), o bien de código incrustado en la implementación SystemC de las primitivas *HetSC* (por ejemplo, código de chequeo dentro de los canales). Varios de los chequeos son dinámicos, esto es, se realizan en tiempo de simulación. Esto permite en algunos casos mayor flexibilidad en el uso del lenguaje SystemC en tanto se asegura la aplicación de las reglas de los MoCs.

## *2.4 Metodología de Especificación General*

### 2.4.1 Facilidades de Especificación y Representación Gráfica

En la Figura 2-31 se muestran las primitivas básicas de especificación en *HetSC*. A cada una de estas facilidades se les asocia una representación gráfica.

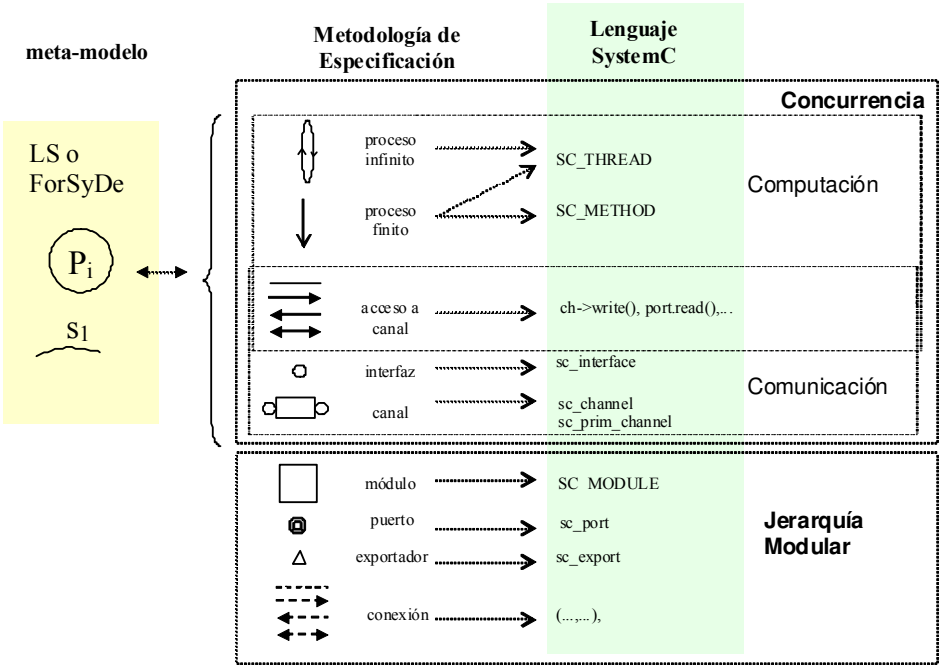



**Figura 2-31. Representación gráfica de *HetSC*.**

Estas facilidades tienen una estrecha relación con las facilidades básicas SystemC, aunque no se fuerza una relación biunívoca. Por ejemplo, un proceso finito se puede especificar usando un proceso SystemC de tipo SC_THREAD o uno de tipo SC_METHOD.

En la Figura 2-31 se distinguen las primitivas de especificación de las facilidades del lenguaje, y a su vez, de los procesos y señales de metamodelos, como LS o ForSyDe, que pueden formalizar la metodología. Estos metamodelos se centran en la

concurrencia y la semántica temporal, mas que en la estructura modular, tal y como se aprecia en la figura.

Para la especificación de concurrencia el usuario dispone de procesos infinitos (lazo con flechas de dirección), finitos (flecha), y canales (rectángulos con uno o más círculos en su contorno). Los procesos acceden a canales a través de métodos de acceso (líneas y flechas continuas). Las interfaces (círculos) agrupan métodos de acceso. Un método de acceso es una función miembro del canal. En *HetSC* se representan como puntos negros dentro del círculo de interfaz, aunque normalmente no se incluirán por claridad y síntesis de la representación. La representación gráfica de *HetSC* presenta una lógica constructiva. Por ejemplo, el canal se representa como una caja porque es un contenedor de una semántica de comunicación y un ámbito compartido entre procesos. Los círculos de interfaz alrededor de la caja representan la única vía de acceso al canal. Las puntas de flecha representan el sentido de la transferencia de los datos.

Para la especificación de jerarquía se cuenta con módulos (representados como cajas o rectángulos), que son accesibles a través de puertos y exportadores, que se representan en el borde del módulo. Los puertos son contenedores de una o más interfaces, por lo que se utiliza un doble círculo en su representación. El exportador se representa como un triángulo. Las conexiones entre puertos, entre exports, entre puertos y exports, entre canales y puertos, etc, se representan a través de líneas discontinuas. Si en esa conexión hay transferencia de datos, se representa mediante puntas de flecha.

Otro tipo de información puede aparecer en la representación gráfica de manera textual. En la Figura 2-32 se muestra un ejemplo, en el que un canal aparece con el nombre de su instancia y el del tipo de canal (entre paréntesis). También se detalla el nombre de los métodos de acceso que usan los procesos (*write* y *nb_read*) y los nombres de las interfaces que implementa el canal.

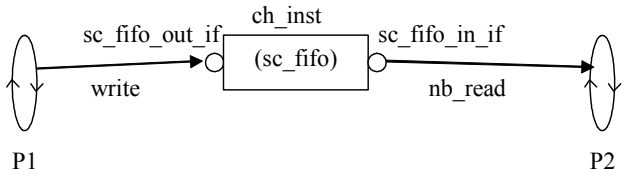



**Figura 2-32. Información textual puede complementar la representación gráfica.**

La representación gráfica permite una representación clara de la especificación, sin embargo, no es la especificación en sí misma ya que no contiene toda la información que la hace ejecutable. La representación gráfica omite partes de la especificación como el código de los procesos. Por conveniencia y claridad de la representación se puede omitir también toda o parte de la información textual.

## 2.4.2 Reglas y Guías Generales de Modelado

La metodología de especificación general centra la tarea del especificador en la descripción de funcionalidad concurrente, a la vez que permite un manejo más claro y seguro de esa concurrencia. También establece un estilo de especificación homogéneo, con un número reducido de primitivas, para las que se ha definido una representación gráfica.

En la Figura 2-33 se muestra la representación gráfica de una especificación *HetSC*. Esta especificación presenta una estructura de concurrencia y una estructura de módulos.

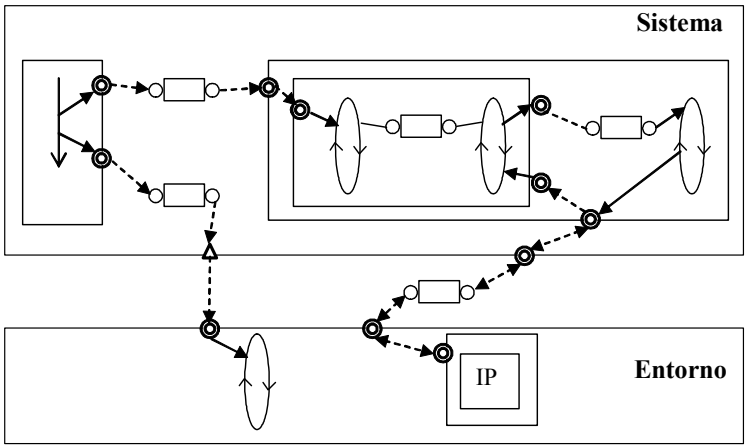



**Figura 2-33. Especificación *HetSC* con la consideración de jerarquía modular e inclusión de IPs.**

La estructura de módulos permite una división jerárquica de la funcionalidad del sistema. El especificador captura también el paralelismo potencial del sistema mediante la partición del cómputo en procesos, es decir, dotando de una estructura de concurrencia a la especificación.

En *HetSC*, la estructura de concurrencia no tiene jerarquía, es decir, no hay procesos conteniendo otros procesos. En la Figura 2-34 la estructura de concurrencia se ha representado eliminado la estructura de módulos. En HetSC los elementos que dotan la especificación de jerarquía no introducen semántica alguna. De este modo, a efectos semánticos, cualquier especificación *HetSC* se puede simplificar en una especificación concurrente no jerárquica como la de la Figura 2-34.

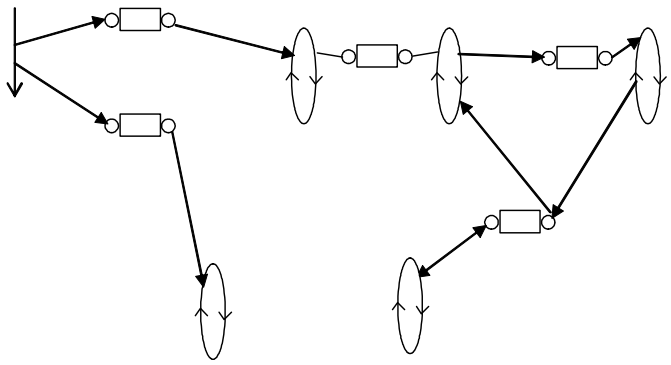


**Figura 2-34. Estructura de concurrencia de la especificación de la Figura 2-33.**

Una característica esencial de *HetSC* es la estricta separación entre computación y comunicación. El cómputo se comprende en los procesos y la comunicación en los canales. Una regla básica y general es que la *única forma de comunicación entre procesos es a través de canales*. No se permite otra vía de comunicación entre procesos. Esto formaliza la separación entre las facilidades de especificación destinadas a

cómputo (*procesos*) y las facilidades de especificación destinadas a comunicación de procesos (*canales*).

La metodología centra el esfuerzo del diseñador en escribir la funcionalidad concurrente, es decir, el código de los procesos. Existen dos tipos básicos de procesos, finitos e infinitos. Los procesos finitos alcanzan una condición de terminación, en tanto que los procesos infinitos no. Los procesos encierran código C/C++, incluyendo llamadas a función C/C++. Por claridad y síntesis, por defecto estas llamadas no aparecen en la representación gráfica. En la Figura 2-35 se muestra un ejemplo de procesos SystemC comprendidos en la metodología general de especificación.

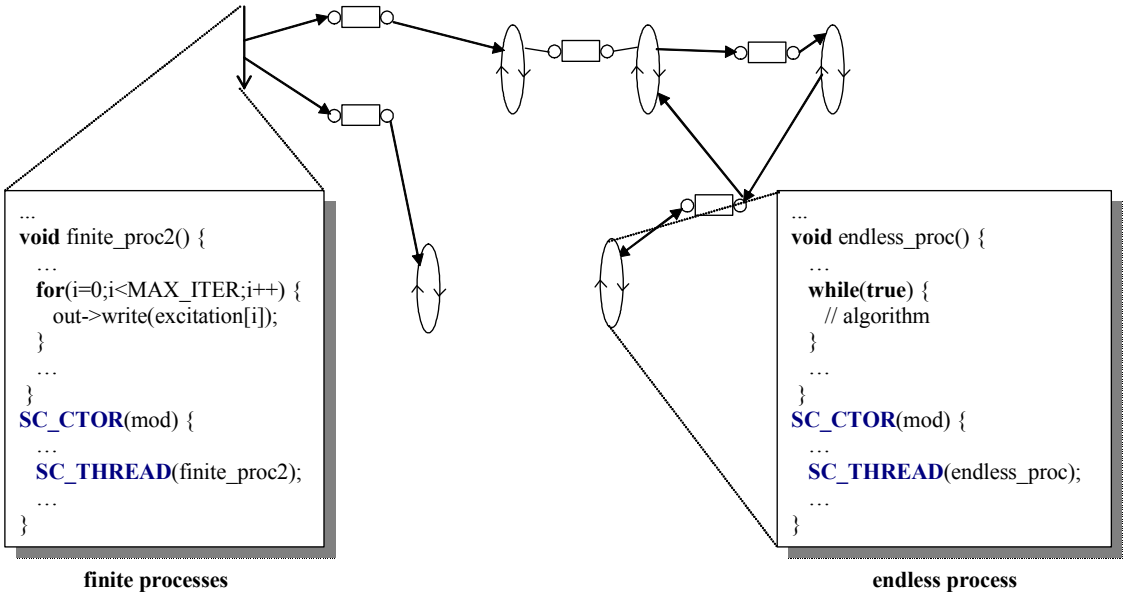



**Figura 2-35. Tipos de procesos en *HetSC*.**

Los procesos infinitos son procesos SC_THREAD con un lazo sin condición de terminación. Estos procesos deben presentar accesos bloqueantes para que el resto de procesos de la especificación pueda ejecutarse. En caso contrario, se dará un problema de inanición de dichos procesos. Los procesos finitos se implementan como procesos SC_THREAD que pueden llegar a la terminación o como procesos SC_METHOD.

Para comunicar los procesos, el especificador sólo necesita conocer los canales disponibles, su sintaxis, semántica y de qué forma se declaran y usan en la especificación. El especificador no necesita implementar los canales. La semántica de canal es la semántica de comunicación, que comprende tanto la transferencia de datos como la sincronización entre procesos. La semántica de un canal define aspectos tales como si el canal transfiere datos o no, si la transferencia implica bloqueo en la lectura y/o en la escritura, en qué condiciones se dan esos bloqueos, etc. Por lo tanto existen múltiples combinaciones y posibilidades que dan lugar a semánticas de comunicación diferentes, y por tanto, a canales diferentes. El tipo de canal es un aspecto distintivo del MoC empleado. La metodología *HetSC* provee un conjunto de canales que complementa el conjunto de canales estándar de SystemC. Las consideraciones y la representación del canal por parte de la metodología de especificación general son generales e independientes del tipo de canal.

La comunicación exclusiva a través de canales tiene implicaciones. En primer lugar, supone que el uso de variable compartida para transferir datos entre procesos no está permitido. Asimismo, tampoco está permitido el uso explícito de eventos SystemC (*sc_event*) para sincronización entre procesos. Una consecuencia inmediata es que no pueden aparecer estamentos de espera a evento (*wait(sc_event)*) en el código de los procesos. En la Figura 2-36 se ejemplifican las situaciones permitidas y no permitidas.

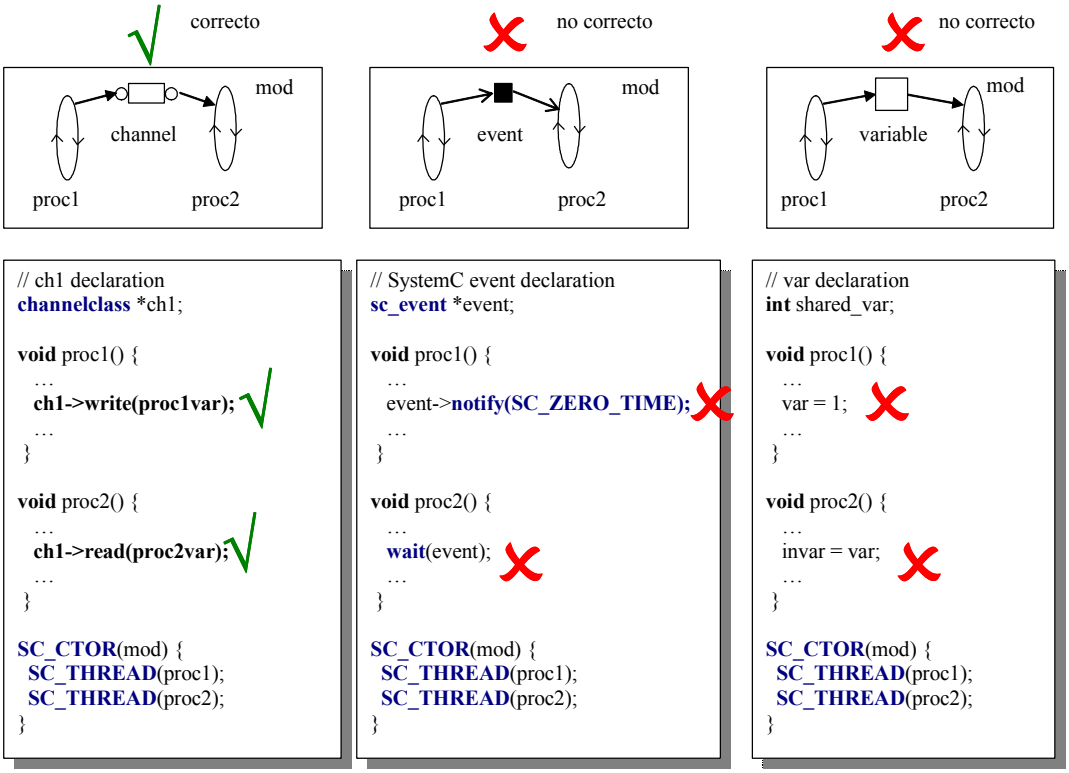



**Figura 2-36. La comunicación entre procesos sólo se puede realizar a través de canales.**

En la Figura 2-36, el evento SystemC se representa como una caja negra para resaltar que es un elemento de sincronización que no contiene valor alguno. Se han utilizado flechas de punta distinta a las que representan accesos a canal para representar que los accesos de notificación y espera a evento no conllevan transferencia de datos. Las variables compartidas son representadas como una caja sin interfaz de acceso alguna. Esto refleja la idea de mero contenedor primitivo de datos sin un control del acceso al mismo.

Los únicos estamentos *wait* que pueden aparecer en el código son los de espera temporal (por ejemplo, *wait(double, sc_time_unit)* ), dependiendo de si su presencia viola o no las condiciones del MoC bajo el que se especifica.

Los accesos a canal o puerto son los puntos del código del proceso donde ése se comunica con otros procesos. Esa comunicación puede significar sincronización y/o transferencia de datos. El acceso Consisten en una llamada a método de interfaz de una instancia de canal o puerto. El canal, es la implementación de un conjunto de interfaces. En muchos casos, se representan canales que implementan sólo dos interfaces, pero en *HetSC* (así como en SystemC) un canal puede implementar una o más de dos interfaces.

Como se ha mencionado, la metodología de especificación general soporta una estructuración jerárquica de la especificación en módulos que es ortogonal a la estructura de concurrencia. Las facilidades de especificación empleadas para ello son el módulo, el puerto y el exportador. El módulo es un contenedor de una descripción de comportamiento concurrente, que contiene procesos, canales y otros miembros de datos del módulo (no representados gráficamente). El módulo puede contener también una descripción estructural, es decir un conjunto de módulos conectados a través de canales, puertos y exportadores en un nivel de jerarquía inferior. El exportador se emplea en esta metodología para permitir incluir instancias de canales en un módulo. De ese modo se podrá, tal y como se representa en la Figura 2-33, conectar directamente un puerto del módulo de entorno a un exportador del módulo de sistema. Es decir, un canal de entrada/salida puede formar parte del módulo de sistema. No tiene que estar necesariamente entre el módulo de sistema y el de entorno.

*HetSC* no exige una correspondencia estricta módulo-proceso. No obstante, existe un grado de relación entre la estructura de concurrencia y la de módulos debido a algunas reglas que impone el empleo de SystemC. Son las siguientes:

- Cada proceso de la especificación debe estar contenido al menos en un módulo
- Un proceso sólo puede estar declarado en un módulo.

La metodología de especificación general permite también la inclusión de bloques de propiedad intelectual (bloques IP). Los IPs se deben incluir como parte del entorno de la especificación. Si el IP no presenta puertos compatibles, se debe desarrollar un módulo de adaptación. En este sentido, se trata de explotar las normas SPIRIT **[SPI06]** para la integración de IPs en SystemC. Siguiendo este tipo de recomendaciones, *HetSC* establece como obligatorio el que cualquier comunicación entre procesos pertenecientes a distintos módulos tiene que pasar a través de puertos y/o exportadores. En ese caso, el acceso a canal ha de ser necesariamente a través de acceso a puerto. La Figura 2-37 representa algunos casos permitidos y no permitidos.

La generación de la estructura de módulos es estática. También lo es la generación de la estructura de concurrencia. Es decir, tanto los procesos y canales, como los módulos, puertos y exportadores se generan antes del comienzo de la simulación. Además, el número de módulos y procesos es finito. En muchos trabajos donde se formalizan los MoCs se asume ese paralelismo limitado y estático.

En la Figura 2-38 se muestra la estructura general del código *HetSC*. Para realizar una especificación ejecutable *HetSC* basta con la inclusión de las librerías SystemC y *HetSC* mediante los ficheros de cabecera *"systemc.h"* y *"hetsc.h"* (Figura 2-38a). Si se va a emplear la especificación para la aplicación de otros flujos metodológicos de nivel de sistema, se puede emplear la cabecera *"general.h"* (Figura 2-38b). De esta forma, se puede, por ejemplo, generar software mediante la librería *SWGen* sin necesidad de cambiar el código fuente. Finalmente, la librería *HetSC* también puede utilizarse con otras librerías de especificación, como SystemC-AMS (Figura 2-38c).

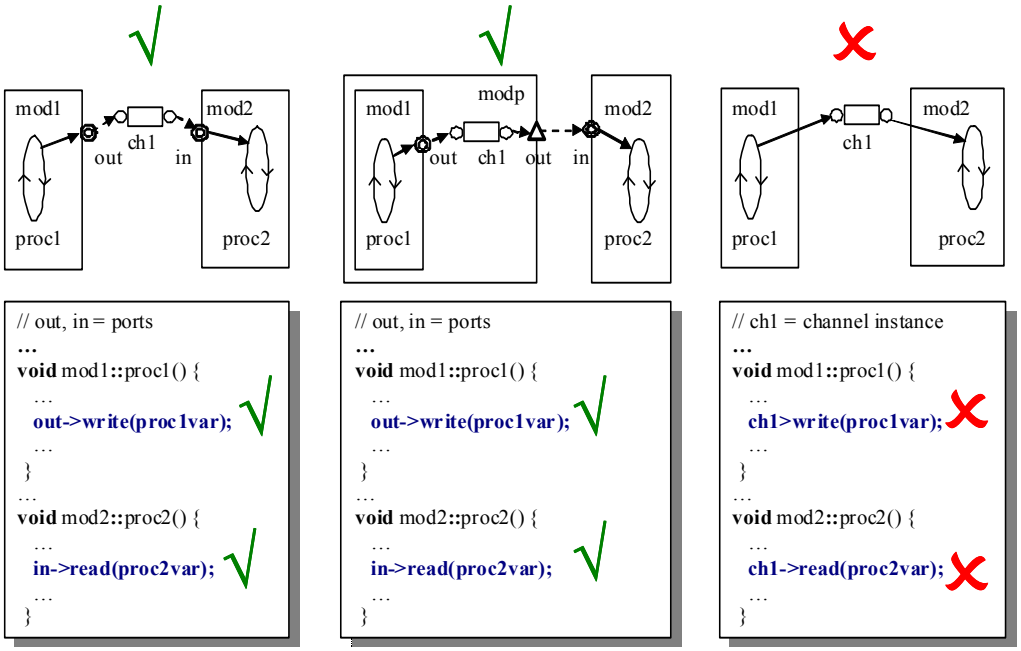



**Figura 2-37. El acceso entre procesos de distintos módulos ha de ser a través de puertos.**

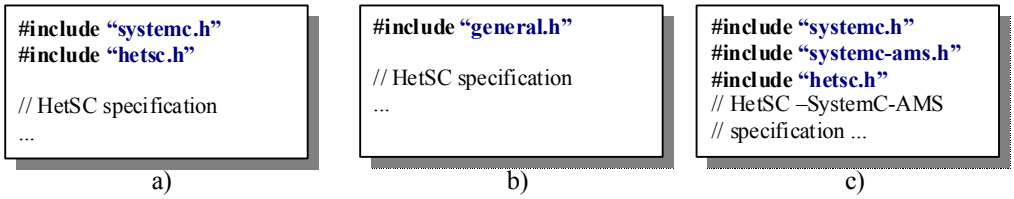



**Figura 2-38. La cabecera "general.h" incluye las librerías núcleo de SystemC y la *HetSC*.**

## *2.5 Especificación Heterogénea en HetSC*

### 2.5.1 Introducción

En un segundo nivel, la metodología *HetSC* provee soporte para especificación heterogénea. El soporte de heterogeneidad viene dado por un conjunto de reglas de modelado que complementan las de la metodología de especificación general y que explican qué facilidades del lenguaje usar y cómo usarlas para escribir una especificación bajo un MoC determinado. Asimismo, la metodología permite que una especificación tenga partes bajo dos MoCs distintos o más. Esas partes pueden conectarse entre sí a través de interfaces de MoC.

De este modo, por cada nuevo MoC incorporado:

- Si es preciso, se extiende el núcleo del lenguaje SystemC según las necesidades sintácticas y semánticas del MoC.
- Se proporcionan reglas y guías de especificación para ese MoC.

- Se proveen elementos que chequean y/o fuerzan a cumplir las reglas del MoC.
- Se proveen facilidades de reporte asociadas al MoC.

Adicionalmente, la documentación de *HetSC* esclarece los supuestos y las ventajas que aporta cada MoC. Esto es una ayuda al usuario para la selección del MoC. En cualquier caso, esto es más una cuestión de conocimiento y comprensión del MoC.

En *HetSC*, el soporte de MoCs se centra en tres de los dominios del metamodelo de Rugby: el manejo de tiempo, la computación y la comunicación. El nivel de detalle del manejo del tiempo permite establecer la clasificación raíz de los MoCs en *HetSC*: atemporales, síncronos y temporales **[Jan04]**. Atendiendo a esa clasificación los MoCs actualmente soportados son los siguientes:

- **MoC atemporales:** Redes de Procesos **[LePa95]**, Redes de Procesos de Kahn (KPN) **[Kahn74]**, Procesos Secuenciales Comunicantes (CSP) **[Hoa78]** y Flujo de Datos Síncrono **[LeMe87]**.
- **MoCs síncronos:** Síncrono Reactivo (SR) **[BeBe91]** y Síncrono de Reloj (CS) **[Jan04]**.

La metodología *HetSC* puede soportar también MoCs temporales. En estos MoCs, la especificación fija una información de tiempo estricto en todos los eventos del sistema y, por tanto, una relación de orden total entre todos los eventos del sistema. Entre ellos se consideran los refinamientos temporales realizados sobre especificaciones bajo MoCs atemporales y síncronos. Por ejemplo, los MoCs atemporales fijan únicamente un orden parcial entre los eventos del sistema. Sin embargo, un proceso de retroanotación temporal emplaza los eventos de la especificación sobre el eje de tiempo físico, de forma que la especificación resultante es temporal. Estos MoC temporales se denominan en *HetSC* como el MoC original más el adjetivo temporal, por ejemplo, KPN temporal (o T-KPN), SR temporal (o T-SR), etc.

Esa añadidura de información temporal también se da en las interfaces de MoCs cuando un MoC de menor de detalle en el manejo de tiempo se conecta a un modelo con un manejo de tiempo más detallado (por ejemplo, un MoC de tiempo continuo). En ese caso, para la realización de una simulación coherente, los eventos del primero se han de situar sobre el eje temporal más detallado.

Algunos MoC atemporales pueden preservar sus propiedades para cualquier refinamiento de la información temporal. Por ejemplo, una red de Kahn en la que los cómputos de los procesos tengan asociadas anotaciones temporales podría seguir conservando el determinismo. Sin embargo, en general, un refinamiento puede perturbar las propiedades del modelo. Los MoC síncronos son un caso claro en el que las anotaciones temporales pueden dar lugar a violaciones en los supuestos básicos del MoC que conducen a pérdidas de las propiedades del modelo.

El soporte de otros MoCs temporales, tales como los de tiempo continuo (CT), pueden justificar el desarrollo de nuevas máquinas de simulación. Esto ha dado lugar a las extensiones analógicas mostradas en la sección 2.2.5.5. Por ello, *HetSC*, en lugar de desarrollar nuevas capas con solucionadores para soportar este tipo de MoCs, ofrece medios para interoperar con esas metodologías, específicamente con SystemC-AMS,

que además de ser un estándar OSCI, al igual que HetSC, extiende el núcleo de SystemC de una forma desacoplada, sin modificar la librería SystemC.

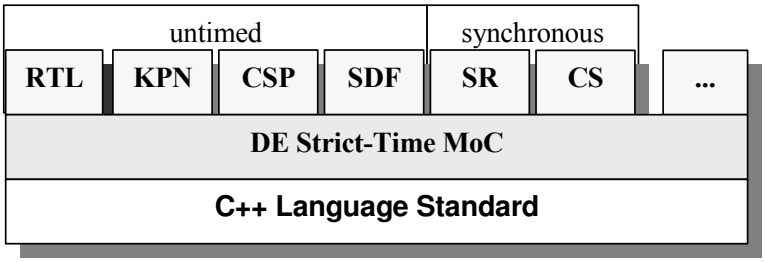



**Figura 2-39. Diversos MoCs son abstraídos a partir del MoC DE de tiempo estricto.**

La Figura 2-39 resume una característica distintiva de *HetSC*. Los distintos MoCs están directamente soportados sobre el kernel de simulación, de eventos discretos (DE) de tiempo estricto. Es decir, no se utilizan capas de sincronización intermedia o solucionadores específicos para el soporte de un MoC. La idea es que es posible abstraer y soportar de forma eficiente MoCs que manejan primitivas de comunicación más abstractas y menor detalle de información en el dominio temporal que el MoC de DE de tiempo estricto.

Dentro de cada nivel temporal, los dominios de computación y de comunicación son determinantes a la hora de establecer distinciones entre un MoC y otro. En cuanto a cómputo, en este nivel, HetSC establece el tipo de procesos que se pueden emplear y provee reglas adicionales para su codificación. En el dominio de comunicación, HetSC define el tipo de canales a emplear, su sintaxis y su semántica. Los canales empleados son un elemento característico y diferenciador del MoC. La Tabla 2 provee una serie representativa de características semánticas asociadas a los canales con posibles valores de las mismas. Esta tabla, si no completa, al menos puede habilitar un segundo nivel de clasificación de los MoCs de HetSC en función de las semánticas de comunicación, es decir, de los canales SystemC empleados en el modelo. Los canales SystemC primitivos (incluyendo los canales estándar, los canales HetSC y futuros canales que pudieran incorporarse a la metodología) pueden clasificarse según esta tabla.

<table>
<tr><th>Característica Semántica de la Comunicación</th><td colspan="4">Valores</td></tr>
<tr><th>Procesos relacionados</th><td>uno</td><td>dos</td><td colspan="2">varios</td></tr>
<tr><th>Sentido de Transferencia</th><td>unidireccional</td><td>bidireccional</td><td colspan="2">no hay transferencia</td></tr>
<tr><th rowspan="3">Condiciones de Bloqueo y Desbloqueo</th><td>No hay bloqueo</td><td>En lectura</td><td>En escritura</td><td>En todas</td></tr>
<tr><td colspan="2">cuando hay espacio</td><td colspan="2">cuando hay datos</td></tr>
<tr><td colspan="2">de llegada/salida del proceso</td><td>de tiempo</td><td>otras</td></tr>
</table>

| | | | |
|---|---|---|---|
| **Capacidad de almacenamiento** | estática | | dinámica |
| | nula | limitada | infinita |
| | local | | compartida |
| **Efecto de Escritura** | Sustitutivo (Destructivo) | | Acumulador (No Destructivo) |
| **Efecto de Lectura** | Muestreo (No-Destructivo) | | Consumidor (Destructivo) |
| **Acceso** | Secuencial | | Directo |

**Tabla 2. Consideraciones de la semántica de comunicación.**

HetSC es flexible en cuanto a que permite la habilitación e inhabilitación parcial de muchos chequeadores de reglas. Esto permite al usuario una sintonización de las reglas aplicables. Esto puede ser útil para usuarios más experimentados, que deseen utilizar conjuntos de reglas no directamente identificables como un MoC específico de *HetSC*. Esta flexibilidad puede ofrecer ventajas, por ejemplo, en velocidad de simulación con la eliminación del costo de algunos chequeos dinámicos. En definitiva, una característica principal de la metodología *HetSC* en el nivel de soporte de especificación heterogénea es que es abierta y puede integrar fácilmente cualquier MoC que pueda ser abstraído del kernel DE de tiempo estricto. Aún más, debería ser capaz de integrar e iteroperar con cualquier MoC que pueda ser simulado en C++.

### 2.5.2 Metamodelado de eventos, señales y tiempo en *HetSC*.

*HetSC* es una metodología inspirada en la base formal establecida por el metamodelo LS **[LeSV98]**. En esta sección se explica como se interpretan conceptos básicos de ese metamodelo, como el evento, la señal y el tiempo, en *HetSC*. Eso requiere, tener en cuenta el modelo de tiempo de SystemC, sobre el que se abstrae el manejo de tiempo de los MoCs de *HetSC*.

En el metamodelo LS, el evento consiste en un par valor-etiqueta, (V,T). En *HetSC*, por V se han de buscar por tanto valores asociados a un tipo computable en C/C++ (numérico, carácter, enumerado, etc). Para modelos síncronos, se añade además la ausencia de valor (⊥). Prácticamente cualquier variable visible en una especificación SystemC sería susceptible de ser interpretada como evento.

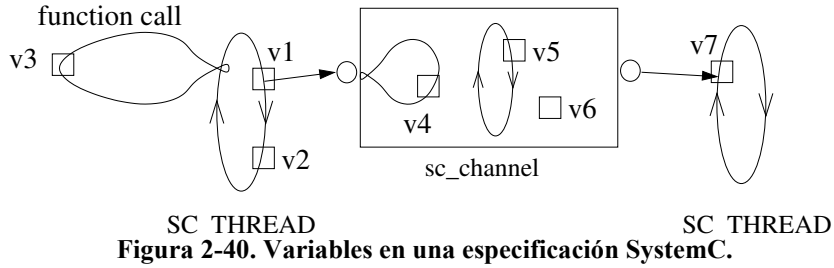



**Figura 2-40. Variables en una especificación SystemC.**

En la Figura 2-40 se representan variables en el cuerpo de un proceso (v1 y v2), de las cuales, alguna puede ser un parámetro de un método de acceso a canal (v1). En la Figura 2-40 se representa también una variable (v3) perteneciente al cuerpo de una función invocada por un proceso. Asimismo, se representan variables propias de la implementación SystemC de un canal (v4, v5 y v6), entre las que se pueden encontrar

variables del cuerpo de un método de acceso (v4), de un proceso interno (v5) si se trata de un canal jerárquico y variables miembro (v6), dentro del ámbito local de la clase canal. No obstante, como se ha visto, en *HetSC* el canal se maneja como una entidad cuya implementación interna no es visible, aunque se conozca su semántica. Por lo tanto v4, v5 y v6 no son visibles, lo que reduce las posibilidades de variables de las que interpretar una señal. En el metamodelo LS la consideración de eventos se centra fundamentalmente en la comunicación de procesos, por tanto, tiene sentido asociar los eventos del metamodelo LS a los valores transferidos en los accesos a canal, tales como los que va tomando la variable v1 en cada llamada de acceso a canal, (por ejemplo, 3, 5, etc, en la Figura 2-41). La etiqueta temporal asociada será la del momento SystemC en la que se realizan esas llamadas ($T_0$, $T_1$, etc en la Figura 2-41).

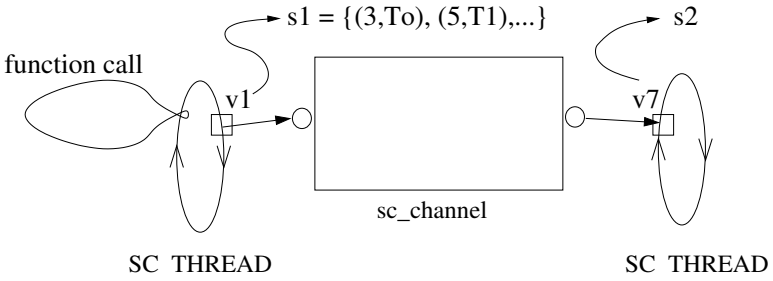



**Figura 2-41. Interpretación del evento y señal de LS en una especificación *HetSC*.**

En el metamodelo LS cada evento $e_i$ tiene una etiqueta T asociada. Dicha etiqueta se representa como $T(e_i)$ o, simplificadamente $T_i$. De esta forma, se puede entender T como un operador tal que, aplicado al evento $e_i$, obtiene la etiqueta temporal del mismo, es decir, $T(e_i) = T_i$. En el metamodelo LS, el dominio de T puede consistir en conjuntos muy distintos, por ejemplo, los enteros positivos, $\aleph$, los enteros, Z, o los reales $\Re$. Estas posibilidades dan lugar a diferentes grupos de MoCs, tales como los discretos (si $T \in \aleph$ o $T \in Z$) y los continuos ($T \in \Re$). En SystemC, la máxima información temporal que se puede manejar viene dada por el par $T=(t,\delta)$, donde t representa la estampa temporal de SystemC (con una noción de tiempo físico) y $\delta$ representa el ciclo delta de simulación en ese t (ciclo delta relativo). De esta forma, $T(e_i) = T_i = (t_i, \delta_i)$. T pertenece al producto cartesiano $\aleph \times \aleph$ ($T \in \aleph \times \aleph$), ya que tanto la estampa temporal como el ciclo delta se manejan con enteros positivos ($t \in \aleph$, $\delta \in \aleph$) [3].

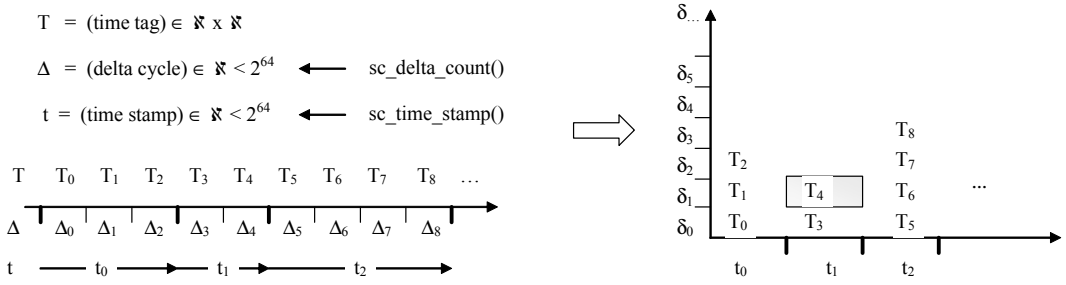



**Figura 2-42. La etiqueta de tiempo en el evento *HetSC* tiene una doble coordenada.**

[3] La estampa temporal de SystemC (t) es a su vez un par (valor, unidad). En SystemC, 'valor' pertenece a $\aleph$ y la unidad es la resolución definida para la simulación (por defecto, picosegundos).

En la Figura 2-42 se muestran dos posibles representaciones de la información temporal. En la de la izquierda, la relación de orden total entre las etiquetas temporales queda reflejada linealmente. En esa representación, Δ representa el ciclo delta de simulación absoluto. Será útil también una representación de doble eje como la de la derecha de la Figura 2-42. En el eje vertical se representa el avance de ciclos delta relativos, δ, que se inicializa tras cada avance temporal. En este caso, la etiqueta temporal T ocupa un espacio rectangular sobre el doble eje cartesiano definido por los ejes t y δ (en la Figura 2-42 se resalta la etiqueta $T_4=(t_1,\delta_1)$). Esto permite representar uno o más eventos de la especificación compartiendo la misma etiqueta temporal.

Independientemente de la representación adoptada, se trata de un modelo de tiempo discreto y las etiquetas temporales son un conjunto totalmente ordenado, es decir, $T_1<T_2<T_3<...$ En la simulación SystemC, la coordinada t es dominante en la relación de orden. Es decir, dadas dos etiquetas $T_i=(t_i, \delta_i)$ y $T_j=(t_j, \delta_j)$, si $t_i < t_j$ entonces $T_i<T_j$. En cambio, Si $t_i = t_j$, entonces la relación en δ es la que determina la relación en T. En general, la información temporal se maneja de forma implícita. Sin embargo, ésta se puede extraer de forma explícita en la especificación a través de las llamadas *sc_time_stamp()* y *sc_delta_count()*.

La máxima información temporal que se puede manejar en la especificación es la coordinada T completa. Ésta es, por tanto, la máxima precisión con la que se puede forzar el orden de los eventos en la simulación SystemC. En *HetSC* se juega con la abstracción de esa información temporal para soportar MoCs más abstractos en términos temporales. Tales MoCs desprecian parte de esa información, y con ello, admiten mayor flexibilidad en el orden de los eventos.

La interpretación del evento LS en *HetSC* da lugar a la interpretación de la señal LS en *HetSC*. En el metamodelo LS las señales son conjuntos de eventos, que tienen una relación de orden en tanto sus etiquetas asociadas la tienen. Por tanto, la señal LS en *HetSC*, es un conjunto de valores transferidos entre procesos. Ese conjunto de valores está ordenado por las etiquetas de tiempo T que les corresponden, por lo cual, la señal *HetSC* es una secuencia ordenada de valores transferidos en el canal. Esa secuencia estará totalmente ordenada siempre que no exista ningún par de eventos con la misma etiqueta T. En la Figura 2-41 se representa la abstracción de la señal s1, así como sus dos primeros eventos, dando por cada uno su valor y etiqueta temporal.

En el flujo de diseño asociado a *HetSC* puede ser interesante la consideración de otra posible abstracción del evento y señal LS. Se extraería una señal *LS* de la introspección del valor que va tomando una variable interna del proceso SystemC (v2). Esto permitiría formalizar otros casos. Por ejemplo, la metodología de perfilado Perfidy, asociada a HetSC, permite establecer puntos de captura en el código del proceso de forma que se podría extraer una traza del valor que va adoptando una variable del cuerpo del proceso. Así, esa traza se correspondería con una señal LS de salida (señal s2 en la Figura 2-43).

En la Figura 2-43, se ha representado la abstracción en el metamodelo LS de las dos señales *HetSC* correspondientes a la secuencia de eventos $e_{1i}$ y $e_{2j}$. El evento *LS* se representa como un aspa. El evento $e_1$ se asocia a un acceso a canal, en tanto que el evento $e_2$ a una introspección del valor de una variable propia del cuerpo del proceso.

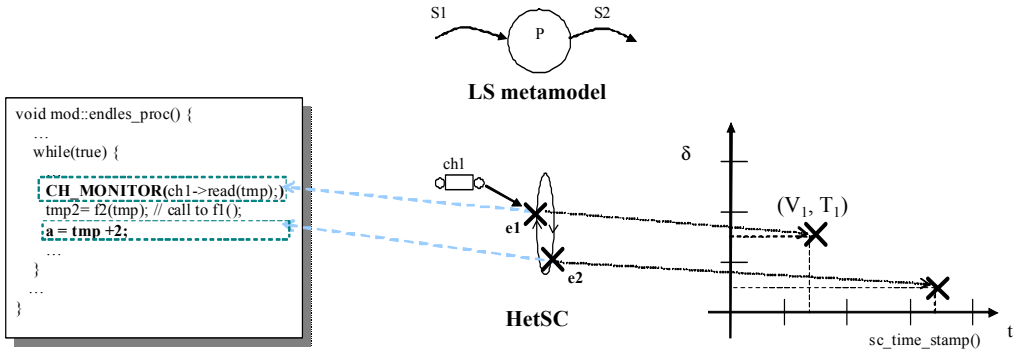



**Figura 2-43. El evento y la señal *HetSC* como interpretación del evento y la señal de LS.**

Es conveniente, por tanto, distinguir el evento LS abstraído de una especificación *HetSC* (o simplemente evento en lo sucesivo) del evento SystemC. El evento SystemC es una primitiva del lenguaje SystemC (*sc_event*), en tanto que el evento LS es un concepto de metamodelado. Una diferencia clara es que el evento LS tiene asociado un valor, en tanto que el evento SystemC no porta valor alguno. Hay que distinguir asimismo la señal SystemC de la señal LS (o señal). La señal SystemC es un canal primitivo estándar del núcleo del lenguaje (*sc_signal*). Es por tanto, una facilidad del lenguaje que posee su propia sintaxis y semántica.

Por otro lado, se puede encontrar una relación entre el evento y la señal LS y el evento y la señal SystemC. Por ejemplo, se puede abstraer una señal LS a partir de una secuencia de dos notificaciones a un evento SystemC, una en tiempo $T_1$=(1s,3) y otra en $T_2$= (1s,5). Esa secuencia de notificaciones se puede representar como la señal de notificación $s_n$= {(⊥,$T_1$),(⊥, $T_2$)} = {(⊥,(1s,3)),(⊥, (1s,5))}. Los detalles de esta relación no entran dentro del ámbito de este trabajo. La relación de *HetSC* con otro metamodelo, el metamodelo ForSyDe, es objeto de una de las líneas de investigación de **[AND08]**.

### 2.5.3 Modelos de Computación Atemporales

En una especificación bajo un MoC *atemporal* la única información temporal relevante es una relación de orden parcial entre los eventos del sistema. Esta definición se puede explicar contraponiéndola a la de los MoC *temporales*, caracterizados por un orden total en los eventos del sistema **[LeSV98]**. En los MoCs temporales todas las parejas de eventos de la especificación tienen una relación de orden establecida entre sí (orden total). Esa relación de orden viene definida por las etiquetas temporales de los eventos. Para dos eventos cualesquiera $e_i$ y $e_j$ de la especificación:

$$T(e_i) < T(e_j) \;\; \text{xor} \;\; T(e_i) = T(e_j) \;\; \text{xor} \;\; T(e_j) < T(e_j) \qquad \textbf{(1)}$$

donde *xor* representa una relación de exclusión mutua en el orden. Es decir, en una especificación bajo un MoC temporal, para cada par de eventos $e_i$ y $e_j$, se puede interpretar sólo una de las relaciones de orden, o bien $T(e_i) < T(e_j)$, o bien $T(e_i) > T(e_j)$, o bien $T(e_i) = T(e_j)$.

En cambio, en una especificación bajo un MoC atemporal existe al menos una pareja de eventos para la que no fija una relación de orden. Es decir, si los eventos de esa pareja son $e_i$ y $e_j$, $e_i$ se puede dar antes, al mismo tiempo o después de $e_j$. Esto se

expresa de forma compacta en *HetSC* mediante el símbolo "><", que expresa la "no-relación" de orden:

$$T(e_1) >< T(e_2) \quad \textbf{(2)}$$

Los MoCs atemporales permiten flexibilizar las relaciones de orden para un conjunto de pares de eventos de la especificación, por ejemplo, para aquellos para los que no existen relaciones de dependencias de datos.

Hay que distinguir entre la información temporal que fija la especificación y la que posteriormente fija cualquiera de las posibles implementaciones, entre ellas, la del simulador (a través del kernel de simulación). Las ecuaciones (1) y (2), se refieren a la semántica fijada por la especificación. Esta semántica tiene consecuencias en cualquier implementación que la preserve. Dado que un simulador es una posible implementación, las relaciones de orden (1) fijadas para parte de los eventos por una especificación atemporal se cumplirán en cualquier simulación.

Más aún, la simulación SystemC de una especificación atemporal, también hará que la semántica (2) de las parejas de eventos para las que no se establece una relación de orden, se convierta necesariamente en alguna de las relaciones de (1). La simulación añade una información adicional que define la relación de orden total entre todos los eventos de la especificación. De este modo, los eventos del MoC atemporal se mapean sobre el MoC DE de eventos discretos de SystemC, lo cual es necesario para que la especificación atemporal sea ejecutable. Sin embargo, en una simulación posterior, realizada con el mismo simulador SystemC, u otro diferente, no se debe esperar el mismo orden total. De la traza de simulación, se debe esperar una preservación de orden exclusivamente para el conjunto de parejas de eventos para las que la especificación fijó un orden de (1). Para el resto de parejas de eventos $(e_i,e_j)$ para los que la especificación atemporal no fija una relación $(T(e_i)><T(e_j))$, el orden de eventos resultante en simulación es irrelevante a efectos de contraste con el resultado esperado.

#### *2.5.3.1 MoC Redes de Procesos de Kahn (KPN)*

*HetSC* recopila las reglas asociadas a la especificación para cada MoC. Posteriormente, estas reglas se aplican al lenguaje SystemC, transformándolas en un conjunto de reglas y guías más sencillas y cercanas para el usuario SystemC. Estas reglas complementan las de la metodología de especificación general. Este proceso se ejemplificará para el MoC KPN, para agilizar posteriormente la explicación del resto de MoCs soportados por *HetSC*.

Las reglas del MoC KPN han sido extraídas fundamentalmente de **[Kahn74][WGM99]**. Existen una serie de reglas referidas al cómputo:

(1) *Un proceso estará computando o realizando un acceso (de lectura o escritura) a una sola instancia de canal. Si está realizando un acceso, en ningún caso estará accediendo simultáneamente a más de una instancia de canal fifo. Tampoco estará realizando simultáneamente distintos tipos de acceso (read y write), bien sea a distintas instancias o a la misma instancia de canal.*

(2) *El cómputo del proceso realiza un mapeo continuo. El mapeo continuo establece que la provisión de más unidades de datos de entrada al proceso sólo puede producir más unidades de datos de salida (monotonicidad) y que, además, no se requiere un número infinito de unidades de datos de entrada para obtener una unidad de datos de salida.*

(3) *La relación entre las unidades de datos de entrada y las unidades de datos de salida es funcional.*

Existen una serie de reglas referidas a la comunicación entre procesos y, más específicamente, a su semántica. En el caso KPN, estas son:

(4) *La comunicación entre procesos se realiza a través de una o más instancias de un único tipo de canal con una semántica de comunicación caracterizada por las siguientes reglas, de la (5) a la (13).*

(5) *El tipo de cada unidad de datos transferida es homogéneo para todos los accesos a la misma instancia de canal. No obstante, puede diferir entre instancias de canal.*

(6) *El sentido de transferencia de los datos es unidireccional. Existen solo dos tipos de accesos al canal uno de escritura y uno de lectura. El dato se transfiere al canal en el acceso de escritura y se obtiene del canal en el de lectura.*

(7) *La comunicación admite un tamaño de almacenamiento de unidades de datos ilimitado. Por tanto, el acceso de escritura es no bloqueante.*

(8) *Los accesos de lectura son bloqueantes. Los accesos de lectura no bloqueantes no están permitidos.*

(9) *La condición de bloqueo en el acceso de lectura es que la memoria intermedia interna del canal esté vacía.*

(10) *No hay más condiciones que la (9) que provoquen un bloqueo de los procesos.*

(11) *El acceso de lectura es destructivo. Es decir, consume el dato del canal, que no estará disponible en el siguiente acceso de lectura.*

(12) *El acceso de escritura es acumulativo. Es decir, la escritura del canal añade una nueva unidad de datos al canal, en donde permanece hasta que es leído.*

(13) *El tipo de acceso es secuencial, tanto en la lectura como en la escritura. El acceso de lectura preserva el orden de escritura (FIFO).*

Cada uno de los puntos (5)-(13) se puede relacionar con las filas de la Tabla 2, referidas a la semántica de comunicación del canal. Finalmente, hay una serie de reglas que se refieren tanto al cómputo de los procesos como a la semántica de comunicación:

(14) *El número máximo de procesos lectores de una misma instancia de canal es 1.*

(15) *El número máximo de procesos escritores de una misma instancia de canal es 1.*

(16) *Los métodos de acceso de introspección del canal (por ejemplo, lectura del número de unidades de datos disponibles, o de espacio en la cola, etc), o bien no están permitidos, o si se usan, no afectan al camino de datos o al flujo de control y, por tanto, a los datos de salida de la especificación. Los datos obtenidos mediante esos accesos no*

*pueden ser considerados para el cómputo de una salida del sistema. Sólo pueden ser usados a efectos de depurado.*

Todas estas reglas pasan a formar parte de la documentación de *HetSC* como un conjunto de reglas simplificado, que se enumera a continuación:

**Reglas de Computación:**

**KPN-1.** El cómputo se realiza en procesos **SC_THREAD**.

**KPN-2.** El cómputo debe realizar un mapeo continuo.

**KPN-3.** El cómputo debe realizar relaciones funcionales.

**Reglas de Comunicación:**

**KPN-4.** La comunicación se debe realizar a través de instancias de canal *uc_inf_fifo*.

**Reglas de Estructura de Concurrencia:**

**KPN-5.** El número máximo de procesos lectores de una instancia de canal es 1.

**KPN-6.** El número máximo de procesos escritores de una instancia de canal es 1.

La implementación de algunas de las reglas como regla HetSC es casi inmediata. Por ejemplo, la regla (1) se convierte en la regla *HetSC* KPN-1. En tanto el usuario usa un proceso SystemC (SC_THREAD) para especificar el cómputo concurrente, la regla (1) de KPN se cumple por construcción. En efecto, el código de un SC_THREAD es secuencial y no contempla sentencias del tipo *select* de ADA. Una sentencia de ese tipo sería capaz de desbloquear el proceso completando uno de entre dos o más accesos a canal, lo cual es una fuente típica de indeterminismo en modelos atemporales.

El cumplimiento de algunas reglas como la (2) y la (3) del MoC KPN depende de la correcta asimilación y uso por parte del usuario. Por ello, estas reglas, se enumeran de forma simplificada (reglas KPN-2 y KPN-3) y se provee en la documentación ejemplos de su significado en el código SystemC. Por ejemplo, la regla KPN-2 significa que el siguiente código no debe aparecer en un proceso bajo el MoC KPN:

```
(1)  ...
(2)  while(true) {
(3)     inchannel->read(datain);
(4)     dataout = (datain + dataout)/2.0;
(5)  }
(6)  outchannel->write(dataout);
(7)  ...
```

Un ejemplo más sutil de un incumplimiento de la regla KPN-2 es el siguiente:

```
(1)  ...
(2)  do {
```

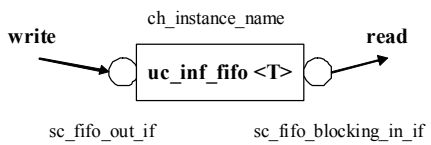
```
(3)     inchannel->read(datain);
(4)     dataout = (datain + dataout)/2.0;
(5)  } while(datain > 0);
(6)  outchannel->write(dataout);
(7)  ...
```

Para que este código de proceso fuera continuo se tendría que garantizar que el proceso que escribe la instancia de canal *inchannel*, de tipo *uc_inf_fifo*, escribe un valor menor o igual a 0 tras haber escrito un número finito de datos. De otro modo, se causaría una inanición local de parte de la funcionalidad del proceso (la sentencia (6) no se ejecutaría nunca), que extendería la inanición al resto de la especificación. Un incumplimiento de la regla KPN-3 se puede ejemplificar con el código siguiente:

```
(1)  ...
(2)  inchannel->read(datain);
(3)  tmp = gettimeofday(time,DST_WET);
(4)  if(time->tv_usec<500000) {
(5)     dataout = datain;
(6)  } else {
(7)     dataout = 0
(8)  }
(9)  outchannel->write(dataout);
(10)       …
```

La introducción de estamentos inderterministas como *gettimeofday* que afectan a los datos de salida del proceso basta para eliminar el determinismo que, de otro modo, el MoC KPN garantiza. Además, puede dar lugar a más situaciones de interbloqueo.

En algunos casos, la regla *HetSC*, consigue sintetizar el cumplimiento de varias reglas del MoC. Si el usuario sigue la regla KPN-4, cumplirá por construcción las reglas de la (4) a la (13) del MoC KPN. Para ello, *HetSC* provee el canal *uc_inf_fifo* a través de la librería *HetSC*. En la Figura 2-44 se da una representación gráfica del canal *uc_inf_fifo* y un extracto de su declaración.




```
template<class T>
class uc_inf_fifo:  public sc_fifo_blocking_in_if<T>,
                    public sc_fifo_out_if<T>,
                    public sc_prim_channel {
public:
    // constructors
    uc_fifo();
    uc_fifo(const char* name);

    // blocking read
    void read( T& );
    T read();

    // blocking write
    void write( const T& );
    // non-blocking write
    bool nb_write( const T& );
   …
};
```

**Figura 2-44. Representación y extracto de declaración del canal uc_fifo.**

El canal *uc_inf_fifo* es una plantilla parametrizable para el tipo *T*, correspondiente al tipo de la unidad de datos transferida por el canal. Presenta una semántica acorde con las reglas (6)-(13) del MoC KPN. Su semántica se aproxima a la de un canal de tamaño infinito por medio de una semántica de capacidad de almacenamiento no limitada. Esto significa que la capacidad de almacenamiento de cada instancia *uc_inf_fifo* puede crecer en tanto la plataforma de simulación lo permita. Esta semántica permite que el usuario no tenga que preocuparse de proporcionar un tamaño de almacenamiento al generar instancias de estos canales. El canal *uc_inf_fifo* implementa la interfaz *sc_fifo_blocking_in_if<T>*. Por lo tanto, solo implementa el método de lectura bloqueante *read*. Si el usuario trata de usar un método no bloqueante *nb_read* el usuario obtiene directamente un error en tiempo de compilación. Para escritura, el canal *uc_inf_fifo* implementa la interfaz *sc_fifo_non_blocking_out_if<T>* (método *nb_write*) ya que al contar con un tamaño de almacenamiento no limitado el acceso será siempre no bloqueante. El canal *uc_inf_fifo* presenta la particularidad de implementar también la interfaz *sc_fifo_blocking_out_if<T>* (método *write*) con la misma semántica no bloqueante que la otra interfaz de escritura. Esto se hace para facilitar el refinamiento de especificaciones KPN en BKPN. El usuario puede usar el método *write* siempre y al refinar los canales de *uc_inf_fifo* a *uc_fifo* no necesitará cambiar el código del proceso.

El canal *uc_fifo_inf* no implementa la interfaz de introspección. Por ejemplo, si se trata de acceder al método *num_available*, que en el canal estándar *sc_fifo* ofrece el número de datos presente en el canal, se obtendrá un error de compilación. Por lo tanto, este método de introspección no se puede utilizar en una especificación *HetSC*, como se representa en la parte derecha de la Figura 2-45. De este modo, el empleo del canal *uc_fifo_inf* obliga al usuario a cumplir la regla KPN (16), cuya violación se representa de forma genérica en la parte izquierda de la Figura 2-45,

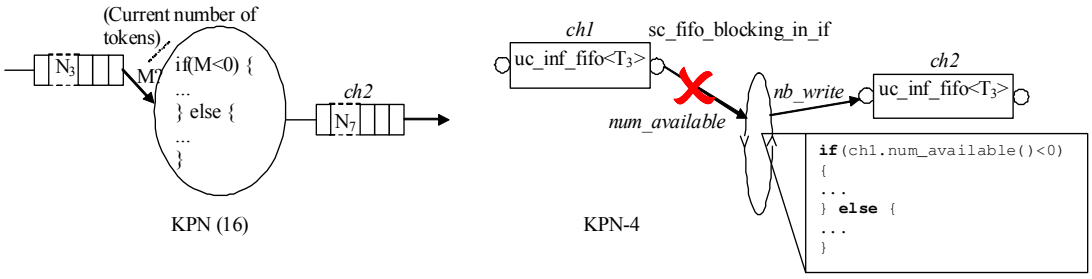



**Figura 2-45. El canal uc_inf_fifo impide los accesos de introspección.**

El canal *uc_inf_fifo* implementa también chequeos dinámicos para detectar si una instancia es escrita o leída desde varios procesos. De este modo, si se detecta una violación de las reglas KPN-5 o KPN-6, tal y como se representa en la Figura 2-46, se para la simulación y se ofrece un mensaje al usuario que permite detectar y localizar el error de especificación. De este modo, se fuerza al usuario a cumplir las reglas (14) y (15) del MoC KPN.

La librería permite configurar el nivel de gravedad de las violaciones de las reglas de especificación. De esa manera se puede permitir la continuación de la simulación y avisar del error, o simplemente ignorar esa situación. También es posible configurar el nivel de detalle de los informes producidos. Por ejemplo, si el acceso de varios escritores se reporta como un error, se puede reportar o no qué procesos accedieron.

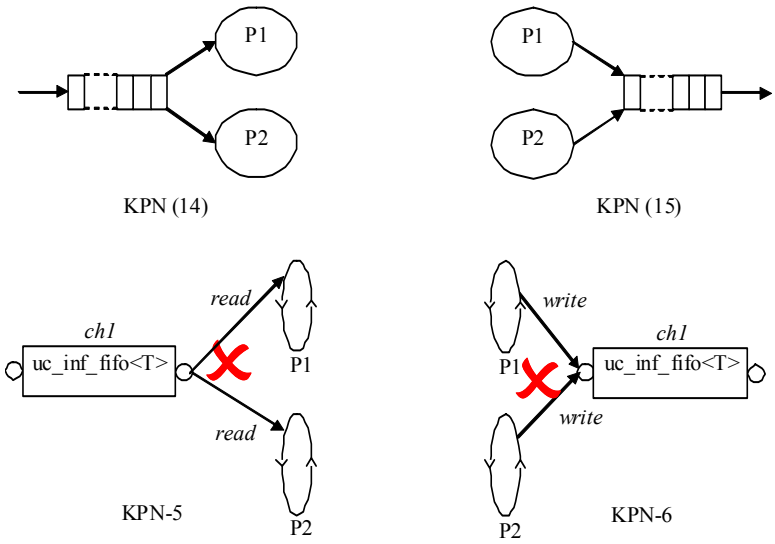



**Figura 2-46. Situaciones prohibidas en el MoC KPN.**

Algunos chequeos son comunes a varios MoCs. Por ejemplo, el chequeo de acceso de más de un proceso lector, es común a los canales de MoCs atemporales; entre ellos el *uc_inf_fifo*, pero también el *uc_fifo*, el *uc_arc*, etc, que se mostrarán más adelante. Sin embargo, en otros MoCs, la situación de varios procesos lectores no supone un problema, por tanto, no se restringe y ese chequeo no se implementa.

El modelo KPN de *HetSC* permite además establecer en la simulación, mediante una variable de configuración, un límite instantáneo para el tamaño total que suman los datos presentes en todas las instancias *uc_inf_fifo*. De esta forma se puede controlar si se llega a un tope de tamaño dedicado al almacenamiento en la transferencia de datos.

El canal *uc_inf_fifo* permite también la configuración de un límite de accesos consecutivos sin un avance de deltas en la simulación. Esto permite evitar posibles problemas de inanición o de desborde de la memoria de la plataforma de simulación en el caso de que se tenga un proceso productor infinito escribiendo sobre un canal *uc_inf_fifo* y sin bloqueos de por medio.

El canal *uc_inf_fifo* también ofrece la posibilidad de dar otros informes, como el número máximo de unidades de datos que han residido en el canal o las tasas de crecimiento del canal. De esta forma, el usuario puede obtener información útil para la generación de un modelo BKPN, basado en colas limitadas, a partir del modelo KPN. Los reportes de tasas de crecimiento y del número máximo de unidades de datos tienen un significado tanto más real cuanto más se anote temporalmente la especificación. Inicialmente, esa anotación puede darse únicamente en el modelo de entorno. En ese caso, los valores obtenidos consideran un comportamiento del sistema ideal en rendimiento temporal (cómputos instantáneos). Si el sistema no cumple las restricciones en estas condiciones, entonces, o las restricciones son imposibles de cumplir o la descripción concurrente tiene algún interbloqueo, inanición u otro tipo de error de especificación que se debe subsanar antes de abordar la fase de implementación. En una fase siguiente, se puede anotar temporalmente el sistema (bien manualmente, bien automáticamente con herramientas como Perfidy).

En definitiva, mediante el uso de SC_THREADs y canales *uc_inf_fifo* se puede construir una especificación *HetSC* con una estructura de concurrencia como la de la Figura 2-47, que refleja el modelo KPN de la Figura 2-3.

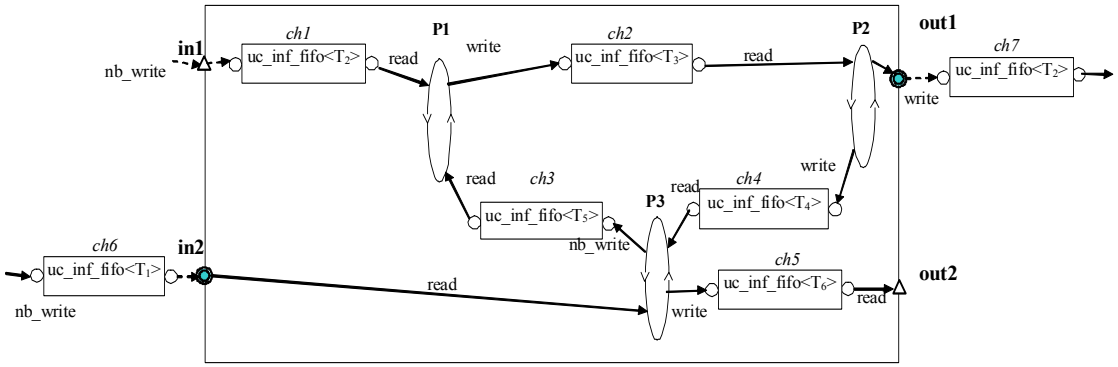



**Figura 2-47. Especificación *HetSC* de la red de Kahn de la Figura 2-3 en un solo módulo.**

La composición jerárquica a través de módulos también es posible. Esta jerarquía de módulos es ortogonal a la estructura de concurrencia, que es sobre la que se imponen las reglas del MoC. A efectos de estructura de concurrencia, y por tanto, a efectos del MoC, las especificaciones de la Figura 2-47 y la de la Figura 2-48 (en la que cada proceso se ha encerrado dentro de un módulo SystemC) son equivalentes. Ambas reflejan la mostrada en la Figura 2-3.

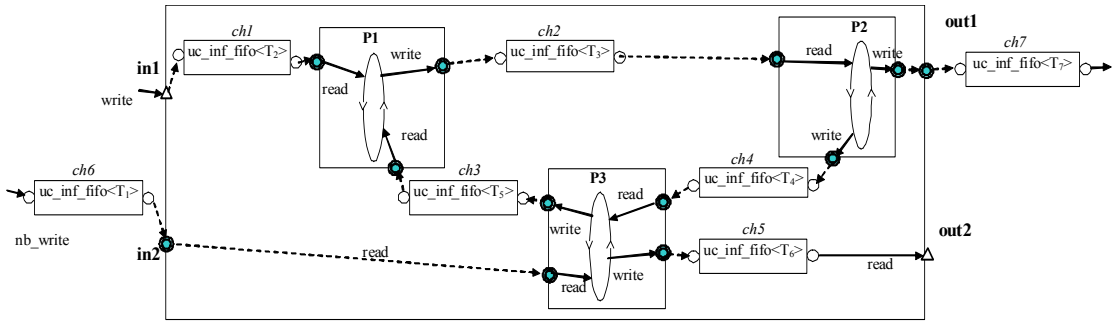



**Figura 2-48. Posible especificación *HetSC* de la red de procesos de la Figura 2-3.**

Para realizar la composición jerárquica se sigue la metodología de especificación general. La comunicación entre módulos se hace a través de puertos. Estos puertos pueden ser genéricos en tanto se empleen las interfaces que implementa el canal *uc_inf_fifo*. *HetSC* también provee puertos especializados no provistos por la librería SystemC. Por ejemplo, el puerto *uc_fifo_bloking_in* contiene exclusivamente la interfaz de lectura bloqueante para canal fifo de SystemC (*sc_fifo_blocking_in_if*). Así, por ejemplo, en la Figura 2-48, el proceso P1 lee el canal *ch1* a través de un puerto que se puede declarar de cualquiera de las dos maneras siguientes:

*uc_fifo_blocking _in<T> >  in_port_name;*

*sc_port<sc_fifo_blocking _in_if<T> >  in_port_name;*

Dentro del proceso P1, el acceso se refiere al nombre de la instancia de puerto:

*in_port_name->read(var);*

### *2.5.3.2 MoC Redes de Procesos de Kahn Limitadas (BKPN)*

El MoC BKPN es una variación del MoC KPN. En ambos casos, se trata de especificar una red de procesos concurrentes comunicados a través de canales de semántica FIFO unidireccional y bloqueante. A diferencia del MoC KPN, el MoC BKPN adicionalmente toma como parte de la especificación un parámetro de tamaño por cada instancia de canal fifo. Los inconvenientes frente al MoC KPN es que, además de exigir más información explícita en la especificación, el MoC BKPN presenta más condiciones de interbloqueo, como se mostró en la sección 2.2.1. Sin embargo, la ventaja es que un modelo BKPN está más cercano a la implementación, ya que está más refinado en cuanto a necesidades de recursos de memoria para comunicación.

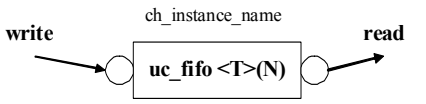
Por lo tanto, la recopilación de reglas es similar a la del MoC KPN, variando las relativas a las de semántica de comunicación. Específicamente, las reglas que varían respecto al MoC KPN son:

- Las instancias de canal FIFO tienen un tamaño de almacenamiento limitado, explícito en la especificación o implícito (valor por defecto).
- Se admiten exclusivamente interfaces de escritura y de lectura bloqueantes. Ahora existe una condición de bloqueo adicional, la de que la memoria de almacenamiento del canal esté llena cuando se va a realizar un acceso de escritura.

De esta forma, se genera un conjunto de reglas de especificación BKPN idénticas a las del MoC KPN, salvo por la regla BKPN-4:

**BKPN-4.** La comunicación se debe realizar a través de instancias de canal *uc_fifo*.

El canal *uc_fifo* es otro canal provisto por la librería *HetSC*. En la Figura 2-49 se da una representación gráfica y un extracto de su declaración.




```
template<class T>
class uc_fifo:  public sc_fifo_blocking_in_if<T>,
                public sc_fifo_blocking_out_if<T>,
                public sc_prim_channel {
public:
    // constructors
    uc_fifo(int size = 16);
    uc_fifo(const char* name, int size = 16);

    // blocking read
    void read( T& );
    T read();

    // blocking write
    void write( const T& );
   …
};
```

**Figura 2-49. Representación y extracto de declaración del canal uc_fifo.**

El canal *uc_fifo* implementa las interfaces bloqueantes de SystemC *sc_fifo_blocking_in_if* y *sc_fifo_blocking_out_if*. En términos semánticos, es similar al canal *sc_fifo* estándar de SystemC. Sin embargo, el canal *uc_fifo* impide que el usuario genere código SystemC con accesos no bloqueantes, tanto en escritura como en lectura. Como el canal *uc_inf_fifo*, también impide el uso de accesos de introspección. Esas situaciones, se detectan en tiempo de compilación. También como el canal *uc_inf_fifo*, el canal *uc_fifo* implementa chequeos dinámicos, como el que detecta el acceso desde más de un proceso escritor o desde más de un proceso lector. Otro chequeo detecta si un

proceso accede como escritor y lector a la misma instancia de canal. Esta situación puede provocar interbloqueo parcial, al menos del propio proceso si, por ejemplo, éste espera un dato que él mismo deba aportarse. Para asegurar que el usuario es consciente de esta posibilidad, *HetSC* plantea esta situación por defecto como un error. No obstante, dado que se pueden escribir procesos escritores y lectores a la misma instancia que no introduzcan interbloqueo, este chequeo se hace configurable en *HetSC*.

Para ilustrar cómo el código SystemC generado bajo el MoC BKPN de *HetSC* se atiene a los supuestos de un MoC atemporal, mostrados al principio de la sección 2.5.3 se utilizará el ejemplo de la Figura 2-50.

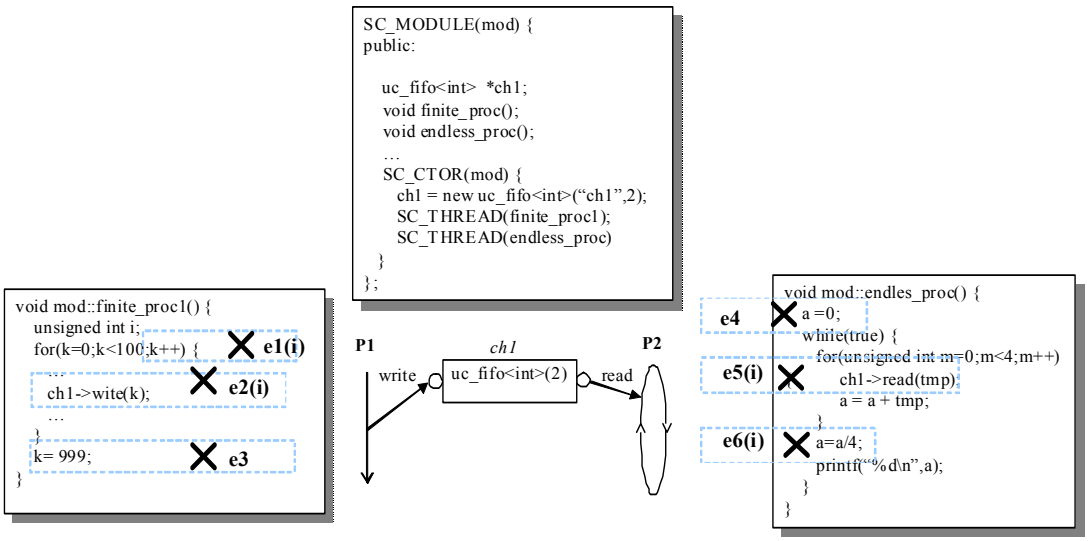



**Figura 2-50. El MoC PN fuerza una relación parcial de eventos.**

Dos procesos, uno finito, P1, y otro infinito, P2, se comunican mediante un canal *uc_fifo*, capaz de transferir unidades de datos de tipo entero (*int*). La capacidad de almacenamiento del canal es 2. En la Figura 2-50 se muestra un detalle del código de cada proceso. Sobre ese código se abstraen eventos correspondientes a los accesos a canal ($e_2$ y $e_5$) y algunos eventos de introspección ($e_1$, $e_3$, $e_4$ y $e_6$). En algún caso, una sentencia de código da lugar a un solo evento (tal es el caso de $e_3$ y $e_4$). En otros casos, la sentencia es recurrente y supone la generación de múltiples eventos. Para representar un conjunto de eventos se ha utilizado la notación $e_j(i)$, donde j identifica el punto de código SystemC, en tanto que i identifica el evento *LS* abstraído. Por ejemplo, $e_1(i)$ representa a los 101 eventos de asignación a la variable k. Existe un 102-ésimo evento de asignación de k (k=999), representado como $e_3$.

Considerando que existen dos procesos concurrentes y que en un MoC atemporal no se maneja otra información temporal que el orden de eventos, se puede afirmar que el evento $e_4$ no tiene relación temporal alguna con el evento $e_1(0)$ (primera asignación de k, consistente en k=0). Es decir, $T(e_1(0))><T(e_4)$. Esto es una razón suficiente para afirmar que se está ante un MoC atemporal, ya que no existe un orden total entre todas las parejas de eventos del sistema. No obstante, sí existe un orden parcial de eventos. Es decir, existen parejas de eventos que presentan relaciones de orden. Las hay de varios tipos. Unas son de causalidad forzada por la semántica secuencial del proceso. Por ejemplo, en el proceso P1, $T(e_2(i)) \geq T(e_1(i)) \ \forall i \in [0,99]$. También se cumple en el

proceso P1 que $T(e_1(i+1)) \geq T(e_1(i))$ y que $T(e_2(i+1)) \geq T(e_2(i))$ $\forall i \in [0,98]$. Se pueden establecer razonamientos análogos para el proceso P2. También es posible aplicar la propiedad de transitividad de relación entre eventos para deducir, por ejemplo, que $T(e_1(0)) \leq T(e_3)$. También existen relaciones de orden entre eventos de distintos procesos. Estas vienen fijadas por las sincronizaciones entre procesos, que en la metodología *HetSC* se dan sólo a través de los canales. En el caso del MoC BKPN, el canal *uc_fifo* fija una condición de sincronización básica entre la escritura y la lectura. En términos de la Figura 2-50, establece que:

$$T(e_2(i)) \leq T(e_5(i)) \quad \forall i \in [0,99]$$

Esta condición, a su vez, por transitividad, fuerza múltiples relaciones de orden entre parejas de eventos en las que cada evento se asocia a procesos distintos. De este modo, se puede establecer, por ejemplo, que $e_1(0) \leq e_6(95)$. Hay que tener en cuenta que para deducir ciertas relaciones de orden se necesita un conocimiento completo de la semántica de comunicación y del código de especificación. Ello permite deducir, por ejemplo, que no existirá un evento $e_6(99)$.

### *2.5.3.3 Variaciones de los MoC KPN y BKPN*

Como se ha dicho, la metodología *HetSC* es flexible en el chequeo y aplicación de algunas reglas. De esta forma, se pueden obtener estilos de especificación como variantes de los MoCs soportados. Estas variantes pueden llegar a ofrecer las mismas propiedades cambiando algunas reglas de especificación por otras más convenientes para el modelo desarrollado. En esta sección esto se ilustra para los MoC KPN y BKPN. Supóngase por ejemplo un modelo como el de la Figura 2-51, en el que un proceso *router* coge datos procedentes de dos procesos generadores de paquetes, *sender1* y *sender2* y los encamina hacia otros dos procesos sumidero, *receiv1* y *receiv2*. La Figura 2-51 muestra un diagrama del modelo que puede representar tanto una red BKPN ($0 \leq N_1 < \infty$, $0 \leq N_2 < \infty$ y $0 \leq N_3 < \infty$), como una red KPN($N_1 = N_2 = N_3 = \infty$).

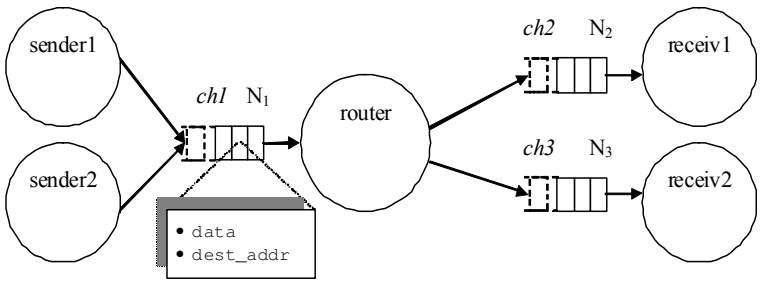



**Figura 2-51. Variación de red de Kahn sin limitación de procesos escritores.**

En cualquiera de los casos, asúmase que la red de la Figura 2-51 cumple todas las reglas de especificación KPN o BKPN de HetSC excepto la regla KPN-6, ya que existen dos procesos escribiendo sobre la instancia de canal *ch1*. El modelo de **[Kahn74]** se basa en esa condición para asegurar formalmente el determinismo de la red. Sin embargo, un diseñador puede concebir de forma más natural un modelo como el de la Figura 2-51, donde los datos de cada proceso fuente confluyen en el canal *ch1*. El usuario perderá la seguridad formal de **[Kahn74]**, pero puede asegurar el determinismo del modelo de otra forma.

Por ejemplo, en el caso de la Figura 2-51, supóngase que la única funcionalidad del proceso *router* es encaminar los datos a uno u otro receptor en función de una dirección de destino. Esa información de destino la conocen y envían los procesos *sender1* y *sender2*. En este caso, el diseñador puede sustituir la regla KPN-6 (o BKPN-6) por una regla de especificación alternativa. Es posible usar la estructura de especificación de la Figura 2-51 en tanto que cada proceso transfiera a través del canal *ch1*, además de información útil (*data*), una información del proceso de destino por cada unidad o grupo de unidades de datos (*dest_addr*). De esa manera, el proceso *router* sabe a qué canal (*ch2* o *ch3*) escribir el dato. De este modo, la especificación es también determinista, ya que la secuencia de datos transferidos en las fifos de salida *ch2* y *ch3* se mantiene cuando las secuencias de escritura de *sender1 y sender2* se reproducen. Nótese que la secuencia de unidades de datos que fluye por *ch1* no tiene por qué reproducirse. Por lo tanto, no se trata estrictamente de un MoC KPN, si no de una variante que preserva la misma propiedad, el determinismo. En cualquier caso, hay que manejar con cuidado estas equivalencias. Por ejemplo, si los procesos sender1 y sender2 son infinitos y no se da ningún detalle adicional en términos de semántica temporal, nada asegura que el modelo de la Figura 2-51 no sufra un problema de inanición (por ejemplo, que sólo sender1 escriba).

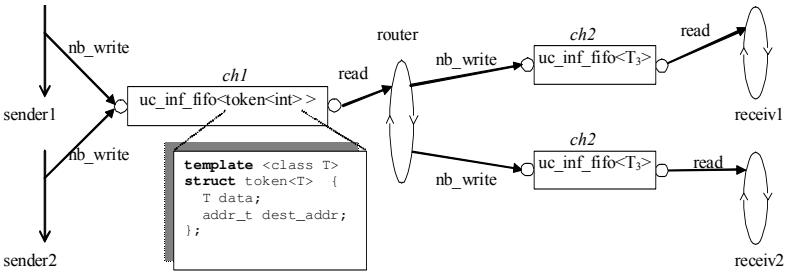



**Figura 2-52. Especificación *HetSC* de la variación de red de Kahn de la Figura 2-51.**

En la Figura 2-52 se muestra un diagrama *HetSC* de la especificación de la Figura 2-51 bajo el MoC KPN. Para la configuración por defecto del MoC KPN, esa especificación genera un error en tiempo de ejecución que informa de la imposibilidad de que, bajo ese MoC, los dos procesos, *sender1* y *sender2*, escriban en la misma instancia de canal, *ch1*, de tipo *uc_inf_fifo<token<int>>*. Sin embargo, en *HetSC* es posible deshabilitar la regla KPN-6. Actualmente, esto se hace definiendo una variable de preprocesado en un fichero de configuración de la librería *HetSC* (*global_opts.h*). Una vez eliminada esa regla, la ambigüedad en cuanto al destino de los datos que llegan al proceso *router* se resuelve transfiriendo a través de *ch1* unidades de datos de tipo *token*. La plantilla de estructura C/C++ *token* tiene un miembro *data* de tipo genérico *T*, que se destina a la transferencia de la carga útil, y un miembro *dest_addr*, que contiene la dirección destino, de tipo *addr_t*.

Imagínese ahora que el proceso *router* realiza un procesado de los datos recibidos, por ejemplo, un filtrado, en función del nodo origen. Ajustándonos a la regla KPN-6, es posible utilizar una instancia de canal *uc_inf_fifo* por cada proceso de origen y realizar un filtrado distinto según la instancia de canal leído. Sin embargo, el usuario puede aún utilizar una estructura como la de la Figura 2-52, añadiendo un nuevo campo a la

estructura *token* con la dirección de origen. Así, el proceso *router* cuenta con la información necesaria para que el resultado de la red sea determinista:

```
template<class T>
struct token {
   T data;
   addr_t dest_addr;
   addr_t orig_addr;
};
```

Si se considera un nuevo caso en el que el proceso *router* realice un procesado que dependa del orden relativo entre los datos enviados por *sender1* y *sender2*, entonces la estructura de la Figura 2-52 no es válida para garantizar la propiedad de determinismo. Habría que buscar condiciones adicionales o retornar a las reglas KPN.

El uso de un tipo *token* como una estructura con campo de carga útil más otros campos de protocolo es una técnica que puede tener otras utilidades. Por ejemplo, un campo puede marcar el final del envío de un número variable de *unidades de datos*. En ese caso, se puede definir la estructura *token* como la siguiente:

```
template<class T>
struct token {
   T data;
   bool eot; // flag to denote end of transfer
};
```

Los campos de protocolo se pueden transferir también en el canal fifo como *información en banda*. Es decir, el canal sólo transfiere datos de un tipo, que se corresponde con el tipo de dato de la carga útil. Entonces, se debe reservar un rango del tipo de dato transferido como información de protocolo y asumir que las ráfagas de unidades de datos transferidos guardan cierta estructura. Así, en el caso de la Figura 2-51 y la Figura 2-52, si el tipo de *ch1* es *unsigned int* de 32 bits, se pueden reservar, por ejemplo, los valores $2^{32}$ y $2^{32}$-1 como direcciones de *sender1* y de *sender2* respectivamente. Entonces, se debe estructurar la información obligando, por ejemplo, al envío de dos unidades de datos, primero una, con la dirección de destino, y luego otra con el dato. Desventajas de enviar los campos de protocolo como información en banda son la disminución del rango de valores transferibles como *carga útil* (en el ejemplo sería $[0,2^{32}-2]$) y la mezcla del cómputo de la funcionalidad con el de las operaciones de protocolo. Por ejemplo, se precisaría asegurar que los dos procesos, *sender1* y *sender2,* no generaran datos fuera del rango $[0,2^{32}-2]$.

Esto ilustra cómo el uso de protocolos y de una estructura en la información transmitida puede ayudar a preservar propiedades como el determinismo de la especificación. Los recursos de comunicación se minimizan, en tanto que se requiere más ancho de banda para transmitir la misma carga útil. *HetSC* se basa en MoCs que juegan con estructuras de especificación que garantizan propiedades independientemente de la estructura de los datos transferidos y el cómputo realizado. No obstante, es flexible en este sentido para permitir al usuario usar esas alternativas.

#### *2.5.3.4 MoC Procesos Secuenciales Comunicantes (CSP)*

El MoC CSP de *HetSC* define una red de procesos que se comunican mediante primitivas de comunicación de tipo *rendezvous*. No se toman todas las facilidades de especificación descritas en **[Hoa78]** o implementadas en **[ADA97]**, sino que ese conjunto se acota a un conjunto conveniente de primitivas de comunicación. El objetivo es habilitar la producción de redes de procesos fuertemente acopladas que, como en el caso KPN y BKPN, garanticen el determinismo de la especificación. Por ello, en *HetSC*, las reglas del MoC CSP reutilizan reglas empleadas en los MoCs KPN y BKPN. Las reglas que cambian son fundamentalmente las referidas a la semántica de comunicación, de forma que se cambia la semántica basada en colas FIFO por una semántica de comunicación tipo *rendezvous*. De este modo, el MoC CSP en *HetSC* ofrece una estructura más sencilla que el amplio lenguaje descrito en **[Hoa78]** e implementado en **[ADA97]** y más cercana a las estructuras de concurrencia para las que ha sido posible demostrar formalmente el determinismo y otras propiedades beneficiosas. En concreto, el MoC CSP de *HetSC*, no implementa primitivas de *guarda*. Por ejemplo, una implementación de una primitiva del tipo *select* de ADA violaría directamente la regla (1) del MoC KPN y BKPN, que se aplica igualmente en el MoC CSP en *HetSC*.

En definitiva, el usuario *HetSC* puede considerar el MoC CSP como parte de la misma familia de MoCs atemporales, entre los que se encuentran los dos expuestos anteriormente (KPN y BKPN). Estos MoCs se pueden clasificar y comparar en función de algunas variaciones en la semántica de comunicación. Esto se muestra en la Tabla 3.

| **Característica Semántica de la Comunicación** | **CSP** | **BKPN** | **KPN** |
|---|---|---|---|
| Capacidad Almacenamiento Necesaria | 0 | $1 \leq N < \infty$ | $\infty$ |
| Sentido Transferencia Datos | Sin Transf., unidireccional y bidireccional | unidireccional | unidireccional |
| Acceso escritura | Bloqueante | Bloqueante | No Bloqueante |
| Acceso lectura | Bloqueante | Bloqueante | Bloqueante |
| Condición de bloqueo | El proceso asociado no ha llegado | Escritura: Fifo llena<br>Lectura: Fifo Vacía | Lectura: Fifo Dacia |
| Condición de desbloqueo | Llegada del proceso asociado y fin de transferencia (si la hubiere) | Escritura: Una Lectura.<br>Lectura: Una escritura. | Lectura: Una escritura. |

**Tabla 3. Comparación de la semántica de canal en los MoCs CSP, BKPN y KPN en *HetSC*.**

La Tabla 3 toma de la Tabla 2 las características semánticas más distintivas en este caso. Específicamente, éstas son las referidas a las condiciones de sincronización y al tamaño de almacenamiento interno del canal, aspectos, por otro lado, ligados entre sí. El resto de las características de semántica de comunicación son compartidas por los MoCs KPN, BKPN y CSP de *HetSC*: escritura acumulativa, lectura destructiva, acceso a datos secuencial (si hay transferencia), etc.

Aunque el MoC CSP se puede entender como un MoC no dirigidos por datos, el tipo de sincronización que ofrece, por llegada y salida de procesos, se puede usar para garantizar el cumplimiento de dependencias de datos entre procesos. La semántica de sincronización fuertemente acoplada permite una transferencia de datos entre procesos en la que la necesidad de almacenamiento intermedio en el canal es nula. En efecto, cada acceso a un canal *rendezvous* sincroniza los procesos relacionados en la transferencia, con lo cual los datos pueden transferirse directamente entre las variables locales de cada proceso. Por eso, los canales *rendezvous* de *HetSC* se pueden considerar como canales con necesidad de capacidad de almacenamiento 0.

Las reglas de especificación del MoC CSP son similares a las de los MoCs KPN y BKPN. De hecho, las reglas CSP-1 a CSP-3 son idénticas: el cómputo se declara en procesos SystemC (SC_THREAD) con un código continuo y funcional. En cuanto a las reglas de comunicación, es posible instanciar tres tipos de canal *rendezvous* provistos por la librería *HetSC*: *uc_rv_sync*, *uc_rv_uni<T>* y *uc_rv<T1,T2>*. Los canales *rendezvous* de *HetSC* proveen de forma compacta, en una sola primitiva, la sintaxis y semántica requerida por la comunicación *rendezvous*. En la Figura 2-53 se muestra la representación *HetSC* y extractos de las declaraciones de estos canales.

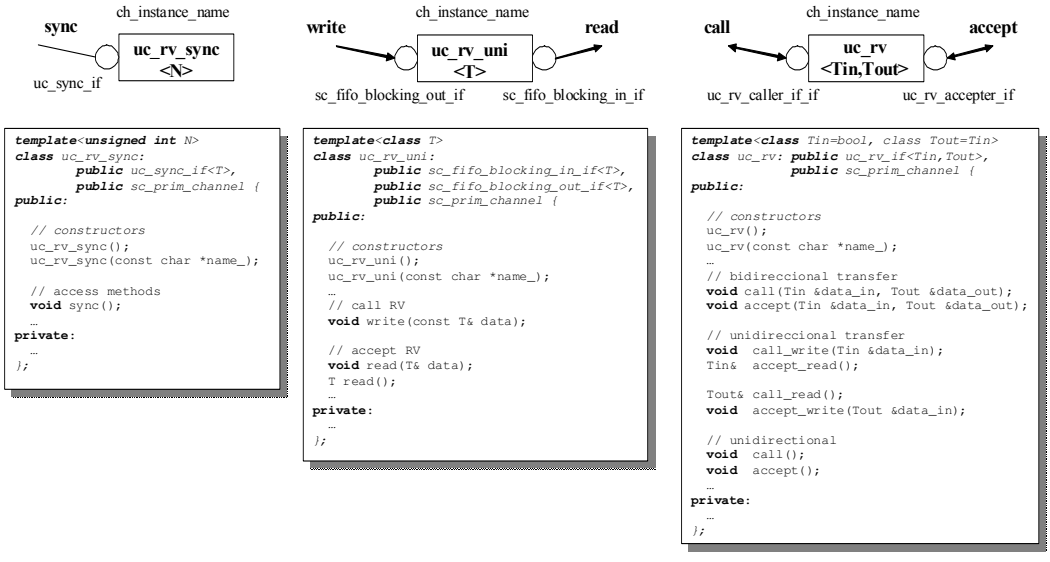



**Figura 2-53. Representación y extracto de la declaración de los canales *rendezvous* de HetSC.**

Los tres canales se distinguen por el número de procesos que asocian en la transferencia y por el tipo de transferencia de datos que realiza. En la Tabla 4 se resume la diferencia entre esos canales en términos semánticos.

| Semántica / canal rv | uc_rv_sync | uc_rv_uni | uc_rv |
|---|---|---|---|
| Nº de Procesos Asociados | $2 \leq N < \infty$ | 2 | 2 |
| Sentido Transferencia Datos | Sin Transferencia | Unidireccional | Bidireccional |

**Tabla 4. Comparación de la semántica de los canales rendezvous en el MoC CSP.**

El canal *uc_rv_sync* permite asociar N procesos, donde $2 \leq N < \infty$, a la misma sincronización de tipo *rendezvous*. El parámetro N se pasa como parámetro de plantilla. La Figura 2-54 muestra dos especificaciones para la sincronización de tres procesos (N=3) mediante el canal *uc_rv_sync*. Las dos son semánticamente equivalentes, con la misma estructura de concurrencia pero distinta jerarquía de módulos.

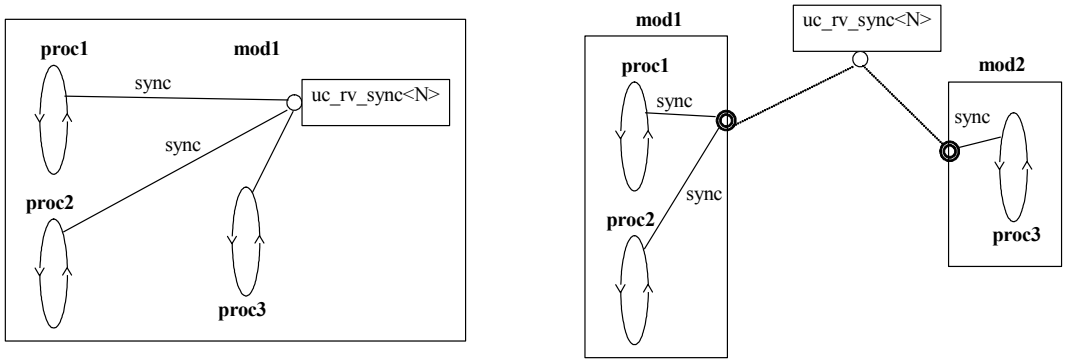



**Figura 2-54. Tres procesos sincronizándose por medio de un rendezvous.**

El canal *uc_rv_sync* implementa una interfaz *uc_rv_sync*, provista por la librería *HetSC*, que contiene el método *sync()*. Cuando un proceso realiza el acceso a *sync()* se sincroniza con otros N-1 procesos. El proceso continúa (desbloqueando a los otros) si y sólo si los otros N-1 procesos ya han llegado, y se bloquea si al menos uno no ha llegado aún. El canal *uc_rv_sync* no transfiere datos, con lo cual ninguno de los procesos se diferencian en términos de acceso al *uc_rv_sync*. El canal *uc_rv_sync* exige que los N procesos asociados al *rendezvous* sean siempre los mismos. Esta condición se establece en las reglas de especificación CSP de *HetSC*. De esta manera, se evitan situaciones en las que el canal *uc_rv_sync* sincronice M procesos, con M<N, o con M>N. La primera situación lleva al interbloqueo, al menos parcial, del sistema, ya que los M procesos quedarían bloqueados en su primera sincronización. La segunda situación lleva al indeterminismo de la especificación, ya que no estaría definido qué N procesos de los M asociados se sincronizan cada vez. El canal *uc_rv_sync* implementa un chequeo dinámico que es capaz de detectar la segunda situación (M>N).

La Figura 2-55 muestra una posible simulación para la especificación de la Figura 2-54. Obviando las líneas verticales que delimitan los deltas de simulación ($\Delta_i$), el diagrama exhibe un nivel detalle de información temporal propio de un MoC atemporal como CSP. No se precisa razonar en términos de deltas de simulación para prever el orden de ejecución de la simulación. Al principio, los tres procesos están listos para la ejecución. La semántica SystemC no define cual arrancará primero. En este caso, arranca el proceso *proc1*, que ejecuta hasta que llega al acceso *sync* del canal *rendezvous* (debido a la semántica no expulsora de SystemC). Entonces entra alguno de

los otros procesos involucrados. Cuál tampoco está definido según la semántica de SystemC. En el ejemplo entra *proc2*, que de nuevo debido a la semántica no expulsora de SystemC, continua hasta el acceso *sync* y se bloquea. Entonces, inevitablemente *proc3* pasa a ejecutarse, debido a que *proc1 y proc2* esperan la sincronización con un tercer proceso. Cuando *proc3* llega a la llamada *sync*, entonces se desbloquean los otros dos procesos y cualquiera de los tres puede continuar la ejecución. En este caso, el proceso *proc3* continua su ejecución el primero.

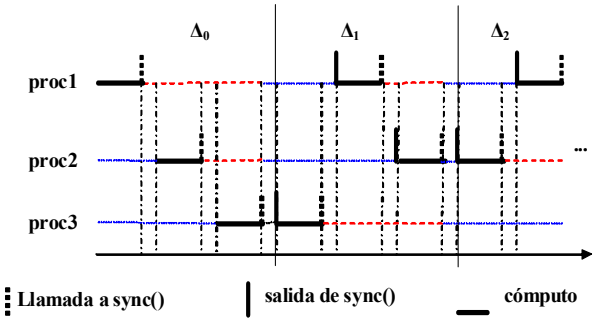



**Figura 2-55. Diagrama temporal de la especificación de la Figura 2-54.**

El diagrama de la Figura 2-55 sirve también para ilustrar cómo un MoC atemporal como el CSP se abstrae sobre el eje de deltas propio del MoC DE de eventos discretos del kernel de simulación de SystemC. En este caso, la sincronización de N procesos requiere al menos 1 delta de simulación SystemC.

Los otros dos canales *rendezvous*, el *uc_rv_uni* y el *uc_rv*, asocian dos procesos, el llamador y el aceptor. En *HetSC*, la disquisición principal entre proceso aceptor y llamador es que el aceptor es siempre único. En la configuración por defecto de la librería *HetSC*, los canales *uc_rv_uni* y el *uc_rv* requieren un único proceso aceptor. Sin embargo, esta regla se puede deshabilitar para permitir la aplicación de técnicas similares a las de la sección 2.5.3.3.

Tanto en **[Hoa78]** como en **[ADA97]**, los puntos de entrada al rendezvous (denotados '¡' y '?' en **[Hoa78])** están asociados al proceso o la tarea. En *HetSC*, esa asociación se hace dinámicamente. La semántica rendezvous se concibe como una semántica de comunicación que, por tanto, es englobada por un canal SystemC (*uc_rv_uni* o *uc_rv*). Los puntos de entrada al rendezvous son los propios métodos de acceso de esos canales. La asociación de esos puntos de entrada al proceso se realiza una vez que un proceso accede a la instancia de canal rendezvous. Entonces, el proceso es registrado como proceso aceptor o llamador esa instancia de canal. Si los accesos posteriores provienen de un proceso SystemC distinto, el canal lo detectará y lanzará, si procede, un error en tiempo de ejecución informando de la instancia de canal y los procesos que provocaron la violación de esta regla.

En la Figura 2-56 se representa un canal *uc_rv_uni* comunicando dos procesos. El canal *uc_rv_uni* soporta una transferencia unidireccional de unidades de datos de tipo genérico T, que van desde el proceso llamador al aceptor. En este canal, el proceso llamador siempre es escritor (transfiere datos hacia el canal), y el aceptor siempre es lector (recibe datos del canal). El canal *uc_rv_uni* implementa, al igual que los canales

*uc_fifo* y el *uc_inf_fifo*, las interfaces estándar de SystemC para escritura y lectura bloqueantes de canal FIFO, *sc_fifo_blocking_out_if* y *sc_fifo_blocking_in_if*.

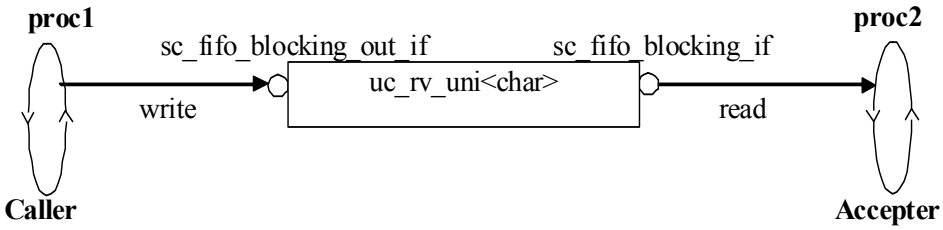



**Figura 2-56. Ejemplo de procesos consumidor y productor relacionados por un canal uc_rv_uni.**

El canal *uc_rv,* como el canal *uc_rv_uni*, asocia únicamente dos procesos. Soporta transferencia bidireccional, unidireccional en cualquiera de los dos sentidos y sincronizaciones sin transferencia. El canal *uc_rv* hereda e implementa la interfaz *uc_rv_if<Tin,Tout>*, provista por la librería *HetSC*. Su declaración es como sigue:

```
template<class Tin=bool, class Tout=Tin>
class uc_rv_if:  virtual public uc_caller_if<Tin,Tout>,
                 virtual public uc_accepter_if<Tin,Tout>  {};
```

Como se ve, la interfaz *uc_rv_if* hereda una interfaz llamadora y otra aceptora. Estas, a su vez heredan otras interfaces. En la Figura 2-57 se muestra un diagrama de las interfaces *rendezvous* provistas y, por tanto, con los métodos de acceso del canal *uc_rv*.

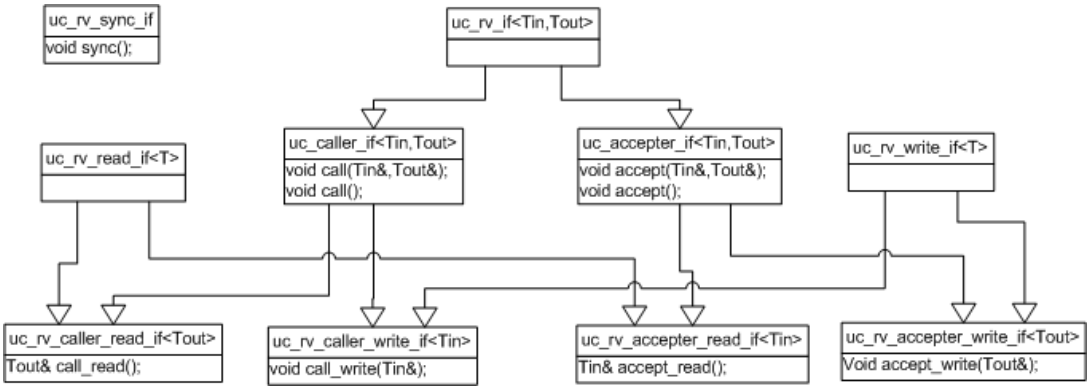



**Figura 2-57. Diagrama de interfaces *rendezvous* específicas provistas por la librería *HetSC*.**

El tipo de las unidades de datos transferidas desde llamador a aceptor (*Tin*) puede ser diferente del tipo de las unidades de datos transferidas desde aceptor a llamador (*Tout*). Se toma al proceso aceptor como referencia en el sentido de transferencia de los datos. El canal toma tipos por defecto, de tal forma que si al instanciarlo no se pasa ningún tipo se asume que se transfieren tipos *bool* en ambos sentidos. En ese caso, el usuario declararía la instancia de canal de la forma siguiente:

```
uc_rv<>  rvch("rvch");
```

El canal *uc_rv* también se puede instanciar con un solo parámetro de plantilla. En ese caso, el tipo pasado como parámetro se emplea en ambos sentidos de transferencia. Por ejemplo, la siguiente sentencia:

```
uc_rv<int>  rvch("rvch");
```

declara un canal *rendevous* bidireccional capaz de transferir atómicamente en la misma sincronización de tipo *rendezvous* dos unidades de datos de datos, ambas de tipo entero en los dos sentidos, una desde llamador a aceptor y otra desde aceptor a llamador. En cambio, una sentencia como la siguiente:

```
uc_rv<char,int>  rvch("rvch");
```

declara un canal *rendevous* bidireccional capaz de transferir atómicamente en la misma sincronización rendezvous una unidad de datos de tipo *char* desde el llamador al aceptor y una unidad de datos de tipo entero desde el aceptor al llamador.

Con cualquiera de las declaraciones anteriores, el canal *uc_rv* también se puede usar para sincronización *rendezvous* sin transferencia de datos entre los dos procesos (de la misma forma en que se usaría una instancia *uc_rv_sync<2>*). Para ello el proceso llamador accede al canal *uc_rv* mediante la llamada *call()*, mientras que el proceso aceptor accede mediante la llamada *read()*. Ambas llamadas no requieren argumentos.

El canal *uc_rv* soporta también transferencia unidireccional en cada sincronización rendezvous en cualquiera de los sentidos. Si la correspondencia entre el sentido de transferencia y las interfaces llamadora/aceptora es como en el canal *uc_rv_uni*, se usa la llamada *call_write* (equivalente al *write* en el *uc_rv_uni*) en el proceso llamador y la llamada *accept_read* (equivalente al *read* en el *uc_rv_uni*) en el aceptor. Si el sentido de transferencia es el contrario, el llamador utiliza la llamada *call_read*, en tanto que el aceptor la llamada *accept_write*.

En este caso, no todas las combinaciones de llamadas desde los procesos llamador y aceptor son válidas en cada sincronización rendezvous. El canal *uc_rv* incluye un chequeo de correspondencia para detectar situaciones en la que se da una combinación de llamadas inválida. De este modo, ese chequeo asegura que sólo se dan las combinaciones de llamadas correctas, mostradas en la Tabla 5. En la Tabla 5 se ha representado también el sentido del flujo de datos en la sincronización *rendezvous*.

| | | | |
|---|---|---|---|
| Sin transferencia | void call(); | — | void accept(); |
| Unidireccional | void call_write(Tin&); | → | void accept_read(Tin&); |
| | void call_read(Tout&); | ← | void accept_write(Tout&); |
| Bidireccional | void call(Tin&,Tout&); | ↔ | void accept(Tin&,Tout&); |
| | void call(Tin&,Tout&); | ↔ | void accept_read(Tin&);<br>void accept_write(Tout&); |
| | void call_write(Tin&);<br>void call_read(Tout&); | ↔ | void accept(Tin&,Tout&); |

**Tabla 5. Tabla de correspondencia de llamadas para el canal *uc_rv*.**

### *2.5.3.5 MoC de Flujo de Datos Síncrono (SDF)*

*HetSC* soporta una versión dinámica del MoC SDF. Esta aproximación permite realizar especificaciones bajo este MoC sobre el kernel de simulación de eventos discretos de SystemC. En *HetSC* se implementa un grafo SDF como una red de procesos SystemC comunicados a través de canales de tipo arco (*uc_arc_seq*). Este es un canal provisto por la librería *HetSC*. En la Figura 2-58 se representa una especificación *HetSC* correspondiente al grafo SDF de la Figura 2-10 (sección 2.2.1) en el que se ha omitido la jerarquía de módulos.

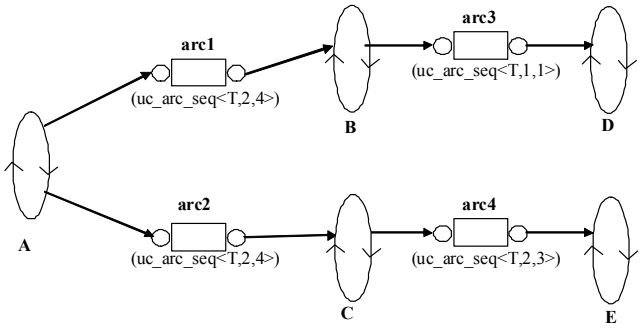



**Figura 2-58. Implementación del grafo SDF de la Figura 2-10 en SystemC.**

El MoC SDF de *HetSC* plantea un conjunto de reglas para el cómputo similares a las tres que presentan los otros MoCs atemporales de *HetSC*. Esto permite mantener un estilo homogéneo basado en procesos SC_THREAD. Sin embargo, el MoC SDF añade una cuarta regla para el cómputo que se reproduce a continuación:

**SDF-4.** Todos los accesos a un arco de entrada se realizan antes que cualquier acceso a cualquiera de los arcos de salida. El cómputo se realiza entre los accesos a los arcos de entrada y los accesos a los arcos de salida.

Esta regla comprende la noción del proceso como un cómputo atómico que lee primero todas las entradas, computa, y finalmente escribe todas las salidas. A eso se lo considera un disparo del nodo. La regla presenta además la flexibilidad de que permite que se realice el cómputo entre dos accesos de entrada o entre dos accesos de salida.

La norma de comunicación en una especificación SDF de *HetSC* es que la comunicación debe darse a través de canales SystemC de tipo arco. Estos canales arco pueden ser escritos por un solo proceso. En ese caso, el canal arco es un arco de salida del proceso. Un canal arco también puede ser leído por un solo proceso. En ese caso, se comporta como un arco de entrada del proceso.

Una vez definidas las reglas de cómputo y, por tanto, como se constituye un nodo mediante un SC_THREAD y una vez definida la implementación de los arcos de comunicación entre nodos como canales SystemC de tipo arco (*uc_arc_seq*), el MoC SDF en *HetSC* distingue entre nodos autónomos y nodos dependientes. Los nodos autónomos no tienen accesos a canales arco de entrada, pero sí canales arco de salida, con los que abastecen de datos a los nodos dependientes. Los nodos dependientes presentan accesos al menos a canales arco de entrada, y pueden ser disparados por los nodos autónomos o por otros nodos dependientes.

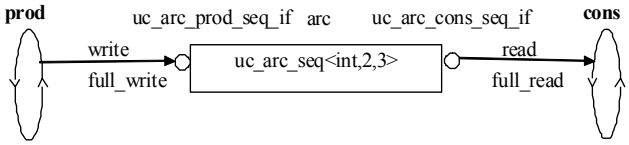



**Figura 2-59. Canal *uc_arc_seq*.**

Como se ha mencionado, la librería *HetSC* el canal arco, denominado *uc_arc_seq* y representado en la Figura 2-59. Este canal es una plantilla que se parametriza con el tipo de dato transferido en el canal y con otros dos números, que son las tasa de producción y de consumo del arco. La primera representa el número de unidades de datos que escribe el proceso productor en cada disparo (2 en la Figura 2-59). La segunda representa el número de unidades de datos que lee el proceso consumidor en cada disparo (3 en la Figura 2-59). El canal *uc_arc_seq* implementa tres interfaces, todas provistas por la librería *HetSC*. Por un lado, implementa una interfaz denominada *uc_arc_set_rates_if*, que, hereda dos interfaces, *uc_arc_set_prod_rates_if* y *uc_arc_set_cons_rates_if*. Estas interfaces proveen los métodos *set_prod_rate y set_cons_rate*, que permiten establecer las tasas de producción y consumo del arco en tiempo de elaboración. Por tanto, el usuario *HetSC* tiene flexibilidad a la hora de establecer las tasas, bien en tiempo de compilación (como parámetro de plantilla), bien en tiempo en tiempo de elaboración.

La Figura 2-59 representa sólo las interfaces del canal *uc_arc_seq* que proveer acceso al canal en tiempo de simulación. Estas son la interfaz de escritura secuencial (*uc_arc_prod_seq_if*) y la interfaz de lectura secuencial (*uc_arc_prod_seq_if*). Cada una provee dos tipos de métodos, uno para la transferencia de una única unidad de datos y otro para la transferencia atómica de un grupo de unidades de datos, definido por la tasa. Así, la interfaz *uc_arc_prod_seq_if* provee el método *write* para el primer caso y el método *full_write* para el segundo, que escribe en el canal en un sólo acceso un número de unidades de datos igual a la tasa de producción. Análogamente, el canal la interfaz *uc_arc_cons_seq_if*, provee los métodos *read* y *full_read*.

El canal *uc_arc_seq* presenta la semántica de comunicación específica para la especificación de grafos SDF dinámicos en SystemC. En algunos aspectos, es similar a los canales presentados anteriormente. Se utiliza para comunicar dos procesos, sincronizándolos y habilitando una transferencia unidireccional de datos. Un proceso tiene carácter productor y otro consumidor, con lo cual, se aplican de nuevo las mismas reglas de limitación a un sólo proceso escritor y un sólo lector de la misma instancia de canal. Además, la transferencia de datos mantiene una semántica FIFO, preservando la secuencia de unidades de datos de salida el mismo orden que la secuencia de entrada. Los accesos de escritura *write* y *full_write* son acumulativos (no destructivos) y los accesos de lectura *read* y *full_read* son consumidores (no destructivos).

Las características semánticas específicas de este canal vienen determinadas en gran medida por las tasas del canal. En función de éstas, se determina el tamaño de almacenamiento interno del canal y la semántica de sincronización. Por medio de la tasa de consumo se establece el mínimo número de unidades de datos que tienen que estar disponibles en la instancia de canal para permitir que el proceso productor se desbloquee. En caso de haber menos, se desbloqueará. Además, añadiendo la

información de la tasa de producción, se establece automáticamente, en tiempo de elaboración, el tamaño de almacenamiento del canal, que puede ser *max{production rate, consumption rate}* o bien (*production_rate* + *consumption_rate* – *1*), dependiendo de las relaciones entre tasas. El establecimiento del tamaño de almacenamiento del canal establece automáticamente la condición de bloqueo/desbloqueo del proceso productor. Cuando éste escribe unidad a unidad de datos, por medio del método *write*, el bloqueo se dará siempre que ya no haya espacio para una unidad más en la memoria interna del canal. Si se usa el método *full_write*, en cambio, el bloqueo se dará siempre que exista espacio para menos de *production_rate* unidades de datos en la memoria interna del canal. Adicionalmente, el canal *uc_arc_seq*, una vez que productor y consumidor pueden ejecutar, favorece la ejecución del consumidor. Esto se hace para garantizar una ejecución libre de interbloqueo en grafos SDF consistentes.

En la Figura 2-60 se representa una simulación para el caso de la Figura 2-59, asumiendo que productor y consumidor utilizan sólo los accesos *write* y *read*.

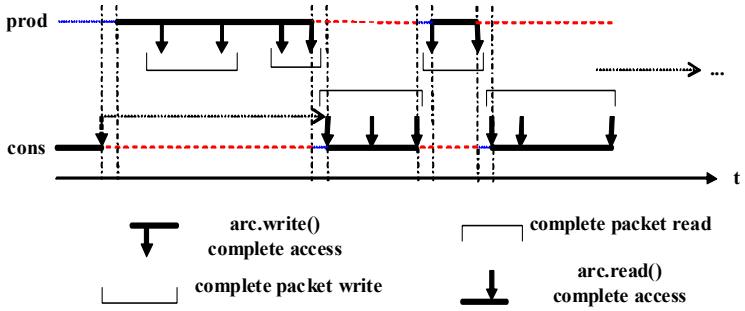



**Figura 2-60. Simulación de la especificación SDF de la Figura 2-59 con accesos *write* y *read*.**

El proceso *cons* comienza a ejecutar, pero tiene que bloquearse ya que no hay unidades de datos disponibles en el canal. Entonces el proceso *prod* comienza su ejecución y realiza cuatro accesos *write*. Cada acceso *write* inserta una unidad de datos en la memoria de almacenamiento intermedia del canal. Dado que la escritura tiene que hacerse en paquetes de *consumption_rate* unidades de datos, en ese caso, cada escritura de paquete requiere dos accesos *write*. Después de escribir el primer paquete, esto es, dos unidades de datos, esto todavía no es suficiente para disparar el proceso *cons*. Por tanto, se escribe un segundo paquete de datos. Después de que la primera unidad del segundo paquete de datos es escrito, sería posible desbloquear el proceso *cons*. Sin embargo, el canal *uc_arc_seq* requiere completar las trasferencias paquete a paquete. Por tanto, el cuarto acceso *write* es necesario. Una vez realizado, el proceso *cons* puede ejecutarse y realiza las transferencias de lectura mediante el método *read.*

En la Figura 2-61 el ejemplo de la Figura 2-59 se resuelve mediante el uso de métodos de acceso completo. Estos métodos permiten resolver la comunicación entre nodos de una forma compacta. El proceso *prod* emplea el método *full_write* para escribir en un solo acceso *production_rate* (2) unidades de datos. Por tanto, necesita solo dos accesos, uno por cada disparo, antes del disparo del proceso *cons*. El proceso cons, a su vez, usa el método *full_read* para realizar en un solo acceso la lectura de *consumption_rate* unidades de datos (3 en este caso).

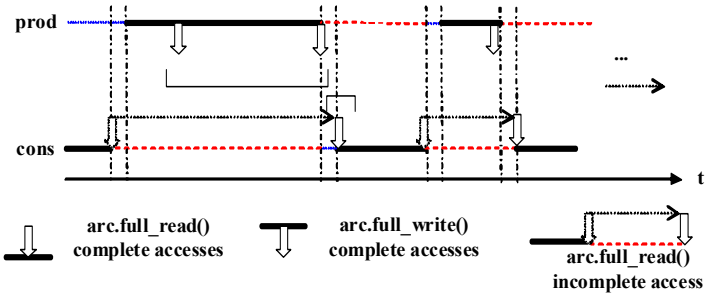



**Figura 2-61. Simulación de la especificación SDF de la Figura 2-59 con accesos *full_write* y *full_read*.**

El canal *uc_arc_seq* también implementa métodos para la inicialización o carga de unidades de datos en el canal antes del inicio de la simulación. Esto es necesario para resolver interdependencias que, de otro modo, causarían interbloqueos en grafos SDF cíclicos. Los métodos que implementa el canal *uc_arc_seq* son el método *init* y el método *full_init*. El primero permite la inyección de una unidad de datos en el canal. El segundo inyecta un número de unidades de datos igual a la tasa de consumo del canal.

Entre los chequeos de reglas que realiza el canal *uc_arc_seq* se encuentran los correspondientes a la limitación a un proceso productor y un proceso consumidor por instancia. También se detecta si un proceso accede como productor y consumidor a la misma instancia. Asimismo, también, es posible configurar los niveles de gravedad de las violaciones de reglas de especificación y el detalle de los informes correspondientes.

La librería *HetSC* provee medios para producir informes específicos del MoC SDF. Un informe provee el número de disparos de cada nodo. Un informe más detallado provee la secuencia precisa de disparos realizada. Con ella se puede obtener una secuencia de planificación estática válida para el grafo SDF. Los medios que *HetSC* provee para este análisis son una variable de configuración que controla la habilitación o no de estos análisis y las macros UC_SDF_NODE y UC_NODE_FIRED, que deben emplearse en el código de especificación. La macro UC_SDF_NODE sustituye a la macro SC_THREAD para la declaración del proceso SystemC que representa el nodo. Esta macro admite un nombre alternativo corto para simplificar el informe. La macro UC_NODE_FIRED se incluye en el código de cada nodo para contabilizar los disparos de cada nodo. Si se deshabilita el informe, estas macros no tienen ningún efecto, de modo que pueden aparecer en el código aunque no se usen.

#### *2.5.3.6 Herramienta para depuración de interbloqueo*

La librería *HetSC* provee una herramienta para el depurado de situaciones de interbloqueo, especialmente útil en MoCs atemporales. Para ello, el usuario sólo tiene que incluir las macros *CH_BEGIN*, *CH_END* al principio y final del código de cada proceso y la macro *CH_MONITOR* envolviendo la llamada a cada acceso bloqueante a canal. Este informe, se habilita o deshabilita mediante una variable de configuración. De este modo, si se habilita, se da un reporte del último acceso a canal realizado en cada proceso, cuando termina la simulación de la especificación. Si el informe no se habilita, las macros no tienen ningún efecto. De este modo estas macros pueden aparecer en el código aunque no se usen.

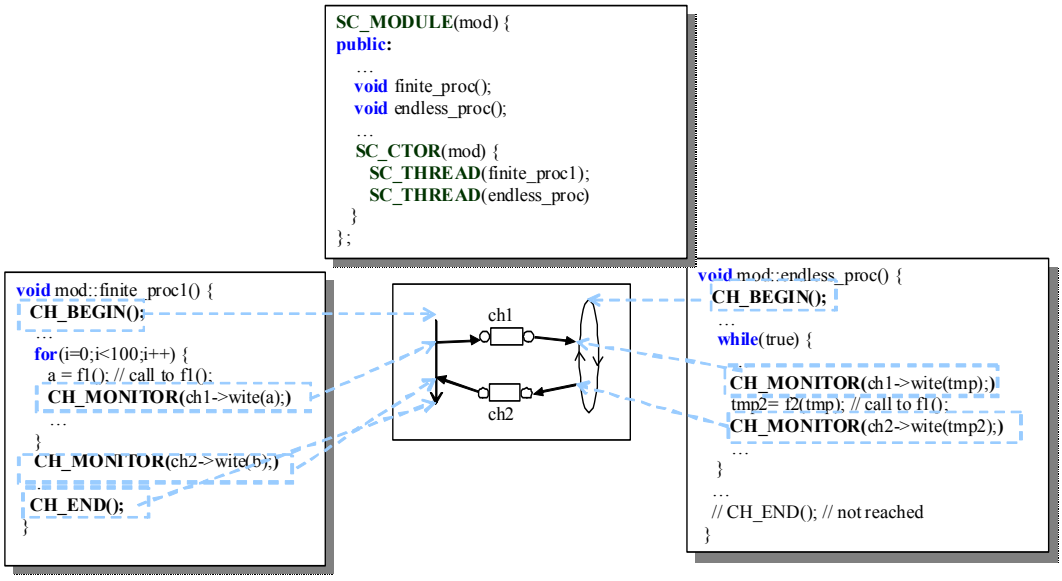



**Figura 2-62. Macros para depuración de situaciones de interbloqueo.**

## 2.5.4 Modelos de Computación Síncronos

Como se explicó en la sección 2.2.1, una especificación bajo un MoC síncrono maneja niveles de información temporal más detallados que los modelos atemporales, tales como la ranura temporal o el ciclo de reloj. Estos niveles tienen una noción de avance temporal según la cual los eventos del sistema se ordenan por su ocurrencia en una y sólo una de las ranuras o ciclos de la ejecución. *HetSC* soporta le especificación bajo estos MoCs sobre el kernel de simulación de eventos discretos y tiempo estricto de SystemC. Por lo tanto, *HetSC* define cómo la ranura temporal o el ciclo se interpreta en el eje temporal de SystemC. Para ello, proporciona una interpretación de sincronía considerando el manejo de tiempo de SystemC.

En un MoC síncrono de *HetSC* dos eventos son síncronos si:

$$t(e_i) = t(e_j)$$

o, denotado de forma equivalente:

$$t_{ei} = t_{ej}$$

Es decir, son síncronos si sus estampas temporales son iguales. Si se tiene en cuenta el valor de la estampa temporal, entonces, esta definición de sincronía tiene un matiz físico. Sin embargo, esta interpretación no es estrictamente necesaria. Nótese también que, según esta definición, es posible que dos eventos síncronos se sitúen en deltas distintos de la simulación. Es decir, pueden existir dos eventos $e_i$ y $e_j$ síncronos tales que $\delta_i \neq \delta_j$, es decir, $T(e_i) \neq T(e_j)$. Esta definición, es conveniente porque permite manejar relaciones de orden entre eventos dentro de la sincronía. Esto se hace patente en la sección siguiente, en la que se necesita interpretar la hipótesis de sincronía perfecta en SystemC. Las secciones siguientes mostrarán, asimismo, qué facilidades de especificación y reglas de uso de SystemC se emplean en *HetSC* para especificar bajo los MoCs síncronos mencionados.

### *2.5.4.1 MoC Síncrono Reactivo (SR)*

Una especificación bajo el MoC SR en *HetSC* consiste en una red de procesos comunicados a través de canales de tipo *uc_SR* (o *uc_buffer*). En la Figura 2-63, se presenta una especificación *HetSC* (omitiendo la información de jerarquía) que representa el sistema de la Figura 2-12.

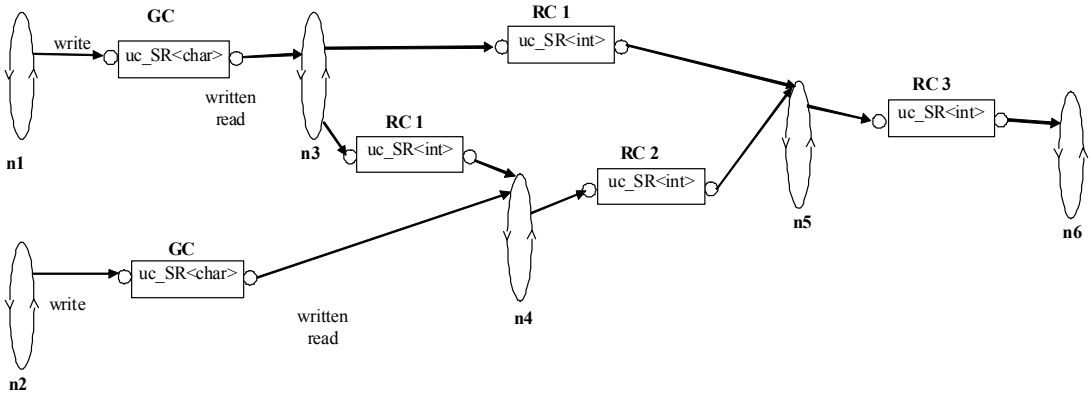



**Figura 2-63. Especificación en *HetSC* de la especificación SR de la Figura 2-12.**

El cómputo se encapsula de nuevo en procesos SystemC (SC_THREADs), que se rigen por las normas de continuidad y de relación funcional descritas para los MoCs atemporales. Adicionalmente se añaden reglas que consideran dos tipos de procesos: procesos *generadores o* GPs (como *n1* y *n2* en la Figura 2-63) y procesos *reactivos* (RPs). Los procesos generadores ejecutan autónomamente, siendo capaces de disparar los procesos reactivos. Los procesos reactivos son disparados por procesos generadores o por otros procesos reactivos a través de instancias de canal *uc_SR*.

El canal *uc_SR* (o *uc_buffer*) es un canal provisto por la librería *HetSC*. En términos sintácticos provee una interfaz como la de la señal estándar SystemC, añadiendo el método *written*, que es un alias para el método *event*, que sirve para saber si el canal fue escrito en el delta de simulación anterior. En términos semánticos es similar al canal estándar *sc_buffer* de SystemC, que se comporta como el canal *sc_signal* de SystemC, salvo por que los eventos se generan debido a los accesos de escritura, en lugar de por el cambio de valor. Por tanto, el canal *uc_SR* permite la transferencia de datos hacia los procesos reactivos. Se transfiere una unidad de datos de tipo *T* en cada disparo. Existe una diferencia semántica entre el canal *uc_SR* y el *sc_buffer*. Dependiendo de la situación de la instancia de canal *uc_SR* en la estructura de especificación, éste puede ser un canal generador (GC) o reactivo (RC). El canal *uc_SR* se configura como generador siempre que recibe su primer acceso de escritura en el primer delta de simulación de la ranura temporal. En caso contrario se configura como reactivo. Esto tiene implicaciones en los chequeos internos que realiza el canal. Estos chequeos, son otra diferencia importante entre el canal *uc_SR* y el canal *sc_buffer*.

El soporte del MoC SR en SystemC exige la interpretación de la hipótesis de sincronía perfecta, expuesta en la sección 2.2.1. Teniendo en cuenta la interpretación de sincronía en HetSC, expuesta al principio de la sección 2.5.4), la hipótesis de sincronía perfecta en *HetSC* consiste en que tanto el cómputo de los procesos reactivos como la comunicación entre procesos (a través de canales *uc_SR*) no implican un avance de la

estampa temporal SystemC (t). Por lo tanto, los eventos ($e_j$) asociados a los proceso reactivos son síncronos con los eventos ($e_i$) asociados a los cómputos autónomos que los han disparado ($t(e_i) = t(e_j)$).

Como entre los eventos causa $e_i$ y los eventos efecto $e_j$ puede existir un avance en deltas de simulación, *HetSC* ofrece una forma coherente de mapear en el eje temporal de SystemC los eventos con relaciones causales propias del encadenamiento de cómputos reactivos, $T(e_i) \leq T(e_j)$, a la vez que permite que estos eventos puedan ser síncronos $t(e_i) = t(e_j)$. Esto habilita también una conexión coherente entre MoCs atemporales y MoCs síncronos (como se verá en la sección 2.5.6.4). Por ejemplo, en la Figura 2-64, los cómputos C-1, C-2 y C-3 se encadenan, de tal forma que C1⇒C2⇒C3. Los tres ocurren en el la misma estampa temporal ($t_0$), aunque en distintos deltas de simulación. A efectos del MoC SR, estos tres eventos ocurren en la misma ranura ( Ranura 0 en la Figura 2-64).

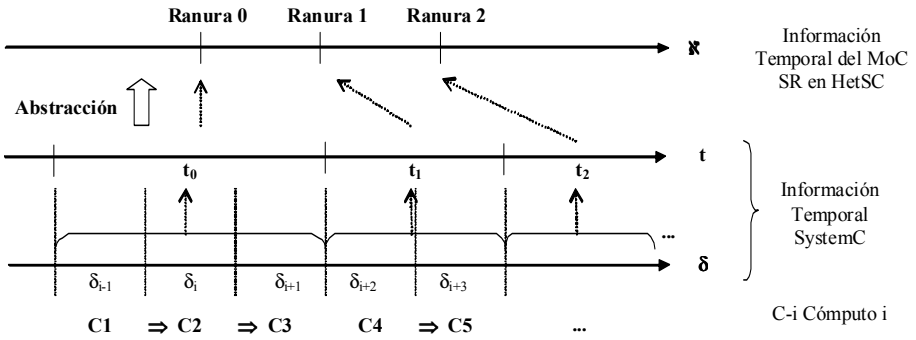



**Figura 2-64. Abstracción de la información temporal de SystemC en el MoC SR.**

Por tanto, aunque la estructura de la Figura 2-63 guarda cierta similitud con la del MoC SDF, existen diferencias. Por un lado, se utiliza un tipo diferente de canal y, por tanto, de semántica de comunicación. Por otro lado, los procesos generadores pueden presentar avances en la estampa temporal de SystemC. Más aún, estos avances deben estar presentes cuando el proceso generador realiza más de una acceso de escritura a un canal *uc_SR* y son los que determinan las diferentes ranuras temporales. El cómputo de los procesos reactivos puede consumir un número finito de deltas de simulación, en cambio no pueden suponer un avance de la estampa temporal.

Como se observa en la Figura 2-63, los procesos reactivos pueden disparar otros procesos reactivos a través de canales *uc_SR*. Esto hace posible el encadenamiento de cómputos reactivos para conformar una cadena reactiva. Los canales *uc_SR* suponen un avance en delta de simulación, de forma que el proceso disparado computa en el siguiente delta de simulación (δ). Sin embargo, no fuerzan un avance de la estampa temporal (t). Como resultado, la cadena reactiva implica un avance de deltas de simulación, pero no en la estampa temporal.

En la Figura 2-65 se muestra una simple cadena reactiva acíclica. En cada ranura temporal, GP1 dispara RP1, transfiriéndole una unidad de datos de tipo carácter. A su vez, RP1 dispara RP2, transfiriéndole una unidad de datos de tipo entero. RP2, a su vez, dispara RP3, sin transferirle dato alguno.

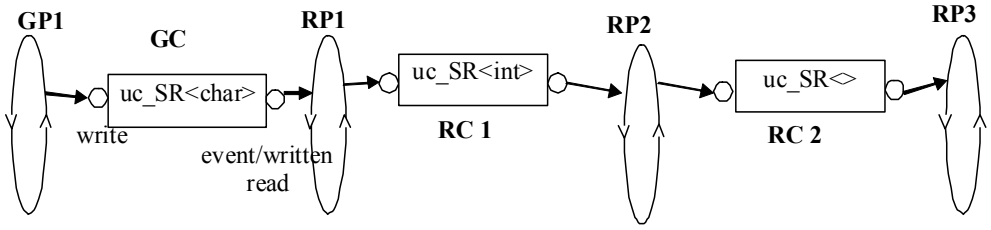



**Figura 2-65. Cadena reactiva sin realimentación ni reconvergencia.**

Diversas estructuras de concurrencia son posibles. La concurrencia de los procesos generadores habilita la concurrencia de diferentes cadenas reactivas. Un proceso generador puede disparar también varias cadenas reactivas sin dependencias entre sí, tal y como se muestra en la Figura 2-66.

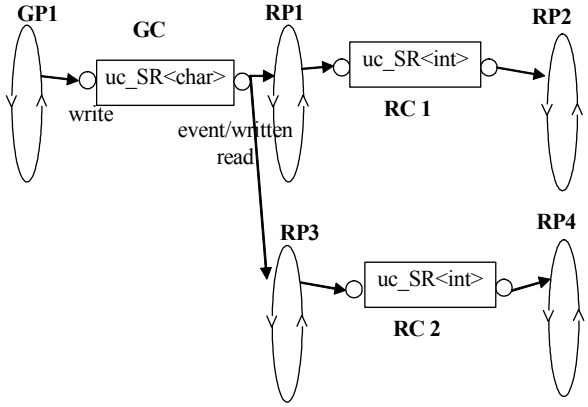



**Figura 2-66. Cadenas reactivas concurrentes.**

El MoC SR de *HetSC* requiere que haya un solo proceso escritor de la misma instancia de canal *uc_SR*. Como en otros canales *HetSC*, para detectar una violación de esta regla se implementa un chequeo dinámico. En cambio, como se puede observar en la estructura de la Figura 2-66, es posible que una instancia de canal *uc_SR* dispare más de un proceso y que cada proceso reactivo disparado lea de dicha instancia. El valor leído por cada proceso reactivo será el mismo durante el mismo delta de simulación y consiste en el valor escrito en el delta de simulación previo. El delta de simulación es el tiempo de simulación mínimo durante el cual persiste el valor leído, lo que permite una lectura no destructiva y, por tanto, lectura desde dos o más procesos sin que afecte al determinismo de la especificación.

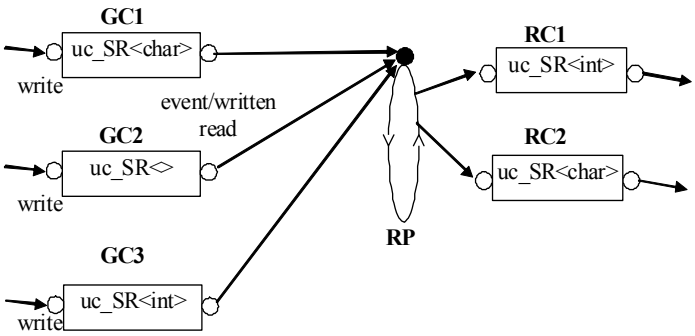



**Figura 2-67. Un proceso reactivo puede ser disparado por varias instancias de canal *uc*_SR.**

En la Figura 2-67, se representa como también es posible que varios canales *uc_SR* disparen el mismo proceso reactivo. Asimismo, también representa como un proceso reactivo puede escribir una o más instancias de canal *uc_SR*. Cualquiera de esas instancias de canal pueden ser escritas en cada disparo.

Los procesos reactivos (RPs) admiten diversos estilos de especificación en SystemC. Actualmente, *HetSC* provee varias posibilidades mediante SC_THREADs que soportan todos los chequeos actualmente disponibles para el MoC SR. A diferencia de los procesos generadores, los procesos reactivos tienen asociada una lista de sensibilidad. En *HetSC* se usa el operador "<<" para definir una lista de sensibilidad estática. Esta lista de sensibilidad recibe instancias de canal *uc_SR* a los que el proceso reactivo es sensible. También puede recibir puertos como parámetros, cuando la instancia de canal *uc_SR* se encuentra fuera del módulo en el que se inscribe el proceso reactivo.

En *HetSC* se soportan dos tipos de procesos reactivos, estrictos y no estrictos. Esta distinción cobra sentido cuando el proceso reactivo es disparado por dos o más instancias de canal *uc*_SR. Los procesos reactivos estrictos requieren el disparo en todas las entradas para ejecutarse. Los procesos reactivos no estrictos se disparan si al menos una entrada se ha disparado. Para especificar un proceso reactivo no estricto es preciso describir un bloqueo inicial, de forma que el proceso se desbloquee únicamente debido a un acceso de escritura a cualquiera de los canales *uc*_SR a los que es sensible el proceso. Para ello, la librería *HetSC* provee la macro *SR_WAIT_POINT*. Su semántica es similar a un *wait* de SystemC, pero adicionalmente introduce los chequeos del MoC SR. Posteriormente, es preciso decodificar la combinación de entradas que han provocado el disparo y realizar el cómputo en consecuencia, lo que puede implicar la escritura a otros canales *uc_SR* de salida. Esa decodificación se realiza con sentencias *if-else-if*, que chequean el disparo de cada canal mediante el método *griten* del canal *uc_SR*. La librería *HetSC* provee también la macro *SR_WAIT_SWITCH*, que permite una decodificación más compacta de la combinación de disparo a modo de cláusula *switch* de C. A continuación se muestra un ejemplo de codificación de proceso reactivo no estricto usando la macro *SR_WAIT_SWITCH* para la estructura de la Figura 2-67.

```
(1)   SC_MODULE(mod) { …
(2)     void RP() { // proceso reactivo no estricto
(3)       …// codigo inicializacion
(4)       while(true) {
(5)         SR_WAIT_SWITCH() {
(6)           case GC1.written():  // disparado sólo GC1
(7)             GP.write(result);
(8)             break;
(9)           case GC2.written():  // disparado sólo GC2
(10)            GP.write(past_result);
(11)            break;
(12)          case GC3.written():  // disparado sólo GC3
```

```
(13)              GP.write(2*GC3.read());
(14)              break;
(15)            case GC1.written()&&GC2.written():
(16)             // disparados GC1 y GC2 …
(17)            // otras combinaciones
(18)            …
(19)            case GC1.written()&&GC2.written()&&GC3.written():
(20)              // todos se han disparado
(21)              GC1.read(a);
(22)              b = GC1.read() + GC2.read();
(23)              result = compute_func(a, b, GC3.read());
(24)              RC1.write(result);
(25)              break;
(26)            case default: //combinacion de disparo erronea
(27)              …// accion de error
(28)        } // end if  } // end while loop
(29)     } // end process functionality
(30)     SC_CTOR(mod) {
(31)       SC_THREAD(RP);
(32)       sensitive << GC1;
(33)       sensitive << GC2;
(34)       sensitive << GC3;
(35)     }
(36)  };
```

Los procesos reactivos estrictos exigen que se hayan disparado todos los canales *uc_SR* a los que es sensible antes de realizar el cómputo. Este tipo de procesos permiten una codificación más compacta del cómputo reactivo que necesita leer datos actualizados en todos los canales de entrada. De ese modo, si a, b y c son los valores leídos por el proceso reactivo de la Figura 2-67, aunque los disparos en esos canales no lleguen necesariamente en el mismo delta, sólo se realizará un disparo. Por tanto, solo es preciso codificar una función de cómputo reactivo f(a,b,c), en lugar de 7 funciones: $f_1(a)$, $f_2(b)$, $f_3(c)$, $f_4(a,b)$, $f_5(b,c)$, $f_6(a,c)$ y $f_7(a,b,c)$. Para mantener el supuesto de sincronía perfecta, esto exige que los disparos, aunque en deltas distintos, aparezcan en la misma estampa temporal. Es posible la codificación de procesos estrictos usando el estilo mostrado antes para los no estrictos. No obstante, la librería *HetSC* provee la macro *SR_STRICT_WAIT_POINT* para la codificación compacta de este tipo de procesos. Ese estilo de codificación se explica también en la documentación de *HetSC*.

El MoC SR cuenta con varios chequeos de reglas. Ya se ha mencionado que algunos de ellos se repiten respecto a otros MoCs, como el de varios escritores, en tanto

que otros desaparecen, como el de varios lectores. Uno de los chequeos más importante, denominado *SAME_TIME_WRITE*, es el que detecta si ha habido más de una escritura del canal *uc_SR* en el mismo tiempo de simulación cuando éste es un canal generador. Este es el chequeo que obliga a los procesos generadores a separar cada escritura de dato en diferentes ranuras temporales. Si los canales no son generadores, entonces se chequea que éstos no sean escritos en el mismo delta de simulación. Otro chequeo importante es el que permite detectar si en algún punto de la especificación se ha violado la sincronía perfecta. El chequeo se puede configurar para que admita esa violación, pero que detecte si la reacción de alguna cadena reactiva excede el lapso de tiempo transcurrido hasta la siguiente ranura temporal que afecta a dicha cadena. De esta forma se habilita una versión relajada de MoC SR interesante en un proceso de refinado a partir de la especificación SR pura. Otro chequeo permite detectar si un proceso reactivo lee un valor antiguo, es decir, si se lee una instancia de canal que no ha sido responsable del disparo actual del proceso reactivo. Esto es interesante en la verificación de procesos reactivos no estrictos. También se habilita por defecto la detección de posibles faltas de reacción ante una combinación de disparo.

La librería *HetSC* también ofrece monitores e informes específicos para el MoC SR. Por ejemplo, se puede configurar la detección de que una reacción ha alcanzado un límite de deltas. Esta es una pista útil para la detección de condiciones en las cuales la especificación precisa una gran actividad para alcanzar un punto de estabilización en la ranura temporal. En ese sentido, también es posible activar un análisis de actividad. Una vez que se activa, el usuario obtiene tras la simulación un fichero que reportar el número de deltas por cada ranura temporal. Esto da una idea espacial y temporal aproximada de qué zonas de la especificación presentan más actividad.

#### *2.5.4.2 MoC Síncrono de Reloj (CS)*

Como se comentó en la sección 2.2.5.1, SystemC ofrece facilidades y guías de modelado en **[SYN02] [OSCI05]** para la descripción en un nivel de transferencia de registros (RTL) y de comportamiento para diseño de hardware síncrono sintetizable. Estas guías cumplen los requisitos de un MoC síncrono de reloj, sin embargo, esta documentación está enfocada en el dominio de diseño de hardware síncrono, además de ser demasiado extensa y poco metodológica. El objetivo de *HetSC* es proveer unas líneas más generales válidas para la especificación bajo el MoC CS en SystemC, capaz de integrar los trabajos de **[SYN02] [OSCI05]**. En *HetSC*, el MoC CS da una perspectiva más general que permite un modelado más abstracto en el dominio de datos y de computación a la vez que en el dominio temporal se asume la división del tiempo en ciclos. La comunicación entre procesos considera esta división.

Esta visión general del MoC CS, permite generar y verificar rápidamente modelos síncronos de reloj abstractos en SystemC y en pasos de refinado posteriores generar un modelo bajo este mismo MoC más elaborado e implementable. Por ejemplo, si se decide una implementación como hardware síncrono, se pueden extender las condiciones impuestas a la especificación por medio de las descritas en **[OSCI05].** De ese modo, se refinarán los tipos abstractos de datos, los lazos ilimitados, etc en tipos de nivel de bit, lazos limitados, etc. Sin embargo, el MoC CS de HetSC es lo suficientemente general para considerar otras opciones de implementación. De esta

forma, un modelo CS abstracto puede implementarse necesariamente también software. Por ejemplo, en **[Sir02]** se propone una técnica de implementación software (explicada en la sección 3.2.4.2 de este trabajo) que parte de una especificación SystemC bajo un MoC CS.

En *HetSC*, el MoC CS es considerado un caso particular de MoC SR, en el que a cada proceso SystemC se le asocia una única primitiva de disparo, el reloj. El reloj puede describirse como un proceso SystemC que genera los eventos generadores. Es la única fuente de disparo que define las ranuras temporales del sistema. Estas ranuras pasan a denominarse ciclos. El caso más habitual es que el reloj transfiera estos eventos al resto de procesos de la especificación a través de una instancia de canal *sc_signal*. Los procesos, en general, pueden ser SC_THREADs sensibles al evento asociado a esa esa instancia, a través de una lista de sensibilidad estática. En esa lista sólo aparece la instancia de canal *sc_signal* escrita por el reloj o el puerto correspondiente asociado a dicha instancia. En el caso del canal *sc_signal*, ese evento se asocia al cambio de valor en los accesos de escritura pertenecientes a deltas de simulación distintos. Por ello, el proceso reloj está obligado a provocar tales eventos en distintos deltas. Aún más, en *HetSC* el reloj debe forzarlos en distintas estampas temporales, para guardar coherencia con la interpretación de ranura en el MoC SR. Por otro lado, al igual que en el MoC SR, no se le atribuye ninguna noción física a la estampa temporal ni se exige una separación regular entre estampas consecutivas. Únicamente se toma el orden total entre las estampas temporales de cada ciclo.

La flexibilidad de SystemC para especificar bajo el MoC CS se manifiesta en otras formas compactas de obtener el mismo estilo de especificación. Por ejemplo, se puede utilizar la primitiva *sc_clock* para ahorrar la escritura del reloj como un proceso. Esta primitiva usa dos procesos internos, uno para generar flancos positivos y otro para los negativos. En un MoC CS pueden emplearse los flancos positivos, los negativos o ambos para determinar el ciclo. Asimismo, el reloj se puede transferir a través de otros canales, como el *sc_buffer*. Este canal tiene la misma interfaz que el canal *sc_signal* y puede disparar varios procesos SystemC. En el caso de usar canales *sc_buffer* para transferir el reloj, el proceso reloj sólo necesita realizar N escrituras (aunque sean del mismo valor) en estampas temporales diferentes para generar un número igual de ciclos. Algo análogo se puede decir acerca de las primitivas empleadas para la transferencia de datos entre procesos. Lo más habitual es el empleo del canal *sc_signal*, como en **[SYN02]**. No obstante, se pueden emplear instancias de canal *sc_buffer* de forma totalmente equivalente. Esto es gracias a que en el MoC CS la transferencia de datos entre procesos no emplea el evento interno del canal, único aspecto en el que se diferencian las semánticas de comunicación de los canales *sc_signal* y *sc_buffer*.

### 2.5.5 Modelos de Computación Temporales

En *HetSC*, una especificación bajo un MoC temporal fija la relación de orden existente entre todos los eventos del sistema. Entre estos MoCs se comprenden:

- Las especificaciones secuenciales.
- Las especificaciones bajo MoCs atemporales o síncronos anotadas en tiempo.
- Los MoCs analógicos.

Un ejemplo del primer caso, podría ser un algoritmo C/C++. Nótese que, en SystemC, esto puede ser el código de un sólo proceso o, simplemente, el del *sc_main*. No obstante, se precisan más condiciones. Una especificación de este tipo dejará de ser temporal en el momento en que la semántica que se le otorgue a la secuencia de sentencias permita un cambio de orden entre sentencias que tienen asociado un evento. Esto no es algo extraño, si se tiene en cuenta por ejemplo, un flujo de implementación SW. Entonces, un compilador puede cambiar de orden instrucciones que no tienen dependencias de datos entre ellos. Más aún, algo así lo puede hacer el compilador con el que se compila la especificación SystemC. Prohibir tales optimizaciones y asumir que el orden en el que aparecen en la especificación se ha de respetar en la simulación/ejecución sería un paso atrás. Por otro lado, establecer esas condiciones exige definir bien qué se entiende por evento en una especificación secuencial y en qué condiciones se preserva el orden. Esto forma parte de la formalización de la metodología y es objeto de estudio en **[AND08].**

Otra posibilidad relacionada que puede convertir un MoC en temporal es la anotación temporal de especificaciones temporales y/o síncronas. Este aspecto se relaciona con metodologías de perfilado de nivel de sistema, tanto si implican una anotación manual y/o explícita, como si esta es automática y/o implícita, tal y como realiza *Perfidy* **[HPSV03]**, Por anotación explícita, se entiende la introducción en el código del proceso de estamentos *wait* temporales, por ejemplo, *wait(sc_time)*. *HetSC* denomina a estas versiones temporizadas de los MoC mediante un prefijo adicional. Por ejemplo, si se introducen anotaciones en una especificación bajo el MoC KPN, se pasaría a tener un MoC T-KPN.

En la medida en que MoCs atemporales como KPN no haga dependientes sus propiedades de condiciones temporales, la introducción de anotaciones no afectará a las propiedades que estos proveen. Esta es una hipótesis de trabajo básica de *HetSC* que permite considerar que el modelo T-KPN ofrece las mismas propiedades que el modelo KPN al usuario. En este sentido, es preciso considerar qué propiedades garantiza cada MoC de forma independiente de la anotación temporal. Esa garantía, cumplidas las condiciones del MoC, debe darse en todo caso. Por ejemplo, una especificación bajo el MoC BKPN puede presentar en ciertos casos una condición de interbloqueo artificial o de sobresincronización que incluso rompa el determinismo de la especificación **[Jan04]**. La ejecución en un simulador específico puede ocultar ese problema, pero la ejecución en otro simulador o la anotación temporal pueden ponerlo de manifiesto. En relación con este trabajo, en **[HeVB06]** se han propuesto una extensión de la implementación del simulador de referencia de SystemC para mejorar la capacidad de manifestar esas situaciones sin necesidad de anotar en tiempo el MoC atemporal. Consiste en el soporte de aleatorización controlada en el planificador SystemC.

Respecto al tercer caso, *HetSC* puede cooperar con metodologías de modelado analógico basado en SystemC. Específicamente, la metodología y la librería *HetSC* proveen guías y facilidades respectivamente para una colaboración eficiente de *HetSC* con SystemC-AMS. Esto se explica en la sección 2.5.8.

## 2.5.6 Integración de MoCs

La metodología de especificación resuelve la integración de MoCs suavemente. En efecto, partes bajo diferentes MoCs se conectan a través de interfaces de MoCs, que, en *HetSC*, son canales y procesos SystemC, es decir, facilidades propias del mismo lenguaje de especificación.

### *2.5.6.1 Interfaces de MoCs: Procesos y Canales Frontera*

En *HetSC*, los procesos y los canales que comunican partes de la especificación bajo MoCs diferentes se denominan procesos frontera (BP) y canales frontera (BC) respectivamente. En la Figura 2-68 se representa esquemáticamente la idea.

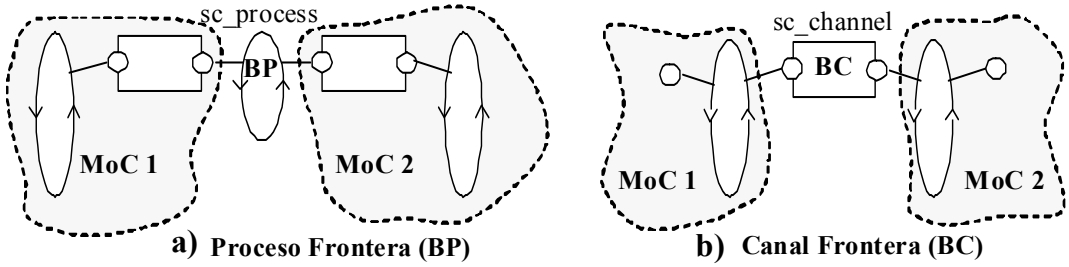



**Figura 2-68. Procesos y Canales Frontera**

Un proceso frontera o BP (Figura 2-68a) es un proceso SystemC que accede canales pertenecientes a partes descritas bajo MoCs distintos. El usuario tiene que especificar explícitamente en el cuerpo del BP el código de adaptación. Este código de adaptación es código SystemC secuencial que incluye accesos a canal. El proceso frontera es un mecanismo muy flexible que puede emplearse cuando el código de adaptación es simple o casi innecesario. También puede emplearse cuando se requiere una semántica de adaptación muy específica que no está comprendida por ningún canal frontera provisto por la librería *HetSC*.

Un canal frontera o BC (Figura 2-68b) es un canal SystemC en el cual se concentra la semántica de adaptación. Los procesos que comunica este canal pertenecen a partes descritas bajo MoCs distintos y, por tanto, en general, tienen un estilo de codificación distinto y propio del MoC. El canal frontera evita al usuario la implementación de la semántica de adaptación. Además, permite una separación explícita entre las partes bajo MoCs diferentes. Cada parte puede ser envuelta por un módulo y estos módulos pueden conectarse directamente mediante los canales frontera, aunque los módulos conectados presenten puertos con distintos tipos de interfaz. Esto es interesante también a la hora de conectar bloques provistos por una tercera parte, tales como los bloques de propiedad intelectual (IPs).

Al igual que con cualquier otro canal empleado en *HetSC*, el usuario sólo necesita conocer del BC su semántica y como instanciarlo y usarlo en la especificación. El principio básico de la semántica del canal frontera es que desde cada parte bajo un MoC determinado, el canal frontera se maneje como si fuera una instancia más del tipo de canal asociado al MoC. Por ejemplo, si una parte bajo el MoC KPN se conecta a un MoC SR, el canal frontera se maneja desde la parte KPN como si fuera una instancia más de canal *uc_inf_fifo*. El cumplimiento de este principio no es siempre posible por

completo. Esto es debido a las contradicciones que surgen al aplicar a la vez la semántica de comunicación y las restricciones impuestas por cada uno de los MoCs que conecta el canal frontera. En estos casos, el BC provee una solución a ese conflicto semántico. Un ejemplo de esto se verá en la siguiente sección.

El concepto de BC se puede entender como una generalización del *transactor* **[SCV03]**. En *HetSC*, el transactor se entiende como una familia de canales frontera capaces de conectar dos procesos de forma que, por un lado, proveen una interfaz funcional, basada en la llamada a un único método (tales como *write*(*data*) o *put(addr, data)*) para realizar la transacción completa, en tanto que, por otro lado, la comunicación se realiza a través de llamadas *write* y *read* a un conjunto de interfaces de señal SystemC (*sc_signal*). Cada una de esas interfaces de señal SystemC se corresponde con un canal señal SystemC que, a su vez, se integra en el transactor. Por tanto, la semántica de varias instancias de canal señal SystemC forma parte de la semántica global del transactor. Un transactor concreto debe todavía definir las relaciones entre la interfaz funcional y la de señales. En cuanto ese transactor específico define íntegramente una semántica de adaptación, éste se constituye como un canal frontera. Sin embargo, pueden existir canales frontera fuera del esquema de transactor, por ejemplo, los canales frontera para la conexión entre MoCs atemporales manejan únicamente interfaces funcionales.

### *2.5.6.2 Tipos de Interfaces de MoC*

*HetSC* establece una taxonomía de conexión de interfaces que organiza el soporte y desarrollo de facilidades para la conexión de MoCs, específicamente de canales frontera. Esta taxonomía se basa en la tipología de MoCs en **[Jan04],** que los clasifica en MoCs atemporales, síncronos y atemporales. A partir de ahí, *HetSC* clasifica las interfaces en atemporales-atemporales, atemporales-síncronas, atemporales-temporales, etc, tal y como se representa en la Figura 2-69.

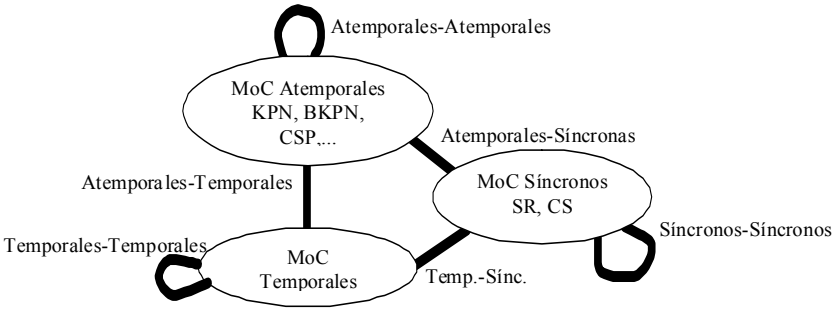



**Figura 2-69. Taxonomía de Interfaces de MoCs en *HetSC***

Actualmente *HetSC* soporta varios de estos tipos de interfaz de MoCs, entre los que se incluye la Atemporal-Atemporal, Atemporal-Síncronas, etc. La librería HetSC provee canales frontera específicos y aborda la codificación de los procesos frontera en estos casos. En algunos casos, como por ejemplo, las Atemporales-Temporales, el soporte de este tipo de interfaces de MoCs forma parte de los mecanismos provistos por *HetSC* para interoperar con la librería SystemC-AMS.

En el estudio de las interfaces de MoCs se han tenido en cuenta unas hipótesis básicas en la conexión de MoCs:

1. Cuando se conectan modelos con distintas condiciones, de forma que la intersección de condiciones no contiene las de ninguno de los modelos, en general, las propiedades de ambos modelos se ven afectadas. Cuando se conecta un modelo con condiciones más restrictivas que las de otro, de modo que las engloba, las propiedades del más restrictivo se verán, en general, afectadas, en tanto que las del menos restrictivo no. Un ejemplo de esto se da en la sección 2.5.6.3.

2. Los modelos que manejan un mayor nivel de detalle temporal transfieren ese nivel de detalle a los modelos menos detallados cuando se considera la ejecución de la especificación en términos globales. Esto se da, por ejemplo, cuando se conecta un MoC atemporal con un MoC reactivo síncrono (ver sección 2.5.6.4).

### *2.5.6.3 Interfaces Atemporales-Atemporales*

Una interfaz Atemporal-Atemporal permite la conexión en una misma especificación SystemC de dos partes bajo dos modelos atemporales distintos, por ejemplo, KPN con BKPN, KPN con CSP, SDF con PN, etc.

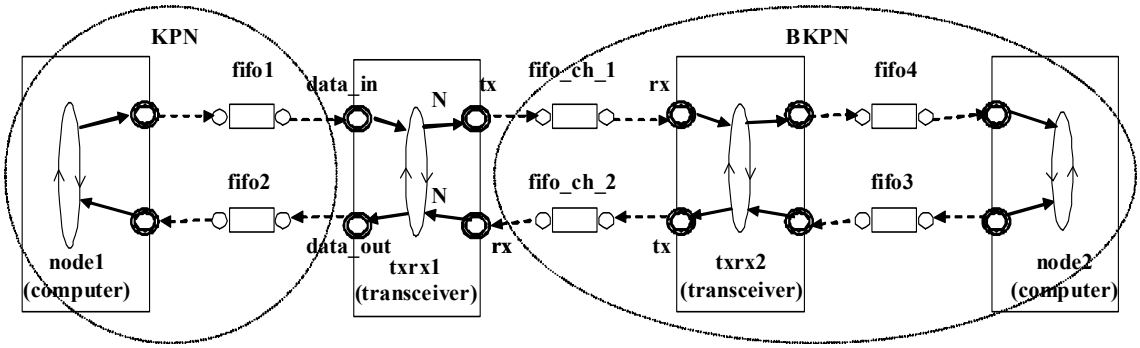



**Figura 2-70. Modelo en el que aparecen dos procesos frontera tras refinar canales.**

En estos casos, la conexión a través de procesos frontera puede ser conveniente, ya que apenas se precisa código de adaptación. Tómese como ejemplo la especificación de la Figura 2-70. Se trata de un modelo abstracto de dos nodos computadores comunicados a través de sendos transceptores. Los nodos son instancias de módulo de clase *computer* (*node1* y *node2*). Cada una contiene un proceso que genera inicialmente un chorro finito de datos y luego se queda indefinidamente leyendo datos. Los transceptores son instancias de módulo de clase *transceiver* (*txrx1* y *txrx2*), que también contienen un único proceso. Este proceso lee primero N unidades de datos de datos de la entrada *data_in*. Luego escribe esos N datos en la salida *tx*. Posteriormente lee N unidades de datos de la entrada *rx*, que transfiere íntegramente a la salida *data_out*. En una primera fase, esta especificación se escribe íntegramente bajo un modelo KPN. Eso quiere decir que todas las instancias de canal son de tipo *uc_inf_fifo*. Esta modelo abstracto, determinista y no presenta condiciones de interbloqueo.

Supóngase que, posteriormente, se realiza un refinado hacia un modelo más cercano a la implementación considerando la limitación de la capacidad de almacenamiento en los canales que comunican el computador *node2* con el transceptor *txrx2* (*fifo3* y *fifo4*) y los que realizan la comunicación entre transceptores (*fifo_ch_1* y *fifo_ch_2*). Para ello, se sustituyen esas instancias de canal, de tipo *uc_inf_fifo*, por instancias de canal tipo *uc_fifo*. De este modo, la parte derecha de la Figura 2-70 pasa a

estar bajo un MoC BKPN, en tanto que la parte de la izquierda sigue bajo las condiciones del MoC KPN, tal y como se ha ilustrado mediante las líneas punteadas. Así, el proceso interno del módulo *txrx1* se convierte en un proceso frontera que, en principio, no requiere transformación ni código de adaptación adicional.

Este ejemplo ejemplifica el primer principio enumerado al final de la sección anterior. Con la estructura explicada anteriormente de código de los módulos nodo y transceptor ocurrirá que siempre que el tamaño de ambas fifos, *fifo_ch_1* y *fifo_ch_2*, sea menor que N, se dará una condición de interbloqueo. En este caso en concreto, se trata de un bloqueo de sobresincronización o sincronización artificial (explicado en la sección 2.2.1), propio del MoC BKPN, no presente en el MoC KPN. Sin embargo, como se ha visto, al realizar el refinamiento y, por tanto, combinar los dos modelos, el KPN y el BKPN, la parte bajo el MoC BKPN presenta un interbloqueo que, en este caso se transmite y afecta a todo el sistema y, en consecuencia, también a la parte KPN.

Es posible encontrar condiciones adicionales para evitar ese interbloqueo tras el refinamiento y la consecuente conexión de MoCs atemporales. Una posibilidad es modificar la estructura de la especificación. En la Figura 2-71, se muestra una primera aproximación en la que los nodos se codifican con dos procesos. Esta solución sólo consigue que el interbloqueo sea parcial, en la medida que el proceso productor del computador *node1* puede continuar hasta el final (en tanto que *fifo1* es una fifo infinita).

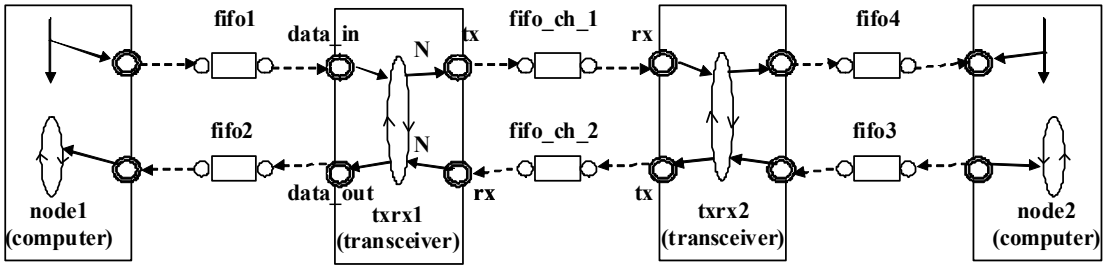



**Figura 2-71. La introducción de paralelismo elimina condiciones de interbloqueo.**

La solución que elimina el interbloqueo y, además es independiente de N, se presenta en la Figura 2-72. Ésta utiliza también dos procesos en el módulo transceptor.

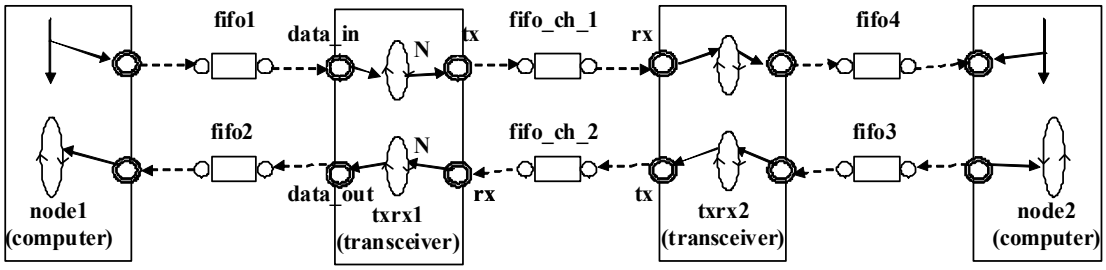



**Figura 2-72. Especificación que no requiere transformación tras el refinamiento KPN →BKPN.**

Finalmente, una solución que elimina el interbloqueo total y parcial sin modificar la estructura de la Figura 2-70 es que los procesos transceptores alternen cada acceso a canal, de unidad en unidad de datos (lectura de una unidad de datos de *data_in*, escritura de una unidad de datos en *tx*, lectura de una unidad de datos de *rx* y escritura de una unidad de datos en *data_out*), es decir, N=1.

En especificaciones grandes será usual que cada parte la desarrollen equipos o personas distintos, con lo cual, es pausible que cada equipo desarrolle módulos como los de la Figura 2-70 y que el posterior ensamblado resulte en un sistema con interbloqueo. Peor aún, no se verá claro cuál es el motivo del interbloqueo ni el bloque responsable del mismo, en especial difícil de juzgar si los módulos ensamblados ocultan la información de concurrencia y además no hay una concepción global del sistema.

Las alternativas libres de interbloqueo mencionadas antes se ven con claridad sólo mediante un estudio de nivel de sistema, que considera la estructura de concurrencia global de la especificación y los modelos bajo los que está cada parte del sistema. Éste estudio debe ser previo, por cuanto decisiones como cuantos procesos deben utilizarse para codificar el transceptor afectan al posterior desarrollo de ese módulo.

Específicamente, el ejemplo de la Figura 2-70 ilustra cómo, bajo ciertas condiciones, la combinación de MoCs puede estropear alguna propiedad beneficiosa del sistema cuando alguno de ellos no la garantiza. En cualquier caso, el empleo de procesos frontera es natural en este ejemplo y si, por ejemplo, se parte de la estructura de la Figura 2-72, el refinamiento de KPN a KPN-BKPN no exige modificación alguna. Para más detalles, estos ejemplos se encuentran disponibles en **[HSC08]**.

El uso de canales frontera en conexiones de MoCs atemporales también puede ser útil. Por ejemplo, la Figura 2-73 ilustra la conexión de dos módulos bajo distinto MoCs (BKPN y CSP). Cada uno de los módulos muestra puertas con interfaces propias de cada uno de los MoCs en *HetSC*. La instancia de canal frontera *bc*, de tipo *uc_fifo2rv*, es capaz de realizar directamente esta adaptación de nivel sintáctico, que implementa, por un lado, la interfaz bloqueante de escritura del canal *uc_fifo*, en tanto que, por otro lado, implementa la interfaz bloqueante de lectura de los canales rendez-vous de *HetSC*.

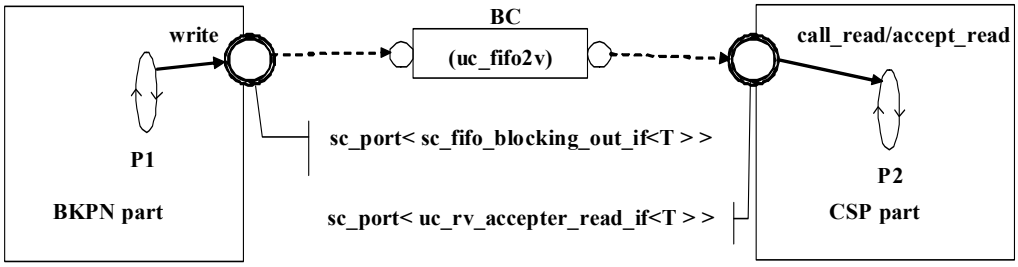



**Figura 2-73. Conexión de dos módulos bajo distintos MoCs atemporales mediante un canal frontera.**

Además de la adaptación sintáctica, los canales frontera realizan, en general, una adaptación semántica. En el caso de conexión entre MoCs atemporales, la cercanía de modelos puede simplificar la solución, pero, en general, ésta es aún necesaria. Por ejemplo, en el caso de la Figura 2-73, considérese la semántica del canal frontera *bc*. Esta es fruto de una intersección de las semánticas de acceso requeridas desde la parte BKPN y desde la parte CSP. En esa intersección semántica existe un conflicto. En efecto, desde la parte escritora BKPN se espera del BC que se bloquee cuando la fifo se llena. En cambio, el módulo CSP se ha codificado asumiendo la semántica fuertemente acoplada del canal rendez-vous, que implica que la parte escritora se bloquee en espera a la llegada del proceso lector, P1. Es decir, al proceso P1 se le piden dos semánticas de bloqueo diferentes.

Ante esta situación, se puede entender que la condición más restrictiva es que P1 se bloquee en ambos casos, tanto si la memoria intermedia interna del canal frontera está llena como si P2 aún no ha llegado a leer. Sin embargo, esto también puede estropear el funcionamiento del modelo si el bloque BKPN fue codificado asumiendo cierto tamaño en las fifos de salida. Es decir, cualquiera de las soluciones puede afectar a la otra parte del modelo, lo que ejemplifica el enunciado general de la primera hipótesis de la sección 2.5.6.2.

Ante esta situación, lo que se ha juzgado importante es proveer conexiones con soluciones de compromiso simples, y flexibles en lo posible, que automaticen y aceleren las tareas de adaptación sintáctica y semántica asociada a la conexión de MoCs. Así, en el caso de que la adaptación semántica del canal frontera ofrezca algún conflicto, la conexión ofrecida ofrece una solución de compromiso que debe ser conocida por el diseñador. Por ejemplo, en el caso de la Figura 2-73, el canal *uc_fifo2rv* tiene una memoria intermedia interna y hace primar la semántica de bloqueo en función del llenado de la memoria intermedia interna. De este modo, en términos semánticos el canal *uc_fifo2rv* es básicamente un canal *uc_fifo*, que, no obstante, implementa la interfaz de lectura de los canales rendezvous. Esto ilustra una solución simple en el sentido en que se opta por una semántica fija y conocida. En algunos casos, los canales frontera permiten al usuario optar por varias semánticas de adaptación, estableciendo siempre una por defecto.

#### *2.5.6.4 Interfaces Atemporales-Síncronas*

Una interfaz Atemporal-Síncrona es aquella que conecta una parte de la especificación bajo un MoC atemporal con otra parte bajo un MoC Síncrono. Es un tipo de conexión que implica la combinación de partes que manejan distintos niveles de detalle en el manejo de la información temporal. Esto hace que las consideraciones y adaptaciones a realizar sean más complejas. Asimismo, también ilustran la segunda hipótesis de la sección 2.5.6.2. En efecto, las consideraciones que hay que hacer acerca de la información temporal del modelo son globales y llegan al mayor nivel de detalle involucrado por los MoCs conectados, en ese caso bien de *ranura* o bien de *ciclo*. En cualquier caso, la conexión se realiza igualmente mediante procesos y canales frontera.

El ejemplo de la Figura 2-74 ilustra la conexión de una parte bajo el MoC CSP con otra bajo el MoC SR. La información de jerarquía se ha eliminado por simplicidad. En la parte CSP los procesos $P_1$ y $P_2$ se conectan mediante un canal rendezvous unidireccional que fuerza relaciones de orden parcial entre los eventos asociados a los procesos $P_1$ y $P_2$. Por ejemplo, asumiendo que el sentido de las flechas en los procesos representa el orden de ejecución, el evento $e_{2n}$ (valor leído en $P_2$ justo tras el acceso *rendezvous)* se ejecutará siempre posteriormente al evento $e_{1n}$ (valor escrito en $P_2$ justo antes del acceso *rendezvous)*, es decir $T(e_{1n}) < T(e_{2n})$. Por otro lado, los procesos $P_2$ y $P_3$ forman una cadena reactiva propia del MoC SR, en la que $P_2$ es el proceso generador y $P_3$ el proceso reactivo. La instancia de canal *uc_SR* fuerza a que $P_2$, como proceso generador, escriba eventos separados en estampas temporales diferentes, es decir, en *ranuras* diferentes. En cada una de esas ranuras, el proceso $P_2$ dispara, a través del canal *ch2*, el proceso $P_3$, que se ejecuta en un número finito de deltas dentro de la misma ranura temporal, lo que en el MoC SR de *HetSC* significa sincronía perfecta.

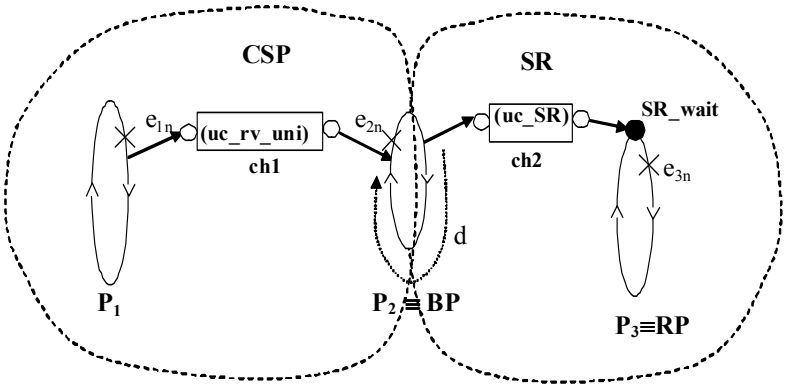



**Figura 2-74. Conexión de MoCs CSP y SR mediante un proceso frontera.**

Como se puede observar, P2 es un proceso frontera. En este proceso, el especificador debe codificar una adaptación en el dominio temporal. Consiste en imponer un avance temporal entre cada dos accesos consecutivos de escritura al canal *ch2,* de tipo *uc_SR*. De otro modo, como se ha dicho, el canal *ch2* reportaría un error en tiempo de ejecución de una regla de especificación del MoC SR.

Si se realiza un análisis local desde la parte bajo el MoC CSP, que emplea menor nivel de detalle temporal, la información de ranura temporal asociada a los eventos $e_{2n}$, $e_{2(n+1)}$, etc, necesaria para la parte SR, es irrelevante. Para la parte atemporal será relevante, en cambio, las relaciones de orden entre eventos, como $T(e_{1n}) < T(e_{1(n+1)})$, $T(e_{2n}) < T(e_{2(n+1)})$, $T(e_{1n}) < T(e_{2n})$, etc. No obstante, un análisis global y coherente de la temporización del sistema requiere manejar el nivel más detallado de información temporal, en ese caso el de ranura, que establecerá que, incluso los eventos de la parte atemporal estarán situados en alguna ranura de la simulación. De este modo, la parte SR añade a los eventos de la parte atemporal condiciones temporales adicionales y, en este caso, compatibles

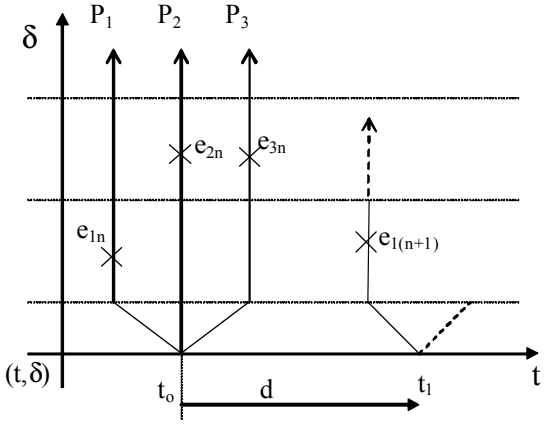



**Figura 2-75. Simulación de la especificación CSP-SR.**

La Figura 2-77 muestra como en la simulación el nivel de detalle de ranura se transfiere a la parte atemporal. Cada flecha vertical muestra el emplazamiento de los eventos de un proceso en el eje de deltas dentro de una misma estampa temporal. De esta forma, se puede observar que el avance temporal exigido por el MoC SR, afecta al MoC CSP. En efecto, inicialmente, cuando el modelo CSP estaba aislado, la condición

$T(e_{1n}) < T(e_{1(n+1)})$ se podía cumplir por una separación en deltas de simulación (siendo $te_{1n} = te_{1(n+1)}$). Sin embargo, tras la conexión con la parte SR, esta pareja de eventos aparecerá necesariamente separada en diferentes ranuras, y por tanto, en distintas estampas temporales $t_0$ y $t_1$ en la Figura 2-77). Es decir, a la condición $T(e_{1n}) < T(e_{1(n+1)})$ se le añade la condición $te_{1n} < te_{1(n+1)}$.

Como se ha dicho, en esta conexión, las condiciones temporales son compatibles, al menos en lo que se refiere a la preservación de las condiciones y propiedades de cada MoC. En efecto, por un lado, el retardo temporal en el proceso frontera separa las escrituras del canal *ch2* en ranuras diferentes, cumpliendo esa regla SR, exigida en la parte derecha de la especificación. Por otro lado, la parte bajo el MoC CSP en *HetSC* sigue preservando el orden parcial de eventos que garantiza el canal *ch1*.

La Figura 2-76 muestra un ejemplo de canal frontera encargado de realizar adaptaciones temporales propias de una interfaz atemporal-síncrona, en ese caso, correspondiente a una conexión de MoCs KPN-SR.

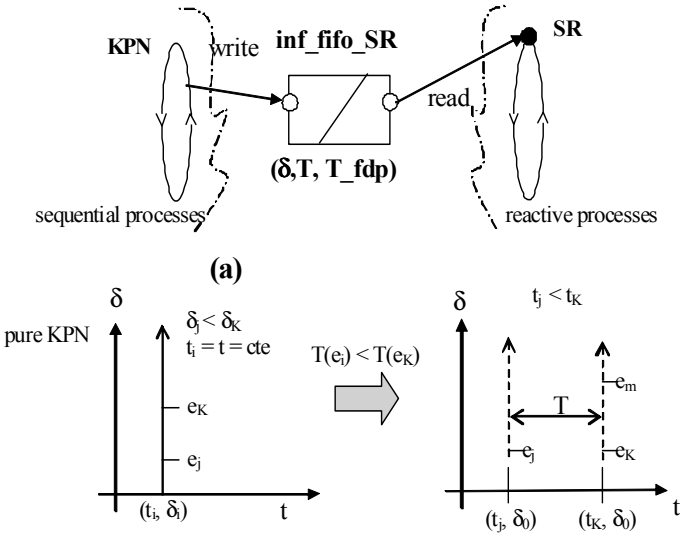



**Figura 2-76. Canales frontera en la conexión de MoCs KPN y SR.**

El canal frontera *inf_fifo_SR* se va escribiendo desde la parte KPN como si fuera un canal fifo no limitado (*uc_inf_fifo*). El canal frontera *inf_fifo_SR* se puede configurar de forma que introduzca un retardo temporal entre las unidades de datos que se van leyendo desde la parte SR. El canal provoca un evento tras cada retardo temporal, con el que es capaz de disparar el proceso reactivo de la parte SR. Por lo tanto, es capaz de actuar como un canal *uc_SR* de cara a la parte SR.

## 2.5.7 Compatibilidad TLM

En *HetSC*, los canales implementan por defecto interfaces estándar de SystemC. Por ejemplo, el canal *uc_fifo* implementa las interfaces *sc_fifo_blocking_in_if* y *sc_fifo_blocking_out_if*. Cuando no se encuentran interfaces SystemC estándar adecuadas, *HetSC* proporciona interfaces adicionales por medio de la librería *HetSC*. Por ejemplo, el canal *uc_rv* implementa la interfaz *uc_rv_if*, que a su vez hereda una serie de interfaces específicas para el MoC CSP (sección 2.5.3.4).

La librería *HetSC* se puede instalar para que algunos canales de la librería *HetSC*, sean accesibles también a través de interfaces básicas de TLM, es decir, estos canales implementan también interfaces TLM. La compatibilidad TLM facilita la aplicación de la metodología *HetSC* en entornos de modelado de plataformas en el nivel de transacción que se basen en el TLM de OSCI. Si no se instala la librería TLM, entonces la librería *HetSC* se configura para no implementar estas interfaces.

Así, por ejemplo, el canal fifo además de implementar la interfaz *sc_fifo_blocking_in_if*, implementa también la interfaz *tlm_get_if*. De este modo, una instancia de canal *uc_fifo* puede ser leída tanto mediante el método *read*, como a través del método *get*. La semántica de acceso es idéntica en ambos casos y la define el canal. La sintaxis se ajusta a SystemC estándar en el primer caso, en tanto que en el segundo se ajusta a sintaxis TLM.

Hay que tener en cuenta que cada método de acceso TLM posee una connotación semántica. Por ejemplo, el acceso *get* acarrea una semántica mínima de acceso:

- el acceso es bloqueante.
- el dato es transferido desde el canal hacia el proceso.
- la lectura del dato es destructiva (el dato se elimina del canal una vez es accedido).

Sin embargo, el método *get* de TLM no dicta más aspectos semánticos. El canal SystemC *uc_fifo* definido en la metodología *HetSC* y provisto por la librería *HetSC* es el que define e implementa la semántica de acceso y de comunicación con el detalle requerido por el MoC. Por ejemplo, el canal *uc_fifo* establece la condición específica de bloqueo cuando se llama a *get()*. Por tanto, la semántica de un método de acceso de una interfaz TLM implementada por un canal *HetSC* será siempre un subconjunto de la semántica completa del canal.

### 2.5.8 Interoperabilidad con SystemC-AMS

*HetSC* puede interoperar eficientemente con otras metodologías de especificación, específicamente con las que soportan MoC analógicos. De esta forma *HetSC* cubriría aquellos MoCs que pueden ser eficientemente soportados sobre un único kernel de simulación común, tales como los MoC atemporales y síncronos, en tanto que el soporte de MoCs analógicos se basaría en núcleos de simulación específicos, cuyo costo estaría más justificado para estos modelos de computación.

La posibilidad explorada ha sido la interoperabilidad con SystemC-AMS (sección 2.2.5.5). SystemC-AMS es una propuesta de carácter abierto y estándar OSCI, una propuesta temprana frente a otras como SystemC-A o SystemC-WMS, lo que ha facilitado esta exploración. Por otro lado SystemC-AMS, propone un mecanismo de sincronización con el núcleo de SystemC muy rápido, que, al igual que SystemC-A, evita la vuelta atrás, pero que admite solucionadores específicos, frente al solucionador de propósito general de SystemC-A. Además, como se explicó, SystemC-AMS no precisa la modificación del kernel de simulación de SystemC.

La posibilidad de interoperación de *HetSC* y SystemC-AMS se ha mostrado en **[HVG07] [CHVG08]**. La librería *HetSC* se depuró para permitir la instalación de

ambas librerías sin problemas de compatibilidad. La Figura 2-77 muestra la jerarquía de instalación de librerías y como se incluyen en la especificación.

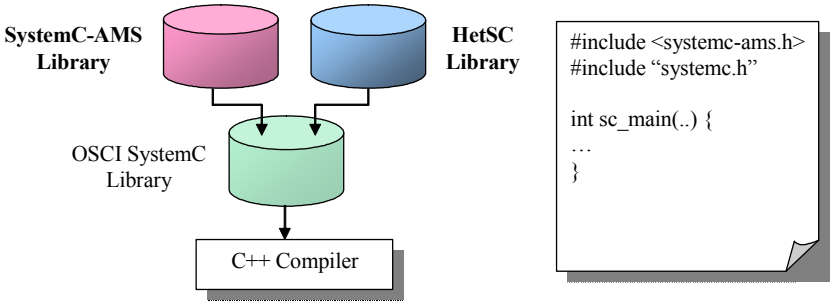



**Figura 2-77. Las librerías SystemC-AMS y *HetSC* se instalan sobre la librería SystemC.**

La Figura 2-78 muestra la cooperación doblemente eficiente de estas metodologías de especificación. Por un lado, se ofrece un soporte complementario de MoCs: SystemC-AMS para distintos tipos de modelos analógicos (LEN, de comportamiento, T-SDF estático, etc); *HetSC* para modelos atemporales (KPN, CSP, etc) y síncronos (SR y CS). Por otro lado, mientras que *HetSC* soporta sus MoCs directamente sobre el kernel de simulación de SystemC, SystemC-AMS provee ese soporte a través de una capa de sincronización y, sobre esta, los *solucionadores* específicos, más justificados para este tipo de MoCs. Ambas metodologías se soportan sobre el núcleo de simulación de SystemC.

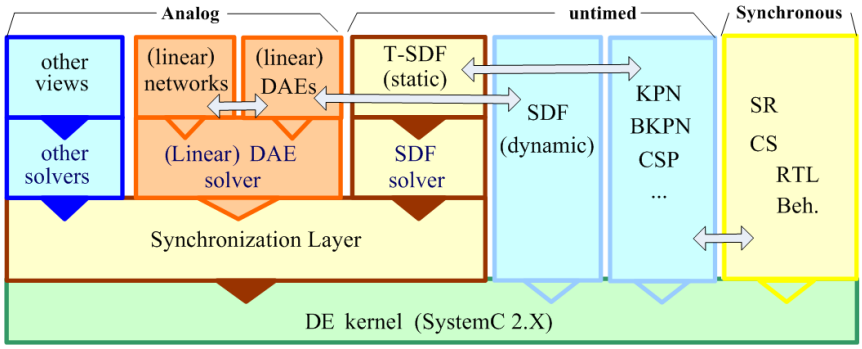



**Figura 2-78. Espectro de MoCs cubiertos por la cooperación de *HetSC* y SystemC-AMS.**

En la Figura 2-78 se ha representado también cómo, además de la conexión entre MoCs de cada metodología, es necesario estudiar y garantizar las conexiones entre los MoCs de las dos metodologías. De este modo, se han analizado ya diversos problemas sintácticos y semánticos que surgen en la conexión de *HetSC* y SystemC-AMS. Se ha encontrado que, para algunos casos, tales como el representado en la Figura 2-79, en el que una cadena reactiva del MoC SR de *HetSC* escribe datos hacia sendos módulos analógicos modelados mediante SystemC-AMS, las facilidades de especificación contempladas en ambas metodologías son suficientes para realizar una conexión coherente. En este caso, un proceso frontera (BP) de la cadena reactiva puede escribir directamente una señal, que será muestreada desde los *grupos* (*clusters*) SystemC-AMS, en un caso, a través de un puerto *sca_scsdf_in*, y en el otro, a través de una facilidad de conversión *sca_sc2r*.

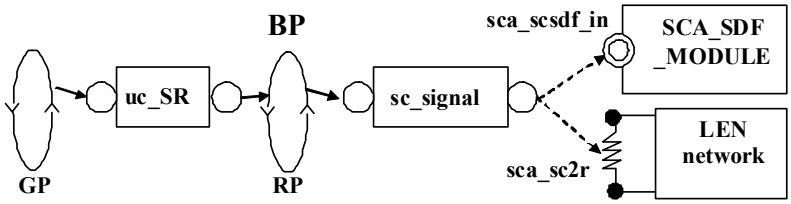



**Figura 2-79. Conexión de *HetSC* y SystemC-AMS por medio de un proceso frontera, una señal SystemC y facilidades de conversión de SystemC-AMS.**

En otros casos, se ha encontrado que el desarrollo de facilidades de conexión más específicas sería de utilidad. Se han propuesto soluciones basadas en el concepto de canal frontera de *HetSC* **[HVG07][CHVG08]**. En trabajos posteriores **[HDG07] [HDG08]**, se ha propuesto también el uso de una facilidad de conversión más potente, el *canal convertidor*. El *canal convertidor* surge de la mezcla de los conceptos de *canal frontera* y *canal polimórfico* **[Schr07]**. El canal frontera implementa una semántica de adaptación, usualmente en el dominio temporal, lo que agiliza la especificación multi-MoC (heterogeneidad horizontal). El canal polimórfico se basa en la capacidad de elección automática de una semántica de adaptación en función del entorno de interconexión, lo que agiliza el proceso de refinado (heterogeneidad vertical). Además de aunar estos conceptos, el canal convertidor provee características adicionales, tales como la conversión automática de los tipos de datos transferidos en el canal.

# Capítulo 3

# Metodología de Generación de Software Embebido *SWGen*

*Un requerimiento esencial para los flujos de diseño ESL basados en plataformas HW/SW es la aceleración de los flujos de implementación, tanto de hardware como de software. En este capítulo se presenta una metodología de generación de software embebido desde SystemC. La metodología toma como entrada código SystemC bajo la metodología HetSC, y produce código fuente y/o binario para una plataforma HW/SW embebida. Esta metodología presenta varias características distintivas. Se trata de una metodología de generación de fuente única, es decir, no requiere modificaciones del código de especificación, excepto una mínima información adicional de la partición HW/SW. La generación es automática y no requiere una traslación manual, de manera que se realiza con un esfuerzo mínimo (apenas editar un fichero y ejecutar un comando) y se eliminan representaciones intermedias, lo que elimina posibles fuentes de error. La metodología soporta diferentes APIs de RTOS, arquitecturas y entornos de desarrollo cruzado y plataformas HW/SW y es eficiente en velocidad y tamaño de huella. Asimismo, se integra junto con otras metodologías ESL del GIM-UC, como la de especificación heterogénea HetSC y la de perfilado PERFidy.*

## 3.1 Introducción

Debido al costo y a los riesgos asociados al desarrollo de soluciones hardware, un numero creciente de compañías están seleccionando plataformas hardware reconfigurables o bien programables mediante software **[San07]**. Desde su edición de 2001 **[ITRS01]** y aún más en la de 2007 **[ITRS07]**, el *ITRS*, señala que el diseño de software embebido ha emergido como el reto más crítico en la productividad en el diseño de SoCs. En concreto, hasta un 80% del costo en el desarrollo de sistemas embebidos puede recaer en el desarrollo del software. La Figura 3-1a) (tomada de **[Chou05]**) muestra la inversión en la proporción de esfuerzo dedicado a desarrollar software embebido y arquitecturas software, respecto al que se dedica a desarrollo hardware (diseño RTL, síntesis, etc). Más aún, la tendencia en los próximos años es a que el costo de la ingeniería y las herramientas de software embebido domine claramente al costo de la ingeniería y herramientas de hardware, tal y como representa la Figura 3-1b), adaptada de **[ITRS07]**.

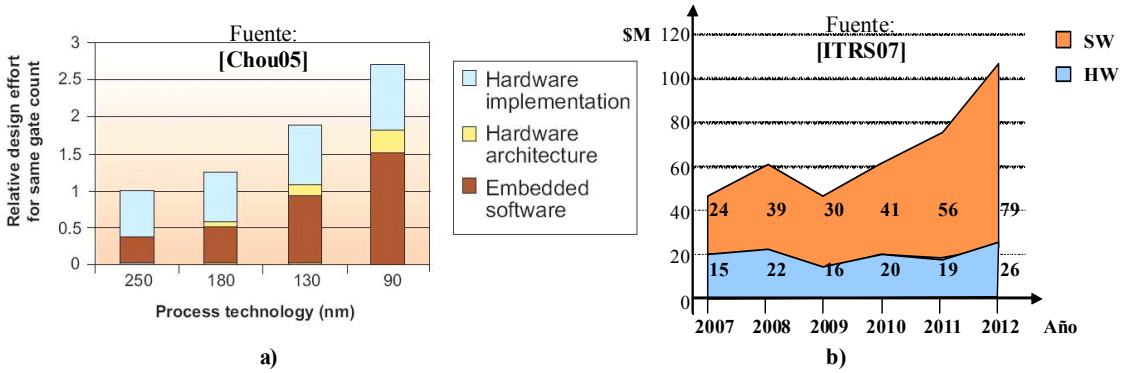



**Figura 3-1. Evolución pasada y futura de los costes de diseño software y hardware.**

En **[ITRS07]** se señalan las tendencias claves para conseguir una mejora de la productividad del diseño. Entre todas ellas, un número importante se refieren o afectan a la necesidad de una mejora apreciable en la productividad del desarrollo de software embebido. Por ejemplo, la automatización del flujo, la consideración de una verificación funcional y de rendimiento en el nivel de especificación, la consideración de una plataforma HW/SW objetivo y la cosíntesis HW/SW. En este capítulo, se presenta una metodología de generación de SW de nivel de sistema llamada SWGen que incorpora estas y otras características capaces de mejorar la productividad en el desarrollo de software embebido.

La metodología *SWGen* va más allá del estado del arte académico e industrial en el desarrollo de software embebido. Esta metodología toma como partida una especificación SystemC bajo la metodología *HetSC* y genera software embebido, como código fuente o ejecutable, para una plataforma HW/SW que incluye un sistema operativo embebido (eOS), de tiempo real (eRTOS o RTOS). El código generado incluye la parte SW correspondiente a la síntesis de interfaces. La generación es automática y requiere una mínima adición de información a la especificación. La generación no requiere la producción de representaciones intermedias y es eficiente en términos de tamaño de código y de velocidad.

## *3.2 Estado Del Arte*

### 3.2.1 Desarrollo Tradicional de Software Embebido (eSW)

#### *3.2.1.1 Plataforma HW/SW: Arquitectura SW y Arquitectura HW*

Desde 1970 los microprocesadores se han convertido en dispositivos comunes en los sistemas electrónicos embebidos. En este mercado han aparecido procesadores específicos, microcontroladores y procesadores de señal (o DSPs). De esta forma gran parte de la funcionalidad de los sistemas embebidos se realiza en SW **[Lee00]**. Para ello, los sistemas embebidos deben incluir un hardware básico, de propósito general: procesadores de propósito general, DSPs, distintos tipos de memoria (RAM, FLASH, etc), periféricos de entrada salida, etc, que se distingue de otro hardware específico (sensores, actuadores, aceleradores, etc. Este hardware de propósito general permite la ejecución de un conjunto de instrucciones máquina, que junto con el *mapa de memoria*, conforman la *arquitectura SW* del sistema, distinguible de la *arquitectura HW* o, simplemente, *arquitectura* del sistema. La arquitectura SW es una abstracción que permite al programador independizarse de los detalles de la arquitectura HW y, a la vez, que los desarrolladores de HW puedan diseñar nuevas arquitecturas más eficientes que reutilicen una gran parte del soporte software. Existen múltiples *arquitecturas SW* (ARM, MIPS, OpenRISC, etc) y muchas de ellas están protegidas con *derechos de uso*. Una plataforma HW/SW es un sistema que presenta una arquitectura HW con unidades de cómputo hardware (FPGAs, ASICs) y hardware de propósito general para soporte de la ejecución de software bajo una arquitectura SW determinada.

La implementación SW ofrece ventajas importantes **[Lee00]**. El programador usa el hardware de plataforma de forma transparente. A efectos funcionales, la plataforma específica pierde relevancia. El código ensamblador se puede portar a varias plataformas que se atienen a la arquitectura. También da una mayor flexibilidad, ya que es posible el cambio y la mejora de la funcionalidad del producto una vez que éste está ya en el mercado, sin requerir un cambio de la plataforma HW y, a efectos del usuario, del producto físico en sí. Ofrece también ventajas en el diseño, que se han ido heredando de los avances propios de la informática. Durante las décadas de los 70 y los 80, el software embebido, tanto el código de aplicación, como el kernel de ejecución te tiempo real, era usualmente escrito en C o en ensamblador por el usuario, y, finalmente, integrado en el hardware **[ShKe92] [BCGH97]**.

El uso y mejora de compiladores o depuradores ha elevado el nivel de abstracción y, por tanto, la productividad, portabilidad y flexibilidad. Sin embargo, las técnicas de diseño de software embebido, difieren en muchos puntos del desarrollo de software de propósito general para ordenador personal (PC). En los sistemas embebidos las prestaciones de rendimiento (consumo, potencia, velocidad, tamaño) pueden tener tanta o más relevancia que los requerimientos funcionales, el software tiene una fuerte interacción con el entorno y, a menudo, está desarrollado por ingenieros con poca experiencia en ciencias de computación **[Lee00]**, cuando, en ese contexto, las decisiones arquitecturales tanto hardware como software tienen consecuencias muy relevantes en el

rendimiento del sistema. Así, por ejemplo, una consideración importante en una metodología de generación de código embebido es la posibilidad de estimar a-priori con cierta exactitud el rendimiento del código. Para ello es típico el empleo de simuladores de nivel de instrucción (ISS) o de plataforma, aunque diversos trabajos han propuesto técnicas para acelerar estas estimas. Por ejemplo, en **[SmSe91]** se propone un sistema de síntesis en el que todas las primitivas para construir el programa están definidas como una secuencia fija de instrucciones. El tiempo de ejecución y el tamaño de código están precalculados, con lo cual, estas cifras se pueden usar para hacer predicciones precisas del rendimiento. Esta técnica presenta limitaciones de precisión cuando esas secuencias de instrucciones admiten optimizaciones en compilación o se consideran arquitecturas alternativas. En **[PHSV04]**, se propuso una metodología de perfilado de nivel de sistema que se basa en la metodología de especificación HetSC y que sirve de partida para la generación de software que se presentará más adelante en esta tesis.

#### *3.2.1.2 Desarrollo Cruzado*

Una peculiaridad básica del desarrollo de software embebido es que es cruzado. Es decir, la arquitectura de la plataforma nativa (en la que se desarrolla y compila el eSW) es, en general, distinta de la arquitectura de la plataforma objetivo (en la que se ejecutará el eSW). La razón principal es la limitación de las prestaciones de la plataforma objetivo, que no hace práctico, ni factible en muchos casos, compilar de forma nativa en ella. Por ello, se utilizan *compiladores cruzados*, que, ejecutándose en la plataforma de desarrollo o nativa, generan código máquina para la plataforma objetivo. Gran parte del esfuerzo en generación de código embebido tiene que ver con portar a nuevas plataformas objetivo estos compiladores. Los *compiladores portables*, es decir, compiladores permiten su adaptación a diferentes plataformas objetivo **[MaGo95]**. Un ejemplo importante de herramientas de desarrollo cruzado adaptables a diferentes plataformas objetivo son las utilidades binarias (*binutils*) **[Binu07]** y el compilador *gcc* **[GCC07]** de GNU. Estas herramientas soportan distintas arquitectura SW (ARM, MIPS, OpenRisC, Intel), y dentro de ellas distintas familias de procesadores (ARM7TDMI, ARM9, etc). Así, una configuración y compilación de estas herramientas cruzadas puede ser válida para una arquitectura SW y varios procesadores. El soporte del resto de la plataforma (control de periféricos, controladores de comunicación hardware/software, etc) ya suele recaer, en el sistema operativo embebido.

En el desarrollo de software embebido, es también habitual que el desarrollador se tenga que preocupar del *script* de enlace, que determina cómo la aplicación se mapea en la memoria del sistema. Es por ello, que el desarrollador de sistemas embebidos, frente al programador de software de propósito general, tiene que tener un mayor conocimiento de aspectos de la plataforma, tales como su mapa de memoria. Tras la compilación, el código embebido ha de ser descargado en la plataforma objetivo. El código es normalmente descargado y almacenado en una memoria de sólo lectura incrustada en el chip. Esta memoria suele ser costosa y, por tanto, de dimensiones reducidas. Por lo tanto, el tamaño del código puede afectar seriamente al costo del sistema. En esta línea, se han propuesto varias técnicas que reducen el tamaño de memoria consumida por el software embebido **[KCA01][PWL01]**. En ocasiones, este código se comprime (**[LHW00]**) y cuando necesita ser utilizado por la CPU se

descomprime en memoria de lectura/escritura, de menor costo y más abundante en el sistema.

La mayor complejidad de los procesadores embebidos hace que haya sido preciso ligar más el desarrollo de software con el de hardware a través de la consideración de elementos de la arquitectura, como las memorias cache. Su correcto diseño tiene un gran impacto en el rendimiento del sistema. El diseño de la memoria cache debe optimizar parámetros básicos, como el ancho de banda de la memoria, el tamaño de bloque, la profundidad, la política de reemplazo, etc. Se han propuesto varias técnicas para la mejora del uso de la memoria cache **[JDER01] [CJRD99]**. Estas técnicas definen la implementación de la cache además del soporte software necesario. Algunas técnicas pueden mejorar el tamaño de memoria con estructuras de datos reservadas dinámicamente **[EHMC00]**, incrementar el ancho de banda de la memoria **[Gebo01]**, y reducir el cálculo de direcciones **[GMCG01]**. El consumo es también una cuestión importante en los actuales SoCs y algunas técnicas de software permiten reducirlo **[FSCG00] [Shiu01]**. En cualquier caso, la flexibilidad y la reutilización de código siguen siendo factores cruciales en el desarrollo de software embebido. Conforme los sistemas embebidos han ido creciendo en complejidad, se ha hecho preciso considerar la inclusión de sistemas operativos embebidos (eOS) en el diseño de sistemas embebidos.

#### *3.2.1.3 Sistemas Operativos Embebidos (eOS)*

Al igual que los OS de propósito general, los eOS proporcionan servicios genéricos ampliamente utilizados por las aplicaciones embebidas. Por lo tanto, el eOS es un elemento de la plataforma que permite la abstracción, reutilización y portabilidad del software embebido. Esto es especialmente cierto cuando los eOS embebidos soportan una API genérica y estándar, como POSIX **[POSIX04]**. Adicionalmente especificaciones de API escalables, como EL/IX **[ELIX]**, que definen una interfaz estándar y adaptable al tamaño del sistema considerado, son interesantes en el contexto de los sistemas embebidos.

Al igual que con las herramientas binarias (*binutils*) y el compilador cruzado, los eOS presentan importantes diferencias frente a los OS de propósito general. La razón fundamental es, de nuevo, la fuerte especificidad y los estrictos requisitos de prestaciones y costo del sistema embebido frente al computador personal. Hay ejemplos de eOS comerciales **[VXW08],** abiertos **[Mas03]** o académicos **[MoBl02][VLM96]**.

Una primera diferencia es que los eOS están incrustados en el sistema y son a menudo transparentes para el usuario. El sistema embebido no suele provee un servicio de consola interactiva o similar de E/S, sino que sus servicios a la aplicación a través de la interfaz de programación (API). Entre los servicios requeridos por las aplicaciones embebidas a un eOS embebido se encuentran a menudo los de tiempo real (de ahí que habitualmente se hable de *Real Time* OS o RTOS embebido[4]), los de concurrencia, sincronización entre procesos, etc. Conforme los sistemas embebidos han aumentado en tamaño y complejidad, se han ido añadiendo servicios, tales como pilas de protocolos de comunicación (por ejemplo, TCP/IP), manejo de distintos tipos de soporte de memoria

[4] En este documento se usará indiferentemente RTOS y eOS, salvo que sea precisa la disquisición.

(por ejemplo, tarjetas de almacenamiento) o de entrada y salida (por ejemplo, un display táctil, conexiones USB, etc).

La necesidad de reducción del tamaño de código embebido hace que un parámetro fundamental del RTOS embebido sea la *huella*, es decir, el tamaño en memoria ocupado por el RTOS. La huella de un RTOS embebido es más reducida que la de un OS de propósito general. A principios de 2000 la huella de un sistema típico rondaba las decenas de Kbytes. En la actualidad no es raro encontrar sistemas embebidos con una huella de poco más de 1Mbyte, caso típico de algunas distribuciones de *Linux Embebido* (*ELinux*), que es perfectamente asumible por las memorias *flash* de 8 Mbytes o más que se encuentran en muchos móviles, cámaras digitales, etc. Una vez que el eOS ofrece los servicios necesarios, cuanto menor sea la huella, mayor cantidad de memoria se deja a la aplicación y, por tanto, para el valor añadido del sistema. En cualquier caso, considerado las mayores huellas de eOS, éstas son notablemente menores que la huella de 1Gbyte que puede rondar un OS de propósito general actual.

La gran diversidad de tamaños y necesidades de los sistemas embebidos da lugar a un requerimiento típico de los RTOS embebidos: la escalabilidad. Por ejemplo, *eCos* (*embedded Configurable OS*) **[Mas03]** es un RTOS embebido de libre distribución que provee una herramienta gráfica que permite la configuración de numerosos servicios y características del kernel, que van desde aspectos básicos, como la política de planificación, hasta servicios de comunicación, como el soporte de una pila TCP/IP o un acceso a bus CAN. Otros eOS como *ELinux* ofrecen también herramientas de configuración, La configuración del *eOS* permite ajustar los servicios y optimizar la huella. La huella y la capacidad de configuración del eOS determinan en gran medida sus dominios de aplicación.

Una característica típica de un eOS es también su arquitectura de implementación. Un eOS suficientemente complejo como *eCos*, soporta tres capas, consistiendo la central en el núcleo del eOS. Por encima, una capa superior provee una o más APIS de programación (por ejemplo, en el caso de *eCos*, las APIs µItron, POSIX y Cygnus-C). La capa inferior provee una o más particularizaciones de arquitectura y plataforma.

Todos estos elementos hacen que una característica distintiva de los eOS sea la metodología de desarrollo. El eOS formará parte, junto con el software de aplicación, del software del sistema embebido. Por tanto, el eOS afectará a las prestaciones del sistema, pero también a la estructura y al flujo de desarrollo del software embebido. En la Figura 3-2, se muestra un flujo típico de generación de software embebido incluyendo un eOS configurable. El eOS se incluye usualmente como una librería estática más. El código fuente (e*OS* en la Figura 3-2) puede estar escrito en un lenguaje de alto nivel, como C o C++, pero al menos una parte mínima suele estar escrita en ensamblador, para optimizar el rendimiento del eOS en las tareas más usuales y dependientes de arquitectura y/o plataforma. Una herramienta de configuración textual o gráfica (*Conftool* en la Figura 3-2), permite generar una configuración del RTOS para la aplicación (*eOS conf.* en la Figura 3-2). Adicionalmente, se ha representado una herramienta (*app2serv*) capaz de realizar un análisis sintáctico del código y extraer los servicios requeridos y, por tanto, generar la entrada de la herramienta de configuración, una plantilla de configuración (*conf. Tpl.*).

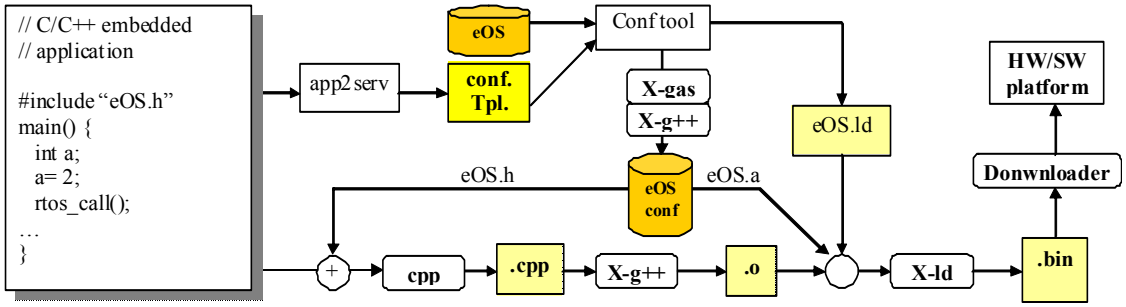



**Figura 3-2. Impacto de RTOS embebidos en el flujo de diseño de sistemas embebidos.**

La configuración del eOS (*eOS conf*) consiste en un fichero de cabecera (*eOS.h*) y otro objeto (*eOS.a*) que permiten incluir en la aplicación el conjunto de funcionalidades y servicios que ésta demanda al eOS. Para compilar la aplicación, ésta incluye el fichero de cabecera del eOS (*eOS.h*) y, por medio del compilador cruzado adecuado, se compila y enlaza con la librería objeto que contiene la configuración del eOS (*eOS.a*). Usualmente, la configuración del eOS también incluye una *script* de enlace (fichero *eOS.ld* en la Figura 3-2). Esta *script* es dependiente de la plataforma ya que informa al enlazador, entre otras cosas, de donde ubicar las distintas secciones (código, datos inicializados, datos no inicializados, la pila, etc) del binario ejecutable (fichero .bin en la Figura 3-2). De este modo, el enlazador une coherentemente el código de aplicación con el del eOS, ayudado por la información de la script de enlace. Finalmente, en este tipo de desarrollo, una aplicación de comunicación permite descargar el código en la memoria de la plataforma HW/SW para su ejecución. En esta categoría se comprenden, desde programadores de memoria flash, hasta versiones específicas de depuradores, como *gdb*, que permiten establecer un objetivo y descargar el código por alguna de las puertos de entrada de la plataforma objetivo (JTAG, puerto serie, paralelo, USB, Ethernet, etc) a través de alguna de las posibles salidas de la plataforma de desarrollo (puerto serie, paralelo, USB, etc). Usualmente, el cargador corre en la plataforma de desarrollo. No obstante, conforme las plataformas embebidas se hacen más complejas, algunas incluyen ya pequeños monitores con su programa cargador incorporado.

## 3.2.2 Generación de eSW en una metodología ESL

### *3.2.2.1 Evolución desde el Codiseño HW/SW*

Durante la década de los 90, la complejidad de los sistemas embebidos y la presión del tiempo a mercado se incrementaron de forma constante, haciendo inviable una metodología manual y basada en la experiencia de los desarrolladores. Como se comentó en la introducción, la metodología de implementación evolucionó hacia circuitos integrados cada vez más complejos, que permitieron la integración del sistema embebido completo en un chip (SoC). Esto implicó una relación aún más fuerte entre las particiones hardware y software, impulsando un diseño simultáneo, es decir, el co-diseño HW/SW. Ejemplo de herramientas clásicas de codiseño son Vulcan **[Gup95]**, Polis **[BCGH97]**, Cosyma **[StWo97]**, COMET, Castle, etc. En estos sistemas, la descripción de entrada se compila en modelos que describen el comportamiento del sistema. Originalmente, estas metodologías de diseño estaban basadas en una metodología de descripción y síntesis **[GaVN]**. Como se mostró en la Figura 1-4, en un

flujo de co-diseño clásico **[BCGH97]**, desde un lenguaje de especificación de alto nivel (HardwareC **[Gup95],** SpecCharts **[GaVN],** Esterel, etc **[ALB99]**,…) se sintetiza el código fuente C del software embebido. Después de que la descripción de entrada es realizada con alguno de los lenguajes antes mencionados, se realiza la partición hardware/software, teniendo en cuenta las restricciones del diseño y la estimación del rendimiento del sistema para dicha partición. Después de este paso, las síntesis de software comienza con la planificación de las tareas concurrentes asignadas a un recurso compartido: la CPU **[SLWS99]**. Dicha planificación puede ser estática (definida durante el proceso de síntesis de software) o dinámica (definida en tiempo de ejecución por un planificador dinámico, que puede ser parte de un eOS).

A finales de los 90, la metodología de "descripción y síntesis" presentada anteriormente no fue capaz de sobrevivir la revolución del SoC ya que no era capaz de cumplir las restricciones de tiempo a mercado. En **[CCHM99]** se propuso una metodología basada en el paradigma "especificación-exploración-refinado" **[GaVN94]**.

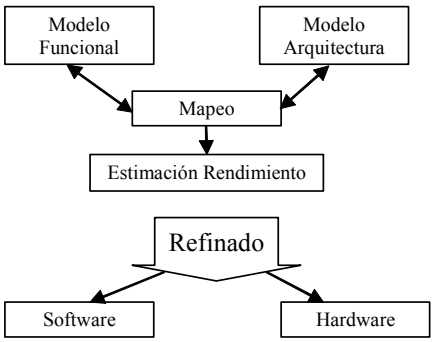



**Figura 3-3.** Metodología basada en Especificación-Exploración-Implementación.

La Figura 3-3 muestra el flujo de diseño de esa metodología **[KMNR00]**. El flujo de implementación parte de una especificación de nivel de sistema. La especificación de sistema de las metodologías "especificación-exploración-refinado" consiste en un conjunto de procesos y canales de comunicaciones, existiendo una clara distinción entre computo (procesos) y comunicación/sincronización (canales). La planificación de procesos es dinámica y normalmente controlada por una plataforma que cuenta con un eOS **[Sanc01]**.

En esta metodología la reutilización de diseños y las arquitecturas basadas en plataforma son aspectos claves del proceso de diseño. De esta forma, en **[San02]** la plataforma hardware se define como una familia de arquitecturas que satisfacen una serie de restricciones que permiten la reutilización de componentes hardware y software. Así, diversas herramientas de codiseño han presentado esta capacidad de reutilización en la generación de software, tales como el entorno *Cierto Virtual Component Co-Design* (Cierto VCC) **[Cie08]** de *Cadence* **[CAD08]**, o Coware *Virtual Platform* **[CowV08]** de *Coware* **[Cow08]**. La idea básica es la de asignar las funciones a ser implementadas a los componentes de la librería o a librerías de bloques de propiedad intelectual (IPs). Normalmente, el comportamiento del sistema viene fijado, pero algunas metodologías permiten la implementación de comportamientos diferentes (sistemas adaptativos) **[ERHT02]**. En definitiva, el mapeo se convierte en un proceso iterativo en el cual las herramientas de evaluación de rendimiento evalúan el

rendimiento de cada mapeo arquitectural y guían la siguiente asignación. Tras el mapeo, la partición hardware/software queda definida y se sigue el flujo de implementación detallado en la Figura 1-4.

#### *3.2.2.2 Modelos de Computación en la Generación de eSW*

Como se ha justificado en secciones anteriores de este trabajo, el soporte de heterogeneidad es necesario en una metodología ESL avanzada. Un aspecto que los trabajos mencionados en la sección 3.2.2.1 han puesto de relieve es la fuerte relación entre los modelos de computación y las posibilidades de optimizar la técnica de generación de software. Ciertos MoCs permiten tratar más adecuadamente en la generación de software aspectos como el control, el camino de datos y la concurrencia. Así mientras que algunos trabajos se centran en la minimización del tamaño de código, otros trabajos, relacionados con la aplicación de MoCs específicos, maximizan la velocidad del software generado.

En **[LHG98]**, se distinguen los modelos orientados a estados y los modelos orientados a actividad. Los modelos orientados a estados usan un conjunto de estados, transiciones entre estados y acciones asociados a los estados y a las transiciones para modelar el sistema. Ejemplos de esta categoría son las CFSM **[BCGH97]**, Redes de Petri **[SLWS99]**, y representaciones HCFSM **[GaVN94]**. De cara a la síntesis de software, algunas de esas representaciones (fuertemente orientadas a hardware), necesitan ser refinadas. Por ejemplo, las CFSM tienen que ser implementadas en un grafo S (grafo software) antes de la generación de software **[SuSa96]**. Otras representaciones (por ejemplo, redes de Petri) minimizan este problema. En los modelos orientados a actividad, el sistema se describe como un conjunto de operaciones relacionadas por dependencias de ejecución o de datos. Estos modelos son comúnmente usados en aplicaciones de procesado digital de señal (DSP). Ejemplo de estos modelos son los grafos de flujo de datos y de flujo de control **[GaVN94]**.

Un trabajo que reduce al máximo la ralentización por cambio de contexto en el software generado se encuentra en **[BML96]**. Esta mejora es posible gracias a que en el nivel de especificación se emplea un MoC SDF, que permite obtener una planificación estática de los procesos del sistema. En este trabajo se presentan de forma extensiva y rigurosa métodos de generación eficiente, incidiendo fundamentalmente en la minimización de código de software (toda vez que no hay cambios de contextos). El objetivo es fundamentalmente la generación de software para DSPs. El marco de especificación de este trabajo es Ptolemy, un entorno clásico para el modelado de sistemas heterogéneos. Como se reseñó en la sección 2.2.3, pese a que en Ptolemy hay una vocación expresa para el modelado, hay también una orientación hacia la síntesis de software embebido. En **[PHLB95]** se introducen los aspectos de síntesis de Ptolemy. Se presenta un entorno modular y extensible que responde a la necesidad de adaptar a cada MoC un método de generación de software para conseguir resultados eficientes. No obstante, el desarrollo en el tema de generación de software está aún limitado a ciertos modelos de computación, como SDF. Desde la última versión (6.0) se ha añadido la capacidad de síntesis de código C desde los MoCs SDF, FSM y HDF **[BLLN07]**. Ptolemy aporta además la infraestructura *codegen*. La generación de código se realiza a través de *helpers*. Los *helpers* son componentes que generan código por cada actor

Ptolemy II, que contienen porciones incompletas de código objetivo (C, Java, VHDL, etc). Existe un *helper* por cada lenguaje objetivo soportado. Un actor puede tener asociados más de un *helper* (uno por cada lenguaje soportado). Solo existe un subconjunto de actores que poseen estos *helpers* de manera que solo un conjunto de modelos pueden ser convertidos al lenguaje objetivo. A la infraestructura *codegen*, se añade el paquete *Copernicus*, que permite construir generadores de código para los modelos Ptolemy II a través del análisis del código *byte* de Java. Uno de los problemas de estos métodos de síntesis de software proviene de las condiciones impuestas por el modelo de flujos de datos (SDF), tales como el no realizar entradas o salidas condicionadas. En caso contrario, la planificación estática no sería posible.

En **[LinA98][LinB98]**, se presenta una técnica, implementada en una herramienta denominada *Picasso*, capaz de sintetizar código C plano para una plataforma monoprocesador desde una especificación concurrente. La sintaxis de la especificación es C. Esa especificación admite una composición jerárquica de tareas (creadas dinámicamente con un estamento tipo *par*, análogo al de SpecC) y la comunicación es unidireccional punto-punto con una sincronización tipo rendez-vous, por tanto, siguiendo un modelo de computación CSP. La generación se basa en la producción de una representación intermedia que sigue el MoC Petri-Net. Por lo tanto, existe una metodología de transformación de MoCs (heterogeneidad vertical) en el proceso de generación. Esta representación intermedia pone de manifiesto las relaciones de orden parcial entre los eventos de los procesos comunicados. Esta información se usa para calcular una planificación estática, que elimine la necesidad de un planificador dinámico, usualmente asociado a un eOS. En estos trabajos se reportan ganancias de velocidad de hasta dos órdenes de magnitud respecto a implementaciones basadas en librerías de hilos. La limitación principal consiste en que sólo se puede aplicar a sistemas cerrados con comunicaciones de tasa única.

En **[SLWS99]** se usa la variante del MoC Redes de Petri, denominado Redes de Petri de libre elección (Free Choice Petri Nets o FCPN) para la síntesis de una implementación de código C concurrente en un solo procesador. La ventaja de esta aproximación es que es capaz de sintetizar software para algoritmos que contengan estructuras dependientes de control (tipo sentencias *if-else-if* o lazos con condición de salida), para los cuales, las decisiones de planificación se tienen que realizar en tiempo de ejecución. Ello se resuelve con planificación cuasi-estática (*QSS scheduling*). Es decir, esta aproximación es mixta en cuanto a planificación (estática y dinámica). La planificación dinámica se realiza únicamente para las estructuras de control dependientes de datos, minimizando, por tanto, el número de decisiones de planificación en tiempo de ejecución. Para el resto del código la planificación es estática y determinable en tiempo de compilación.

En **[CKLM00]** se provee una solución para generar software desde una especificación multiproceso bajo el MoC Petri-Nets, para una plataforma de un solo procesador. Este trabajo está relacionado con el de **[SLWS99],** especialmente en cuanto a que parte de un MoC Petri-Net. Sin embargo, la especificación en **[SLWS99]** no puede manejar múltiples escrituras/lecturas de o hacia el mismo canal desde el mismo proceso ni tampoco admite sincronización dependiente de control en múltiples puertos. También mejora la limitación de **[STZE99]**, trabajo que también emplea redes de Petri

y admite especificaciones con control dependiente de datos pero que, en cambio, requiere hacer explícito los tamaños máximos de cada canal de comunicación. En **[CKLM00]** es posible tanto tamaños de canal definidos por el usuario, como canales no limitados, cuyo tamaño es determinado por la técnica. Sin embardo, en **[CKLM00]** se consideran aún falsos caminos que pueden hacer que algunas especificaciones sean decididas como no planificables cuando sí lo son. Esto es debido a que la forma genérica en la que se abstraen las dependencias de datos, sin considerar las correlaciones entre los controles dependientes de datos. En **[ADLP02]** se propone un algoritmo semiautomático para solucionar este problema. Este algoritmo puede usar como información de entrada el conjunto de controles dependientes de datos correlados que el usuario haya sido capaz de identificar en la especificación. En **[CKLP05]**, se propone una solución heurística (exacta para algunas clases de especificaciones) para determinar los límites que permiten decidir la planificabilidad del grafo. Todas estas técnicas, como otras muchas de las presentadas aquí se proponen para una arquitectura monoprocesador. En **[CKLT04]**, se da una extensión de las técnicas QSS aplicable a arquitecturas de implementación concurrentes.

Existen también trabajos para la síntesis de software a partir de especificaciones descritas bajo modelos síncronos. El compilador *V3* de Esterel **[Bego92]** usa una técnica basada en autómatas, entretejiendo los hilos que se están ejecutando en un momento dado para producir un código sin sobrecarga. El código generado puede crecer exponencialmente. En **[CGJL95]** se trata de atajar este problema reduciendo el tamaño de código compartiendo código entre estados. Para ello, usa árboles de decisión binarios para encontrar subárboles comunes. Sin embargo, el peor caso sigue siendo exponencial. En el compilador *V5* de Esterel, se opta en cambio por transformar cada estamento en un hilo de ejecución. El código que se genera consiste en una serie de instrucciones concurrentes del tipo "if(instrucción_ejecutándose) ejecuta_instrucción", donde instrucción_ejecutándose puede ser *true* o *false*. En **[Edw00]** se propone el compilador de Esterel *EC*, que permite generar código más pequeño y un orden de magnitud más rápido que el de *V5*, aunque un 50% más lento que las soluciones basadas en autómatas. En general, la técnica propuesta genera un código secuencial a partir del grafo de flujo de control concurrente usado como representación intermedia.

Como se ha visto, el término "síntesis de software" a menudo se reserva para optimizaciones en la planificación de las tareas concurrentes (usualmente pasando de dinámica a estática) buscando el aumento de la velocidad de ejecución y la reducción del uso de memoria **[SLWS99] [JiBr02]**. Ejemplos de este tipo son también algunos de los trabajos en síntesis de software dentro del marco de Ptolemy **[LeMe87] [PHLB95]**.

Una desventaja importante de muchos de estos marcos es la carencia de una metodología de reutilización de código. Además, la modificación de la descripción de entrada genera dificultades de depurado del software. Tal y como se señala en **[YDG04]**, las metodologías de síntesis de software en estos marcos son específicas y no pueden ser extendidas a otros marcos más generales. Por ejemplo, si el código embebido tiene que soportar concurrencia y los procesos cambian dinámica e imprevisible sus tasas de consumo y producción de datos, muchas de las técnicas de generación de software anteriormente expuestas no son aplicables.

Por eso, en cualquier caso, en una metodología que soporte especificación heterogénea y partes bajo múltiples MoCs, es deseable que exista un método de generación de software común, válido independientemente del MoC al que se ciñe cada parte del sistema. Esto garantiza la posibilidad de generación de software y no impide que se puedan aplicar técnicas optimizadas y adaptadas a aquellas partes que puedan ceñirse a los supuestos de un MoC. La metodología de generación de software propuesta en este trabajo será general en este sentido.

#### *3.2.2.3 Generación de eSW desde SLDLs*

Como se ha explicado, en una metodología ESL, el diseño se centra en la especificación. También se ha explicado la importancia de los lenguajes de descripción de nivel de sistema (SLDL) en esta tarea. Una necesidad patente en los flujos de generación de software en metodologías de diseño ESL es la posibilidad de generar código directamente desde la especificación de sistema, descrita mediante un SLDL.

La generación desde SLDLs y la aplicación de técnicas de generación SW optimizadas según el MoC (introducidas en la sección anterior) es totalmente compatible. De hecho algunos SLDLs responden a MoCs específicos, por ejemplo, *Esterel* al MoC *SR* o *Embedded Matlab* a un MoC de flujo de datos.

Algunos trabajos reflejan el intento de utilizar lenguajes ampliamente extendidos en el modelado de sistemas. Por ejemplo, *Embedded Matlab* **[EML07]** es un subconjunto del lenguaje Matlab que soporta la generación de código fuente C para el desarrollo de sistemas embebidos y la aceleración de algoritmos de punto fijo. Se puede usar también desde *Simulink*, por tanto, desde un estilo de especificación típicamente de flujo de datos o procesado de señal. De esta forma, se propone el desarrollo y optimización de algoritmos desde el lenguaje *Matlab* o desde *Simulink* y mantener una versión software coherente generándola automáticamente con la herramienta *Real-Time Workshop*. Esta herramienta es capaz de convertir el código dinámicamente tipado de Matlab en el código estáticamente tipado de C sin usar reserva dinámica de memoria. Para convertir los tipos de datos de forma precisa, Embedded Matlab requiere la definición de la clase, tamaño y complejidad de los datos en el código fuente, de forma que pueda asignar los tipos de datos de forma correcta en tiempo de compilación. Una limitación de *Embedded Matlab* es que el código generado carece de concurrencia.

Otra propuesta realizada en diversos trabajos, por ejemplo en **[LMS03]**, consiste en tomar UML (Unified Modeling Language) como punto de partida de la metodología de diseño. Esta propuesta es más complementaria que sustitutiva de las metodologías de generación existentes. El uso de UML proporciona una serie de beneficios adicionales. Entre ellos, un modelo unificado de descripción y diseño del sistema y una disposición hacia la captura gráfica, a la que dotaría de un mayor grado de unificación. Sin embargo, UML, carece de una definición semántica, lo que, entre otras cosas, impide que sea ejecutable. Por ello, aparecen propuestas en las que UML se complementa con algún SLDL dotado de una definición semántica y empleado como un lenguaje de acción. Esta posibilidad de cooperación de SDLs no es nueva. En **[Jon02],** UML se complementa con las capacidades formales de SDL **[Dol01]**. En trabajos más recientes, dada la importancia adquirida por el lenguaje SystemC, se está aprovechando las capacidades de extensión de UML a través de estereotipos para definir una serie de

perfiles. Ejemplos de ello son un perfil para generar SystemC desde UML 2.0 **[RSRB05]** y SysML **[SML06]**. En **[RSRB06]** se presenta un entorno de diseño de SoCs basado en modelos que provee una representación gráfica de los componentes software y hardware y que permite la generación de código C, C++ y SystemC desde esos modelos.

Otros SLDLs más recientes, como AADL **[FGH06]** presentan una definición semántica. No obstante, ésta se orienta al comportamiento temporal, de manera que ésta semántica, junto con la información estructural de la especificación habilita un análisis de tiempo real. Por lo tanto, la especificación funcional ejecutable queda en un segundo plano y no queda resuelta. Por ello, aparecen propuestas en las que se complementa AADL con un SLDL como SystemC **[DLHV06]**, capaz de hacer ejecutable la especificación AADL. Existen trabajos que permiten la generación de software desde AADL. Por ejemplo, en **[ARC08]** se puede descargar un complemento para el entorno integrado *Eclipse* **[ECL08]**, capaz de generar código ADA desde AADL. En **[BDT08]** se propone la generación de código C (usando los servicios de un eOS) desde AADL usando herramientas MDA y se remarca la necesidad de flexibilidad en las herramientas generadoras de código, que a menudo son cajas negras cuyo comportamiento es difícil de configurar. En este sentido, como se verá, la propuesta realizada en este trabajo opta también por la posibilidad de tener capacidad de configurar la generación de código, tanto en lo que respecta a la configuración del eOS embebido como en otros aspectos del código generado.

El impulso que han experimentado los SLDLs basados en C/C++, tales como Handel-C, SpecC o SystemC, ha impulsado la propuesta de metodologías de generación de software desde estos SLDLs. Estos trabajos se repasan en la sección 3.2.3.

#### *3.2.2.4 El eOS en metodologías de diseño ESL*

El aumento de productividad que suponen los eOS embebidos hace que su consideración en el desarrollo de software embebido en el marco de una metodología ESL sea ineludible. Esta consideración se ha reflejado en diversos trabajos que han desarrollado modelos abstractos de eOS, capaces de modelar eficientemente servicios básicos, como concurrencia y sincronización, pero independientes de los detalles más específicos del eOS usado en la implementación. La idea es posibilitar una contabilización del impacto del eOS en el rendimiento de la implementación software tempranamente, usando la especificación de sistema durante la exploración del espacio de diseño. Por tanto, es vital hacerlo rápido y con una precisión aceptable.

Un ejemplo de estos trabajos es SoCos **[DVM00]**, un entorno de diseño de nivel de sistema que es usado para el modelado, simulación, análisis e implementación del sistema a través de un refinamiento gradual. El proceso de generación de software comienza desde una especificación ejecutable. Esa especificación consiste en una red de procesos que incluye funciones y construcciones SoCOS que permiten el modelado de concurrencia, sincronización y comunicación a través de canales. En la generación de software, el código fuente se produce de tal modo que las funciones SoCOS que modelan la concurrencia y accesos a canales de comunicación/sincronización son reemplazadas por funciones de una librería especial: el API de sistema operativo (OSAPI). Posteriormente esta OSAPI es remplazada por una plataforma eOS objetivo a

través de un simple proceso de traducción entre las llamadas a la OSAPI y las del eOS seleccionado. Una de las desventajas de esta aproximación es la de no estar basada en un SLDL, lo que atenúa el impacto de este trabajo en la comunidad de diseño embebido.

En la medida en que SLDLs como SpecC y SystemC aparecen, e incluso se estandarizan, como es el caso de SystemC **[IEEE05]**, crece el interés en realizar el modelado del eOS mediante estos lenguajes. En **[GYG03]**, se habilita el modelado de eOS en SpecC. En **[GYG03]**, a diferencia de lo que ocurre en SoCOS, el modelo de eOS se escribe por encima del SLDL (SpecC), lo que evita la necesidad de una máquina de simulación propietaria. En **[YNGJ02]** se presenta un método para generar automáticamente modelos de simulación de OS con información temporal para permitir una rápida estimación de distintas posibilidades de distintas implementaciones HW/SW en un proceso de refinado de arquitectura de comunicación en un SoC. En este trabajo se generan modelos SystemC, aunque se afirma que se puede generar código SpecC, y se reporta una aceleración en la velocidad de simulación de hasta dos órdenes de magnitud respecto a simulaciones mediante simuladores de nivel de instrucción (ISS).

En **[BRCB04] [CNFB06]**, se presenta una metodología que agiliza la exploración del espacio de diseño ya que permite fácilmente mover bloques de la especificación, descritos en SystemC, de hardware a software y viceversa y, por tanto, estudiar rápidamente distintas particiones. En un primer nivel, los bloques de especificación son módulos SystemC. Esos módulos se comunican mediante un canal funcional y, por tanto, sin información de tiempo. Tras una partición HW/SW, se puede refinar el sistema, es decir, algunos bloques se asignan a hardware en tanto que otros a software. Los módulos que van a software se identifican como procesos y se simulan mediante un modelo del eOS. La comunicación entre esos módulos y el eOS no es directa, sino que pasa por una API SystemC. De este modo, en un primer nivel de refinado (nivel 2), el sistema operativo se simula como un proceso de host (Linux en este caso) aparte del proceso SystemC que se encarga de simular los módulos hardware. En este nivel es posible determinar el comportamiento del sistema mediante un planificador expulsor con manejo de prioridades y detectar problemas como los de inversión de prioridad. Un nivel de mayor refinado (nivel 3) se basa en la plataforma SPACE **[CBRB04]**. El nivel 3 detalla el nivel dos en dos aspectos. El primero, en que la parte hardware incluye un modelo de la arquitectura SW a través de un ISS. La conexión entre las llamadas de nivel de transacción *read(address,data)* y *write(address,data)* del ISS con los módulos de hardware específico la realiza un canal controlado por una señal de reloj. Esto habilita el segundo aspecto diferenciador del nivel 3: el modelo de eOS en la partición software incluye una capa de abstracción de hardware (HAL) capaz de realizar llamadas al modelo de arquitectura hardware de SPACE. De este modo, el modelo de RTOS opera sobre la arquitectura objetivo.

Como se explicó en la sección 1.4, el GIM/UC también ha contribuido a este campo de investigación. En **[PASV06]** se habilita la simulación en SystemC de un código C/C++ con llamadas POSIX. En **[PAVE06]** se extiende **[PASV06]** para la consideración del sistema completo, incluyendo el eOS y otros elementos, como modelos TLM del bus, varios procesadores, etc.

Los trabajos presentados en esta sección se orientan fundamentalmente al modelado, consideración, simulación y análisis del eOS embebido, en algunos casos,

empleando SLDLs como SpecC y SystemC. Con esto, se generan modelos de nivel de sistema que sirven para acelerar las estimaciones de rendimiento y, por tanto, la exploración del espacio de diseño (DSE). Sin embargo, estos trabajos no resuelven la generación automática y directa desde un SLDL del software embebido que incluya de forma automática, además del código de aplicación, las llamadas a un eOS. En casos como SoCOS, la generación de software se afronta mediante un proceso de refinado que requiere aún un proceso de traslación. Este proceso, es manual, por tanto lento y propenso a errores. Además, el usuario debe mantener la coherencia entre dos representaciones (la especificación SoCOS y la traslación con llamadas de la OSAPI). En cambio, la metodología presentada en este trabajo, considera la generación automática de software desde un SLDL incluyendo las llamadas al RTOS embebido.

### 3.2.3 Generación de eSW desde SLDLs basados en C/C++

Los SLDLs basados en C/C++, tienen la ventaja de que se construyen sobre un lenguaje anfitrión familiar para los diseñadores embebidos. Como se detalló en la sección 2.3.1, C y C++ son los lenguajes más empleados en sistemas embebidos. Además, en casos como el de SystemC, el lenguaje tiene una semántica bien definida y estable. La documentación estándar de SystemC define la semántica de cada primitiva de especificación con detalle, de forma que es posible predecir el comportamiento de la especificación con bastante precisión. Ello permite constreñir y fijar el resultado de la generación de software y, aún más, da la posibilidad de usar la especificación de sistema como un modelo de oro con el cual contrastar los resultados de la generación. Por ello, dado el impacto en la comunidad de diseño de sistemas embebidos de estos SLDLs, han aparecido algunos trabajos en el campo de la generación de software embebido que parten de SLDLs basados en C/C++. Muchos de estos trabajos han sido desarrollados a la vez que los que comprende esta tesis.

En **[GLMS02] [YDG03]** se reseñó la posibilidad de generación de software desde un SLDL basado en C/C++ por compilación directa de la especificación, incluyendo los elementos propios del SLDL y el kernel de simulación. Por ejemplo, se podría tomar una especificación SystemC y realizar una compilación cruzada incluyendo la propia librería SystemC. Sin embargo, esta técnica se descarta por su ineficiencia. La inclusión del kernel de simulación en el software resultante supone un gasto en tamaño de código (y por tanto de memoria) que en su mayor parte es innecesario para la aplicación embebida. Por otro lado, no es trivial realizar una compilación cruzada de todas las estructuras C++ implicadas en la librería SystemC. Además, en cualquier caso, seguiría siendo preciso proveer elementos de código específicos de la plataforma de implementación, por ejemplo, lo referente a manejo de dispositivos de entrada/salida.

Frente a esta posibilidad, otras técnicas de generación de software desde SLDLs basados en C/C++ explotan el hecho de que el SLDL se comporte como un *lenguaje envoltorio* o extensión de C/C++, considerado como *lenguaje anfitrión*. Estas técnicas, se basan en eliminar en mayor o menor medida parte de los elementos del SLDL. Algunos de esos elementos aparecen explícitamente en la especificación, en tanto que otros son implícitos, por ejemplo, el kernel de simulación en el caso de SystemC. Inicialmente, en el nivel de especificación de sistema, esa información es útil, por ejemplo, para realizar la simulación o para realizar chequeos de algún tipo. Sin

embargo, esta información es innecesaria en la implementación software para la plataforma objetivo, por no aportar nada a su funcionalidad o a su rendimiento.

De este modo, las diferentes técnicas de generación de software desde un SLDLs que generan código en lenguaje anfitrión (por ejemplo, C/C++) desde un lenguaje envoltorio SLDL (por ejemplo, SystemC) se caracterizan por:

- El grado de preservación de las facilidades adicionales provistas por el SLDL al lenguaje anfitrión.
- En si y cómo se reutiliza la implementación de las facilidades o características del SLDL preservadas.

Por ejemplo, en el caso SystemC→C/C++, el primer punto se refiere a si se mantiene la estructura de concurrencia, la estructura de módulos, la semántica temporal, etc. Por ejemplo, unas técnicas pueden optar por generar implementaciones puramente secuenciales en tanto que otras pueden preservar la concurrencia. El segundo punto, se referiría, una vez que se ha decido mantener la concurrencia, a si la técnica de generación reutiliza por completo la solución provista por el kernel de simulación o si, por el contrario, provee una implementación *ad hoc* del planificador. También serán posibles soluciones intermedias, en las que la implementación del SLDL se sustituya por código existente. Por ejemplo, para implementar la concurrencia, se pueden utilizar los servicios de un eOS. De este modo, cada técnica puede buscar un compromiso específico entre eficiencia, portabilidad, robustez y rapidez en la generación. En las subsecciones siguientes se repasan trabajos concretos. Cada uno ha realizado una decisión de compromiso respecto a los puntos anteriores.

#### *3.2.3.1 Técnicas basadas en varios pasos de refinado y modelos intermedios*

En **[YDG03] [YDG04]** se propone una metodología de generación de software ANSI-C eficiente desde una especificación descrita en un SLDL, aunque se desarrolla en el ámbito del lenguaje SpecC. Se genera código ANSI-C con ánimo de validar el método para un gran número de plataformas objetivo, ya que la gran mayoría de compiladores cruzados soportan C. Consiste en una serie de pasos de refinado y modelos intermedios en el flujo de generación, hasta que se alcanza el código ANSI-C.

En la Figura 3-4 se detallan los tres pasos fundamentales explicados en **[YDG03][YDG04]**:

1. **Creación de Tareas:** En este primer refinamiento los módulos y procesos de la especificación bajo el SLDL se convierten en tareas con una asignación de prioridades. Esa conversión se hace usando primitivas correspondientes al modelo de eOS (RTOS en la figura) presentado en **[GYG03]**.
2. **Generación del código C:** Se genera el código ANSI-C correspondiente a las tareas generadas en el paso anterior. No obstante, se mantiene aún el modelo de eOS. Esto produce un modelo de simulación que permite validar el código C generado.
3. **Particularización para el eOS más Compilación y Enlace:** En este tercer y último paso el modelo de eOS es mapeado a un eOS específico y el código C

resultante es compilado para la arquitectura objetivo. El binario final se genera enlazando contra las librerías del eOS.

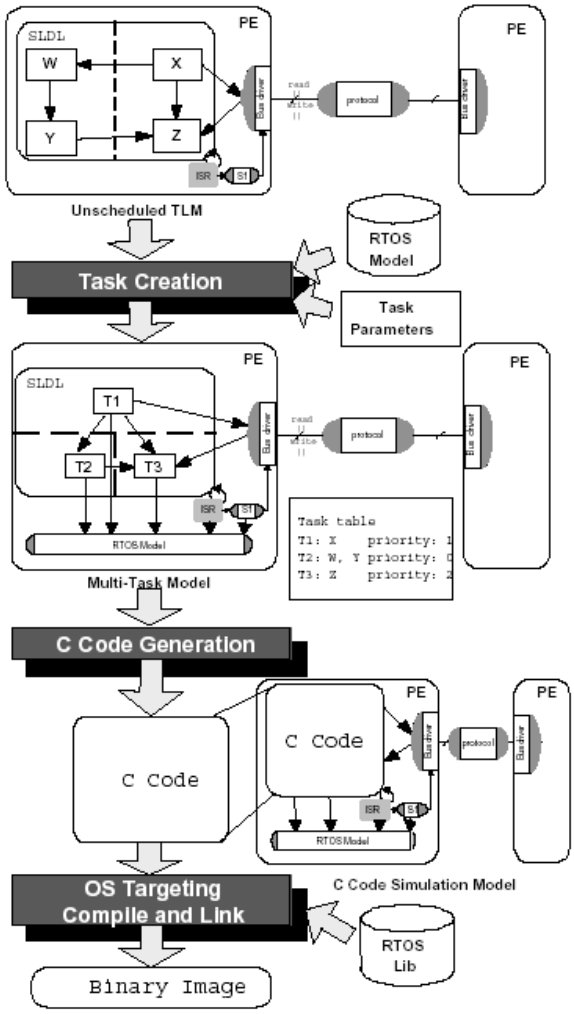



**Figura 3-4. Flujo de Generación de software por refinamiento (tomado de [YDG04]).**

En **[YDG05]**, esta metodología se desarrolla y permite, desde un modelo TLM, la generación automática del código, que incluye los controladores necesarios para la comunicación HW/SW, además del código de aplicación y las primitivas del RTOS. Esto permite que esta técnica pueda generar software pata plataformas que incluyen HW específico o nuevos periféricos con un controlador HW asociado conectados al bus del sistema.

La metodología asume que, en alto nivel, la comunicación entre los elementos procesadores que aparece tras la partición del sistema, se modela a través de paso de mensajes abstractos. A partir de ese modelo abstracto, se dan los pasos de refinado de esos pasos de mensajes a pines y conexiones físicas. Para ello, se da un refinamiento de canales. El refinamiento de canales supone la definición de la topología de red de comunicación y la generación de los enlaces punto-a-punto. Cada enlace punto a punto permite la transferencia de mensajes. Los enlaces punto-a-punto se agrupan a su vez en enlaces físicos. Un aspecto característico de esta metodología es que considera una

jerarquía de cuatro capas de canales de comunicación: Presentación, Enlace, Acceso al medio y Protocolo (de mayor a menor nivel de abstracción). De esta manera, la capa más abstracta (Presentación) acaba usando los servicios de la menos abstracta (Protocolo), que trabaja con transferencias de datos en modo palabra o trama, mapeados directamente en memoria, y con temporización. Una vez definida la topología de red, agrupados los enlaces punto-a-punto en enlaces físicos, y teniendo en cuenta esta distribución de capas, se genera por cada una el código C, que es básicamente un control de acceso al bus de comunicación para cada PE.

#### *3.2.3.2 Generación de eSW por Síntesis de RTOS*

En el caso de la técnica de la sección 3.2.3.1, la concurrencia se implementa en software mediante los servicios de un RTOS. Eso requiere el mapeo a llamadas de un RTOS específico existente. En cambio, en **[GYJ01],** dentro de un marco de diseño de nivel de sistema basado en componentes **[DCBG02],** se propone una técnica que genera automáticamente un RTOS específico para la aplicación. El RTOS específico generado comprende únicamente los servicios estrictamente necesarios e invocados por la aplicación. De este modo, se optimiza la huella o tamaño de código correspondiente al RTOS embebido.

La entrada al mecanismo de generación de software es una especificación del sistema que consiste en una descripción estructural jerárquica de la comunicación entre módulos, el comportamiento de los módulos y la información de la asignación de memoria. La técnica no fija el *front-end* específico, que, en general usa un SLDL. En **[GYJ01]**, el ejemplo presentado usa una extensión de SystemC.

Esta técnica utiliza una serie de componentes encargados de misiones específicas dentro del proceso de generación de software. Un analizador de la arquitectura toma la información de estructura y la información de asignación de memoria y determina los parámetros de los módulos y los servicios de RTOS requeridos, que pasa al selector de código. El selector de código, mediante un algoritmo recursivo, se encarga de determinar los elementos de la librería del RTOS necesarios para cubrir los servicios requeridos. De esta forma se pasa una serie de macros al expansor de código, que genera el RTOS específico con los servicios requeridos. Un adaptador del código de tareas es el componente metodológico encargado de la generación del código de las tareas mapeado al RTOS generado. El adaptador del código de tareas necesita información del selector de código para incluir las correspondientes llamadas al sistema en el código de tareas, así como los protocolos de comunicación y parámetros extraídos por el analizador de arquitectura y de la propia descripción de las tareas en el nivel de sistema (mediante el SLDL). La metodología genera los *Makefiles* necesarios a través de un generador de Makefiles que recaba información provista por el selector de código y por el adaptador de código de tareas.

Un inconveniente de este trabajo es que está basado en una librería de RTOS específica de esta metodología. Dado que la técnica permite generar un número grande de combinaciones en función de los servicios requeridos, es difícil asegurar la robustez de cada RTOS sintetizado. El empleo de RTOS extendidos, bien comerciales, bien de libre distribución, sería beneficioso en ese sentido, ya que llevan detrás un proceso de depuración importante en el que han colaborado muchos desarrolladores, mantenedores

y usuarios. Además, la especificidad de esa librería RTOS puede provocar un mantenimiento complejo y vertical, que fuerce a adaptar y mantener otras herramientas asociadas al RTOS, como el depurador, el compilador, etc.

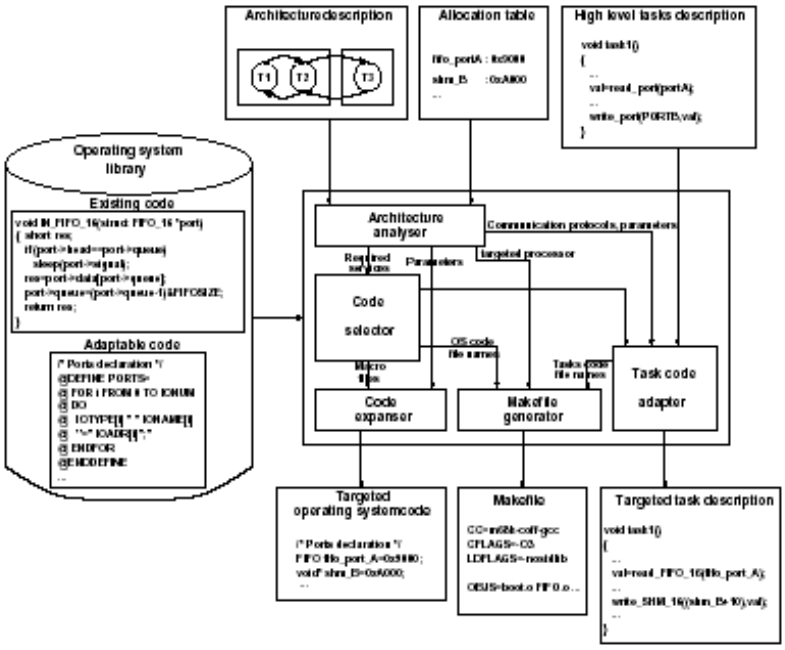



**Figura 3-5. Flujo de generación automática de RTOS (tomado de [GYJ01]).**

## 3.2.4 Generación de eSW de Fuente Única desde SystemC

### *3.2.4.1 Introducción*

El auge de SystemC como lenguaje de especificación de sistema ampliamente aceptado ha fomentado la aparición de propuestas para la generación de software desde este SLDL. En el trabajo presentado en la sección 3.2.3.2, se empleó una extensión de SystemC para realizar un modelo de nivel de componente del sistema.

Algunos trabajos presentados anteriormente, como **[YDG03]**, **[YDG04] [YDG05],** trabajan con representaciones intermedias, que reflejan las transformaciones producidas entre el código de especificación y el producto final de la generación. Mantener la coherencia entre la especificación más abstracta y las representaciones intermedias puede ser ineficiente y dar lugar a errores de implementación y verificación. El que el usuario no tenga que tratar con estas representaciones es deseable y contribuye a la eficiencia y abstracción del método. Pero, aún considerando la automatización de los refinamientos, a veces es necesario editar y/o controlar el software generado. Si las transformaciones son profundas, es costoso identificar la estructura del código de especificación en el código de implementación y facilitar así su edición.

En contraste con esta aproximación, aparecen las técnicas de generación de *fuente única*, que provienen de la observación de que el código de especificación de sistema y el código software mantienen, en general, una alta correlación **[CFHS03]**. Así, una metodología de generación de software de fuente única produce el software de

implementación tomando la especificación de nivel de sistema como entrada y sin generación de representaciones intermedias. La idea es que el usuario maneja el código fuente de la especificación del sistema, específicamente, el de la partición software, prácticamente como si fuera el código fuente del software generado. El usuario asume una alta correlación entre el código generado y el que el usuario maneja. Esto le da al usuario un grado alto de control y confianza en el código generado, a la vez que el proceso de generación de software se automatiza y se elimina la necesidad de mantener representaciones intermedias. La alta correlación entre el código de entrada y de salida no impide que pueda existir cierto grado de refinamiento y de configurabilidad, ni que el método de generación de *fuente única*, al igual que los anteriores, pueda realizar una generación eficiente, necesaria en el ámbito de sistemas embebidos.

La correlación entre el código SLDL y el código del software de implementación es aparente cuando el SLDL se constituye como una extensión o envoltorio de un lenguaje anfitrión y dicho lenguaje anfitrión se emplea para codificar el software de implementación. Tal es el caso cuando se trata de una generación desde SystemC a C/C++. Se da una similitud sintáctica y estructural entre la especificación SLDL (SystemC) y el código software C/C++ resultante de un refinamiento manual. Los bloques de código para el cómputo secuencial son prácticamente idénticos. Las invocaciones y construcciones al declarar procesos concurrentes, y al sincronizarlos y comunicarlos entre sí son muy parecidas. Por ejemplo, en SystemC, la utilización de un proceso requiere su declaración, para lo que se pasa la función que contiene el código del proceso como parámetro a una macro específica (por ejemplo, SC_THREAD). Esto tiene un parecido notable con una implementación software, que usualmente requiere declarar y pasar como parámetro la función del proceso a una función de declaración de hilo, cuya sintaxis específica depende de la o las API(s) sorportada(s) por el RTOS.

Partiendo de esa correlación, en **[GLM02]** se propuso reemplazar las construcciones SystemC por construcciones equivalentes en C. A continuación se muestra un ejemplo:

```
#ifdef SYSTEMC
#include "systemc.h"
#else
#define SC_MODULE(name) struct name
#endif

SC_MODULE(my_module) {
#ifdef SYSTEMC
public:…
 sc_fifo_out<int>  out; // SystemC port
#else
 struct fifo *out; /* port equivalent in C */
#endif
};
```

En el ejemplo, si la variable SYSTEMC está definida, se compila una descripción SystemC. En caso contrario, se realiza una compilación cruzada de código ANSI-C plano, también presente en el código. La declaración de la variable SYSTEMC determina qué código pasa el preprocesador al compilador o al compilador cruzado en su caso. En caso de que la variable de preprocesado SYSTEMC esté definida, se utiliza el código con sintaxis SystemC. Así, el módulo *my_module*, declarado mediante la macro *SC_MODULE*, se declara como una estructura C++ que hereda la clase *sc_module*, que contiene todo el artificio preciso para elaborar y simular el módulo en SystemC. En caso contrario, el módulo *my_module,* mediante una redefinición de la macro SC_MODULE, se declara como una simple estructura C. Asimismo, se emplea código C para la declaración e implementación de entidades SystemC como puertos. Por ejemplo, la instancia de plantilla de clase de tipo puerto, se sustituye por un puntero de tipo *fifo* que ahora tiene una declaración *ad-hoc* ANSI-C.

Esta técnica no está exenta de ciertas dificultades al realizar la traslación. Por ejemplo, existen dificultades propias de una traslación C++ (de SystemC) a C. En algunos casos el mapeo es factible y permite preservar la estructura y ámbitos de visibilidad. Por ejemplo, si un módulo SystemC (una clase C++) se implementa como una estructura C, todos los miembros de datos públicos del módulo tendrán un ámbito de visibilidad similar en la especificación y en la implementación. Para los casos en los que no exista un mapeo inmediato resoluble con construcciones de C, en **[GLM02]** se propone la utilización de las llamadas y/o servicios del RTOS presentes en la plataforma objetivo. Por ejemplo, se propone la sustitución de los procesos SystemC SC_THREAD por hilos POSIX. En ese caso, se consigue una gran portabilidad, debido a que múltiples RTOS proveen esa API. En resumen, en este trabajo, se esboza la idea de generar software explotando la correlación entre SystemC y C, pero no se desarrolla ninguna metodología específica. Además, existen varios aspectos que impiden hablar de una metodología de fuente única:

- Las fuentes reflejan aún dos versiones del código, la de especificación y la de refinado software.
- Estas versiones se entrelazan en el código y lo hacen menos legible y comprensible.
- El método obliga aún al usuario a una traslación manual, que obliga a escribir tanto la versión de especificación, como la de implementación software.

La propuesta presentada en esta tesis ha contribuido a superar esos inconvenientes. En **[CHPS03] [CFHS03]** se refina y pone en práctica la generación de fuente única en SystemC, separando las fuentes de la especificación del sistema del software generado, por tanto, habilitando el manejo por el usuario de una única versión: la de sistema; y automatizando el proceso de generación como un intercambio de librerías (la de SystemC por una librería de generación de software). Asumiendo una generación C++, la metodología propuesta preserva tanto la estructura de módulos (y por tanto de ámbitos de visibilidad), como la de concurrencia, con lo que se facilita la identificación y correlación del código generado respecto al código de especificación. Se introducirá y explicará esta técnica en la sección 3.3. Antes, la siguiente sección cerrará el estudio de estado del arte realizado en este trabajo, repasando un trabajo

relacionado, que toma SystemC como *front-end* y que se puede encuadrar dentro de las metodologías de fuente única.

#### *3.2.4.2 Generación de eSW desde SystemC del Virginia Tech (VT)*

En **[Sir02] [SAB02]** se presenta un método de generación de SW de fuente única (o modelo único, como se denomina en **[SAB02]**) desde una especificación descrita en SystemC. En **[SAB02]**, el preprocesado genera código C++ y la metodología se enmarca en una metodología de co-diseño hardware-software, en la que existe un perfilado de la especificación de nivel de sistema, una decisión de la partición, síntesis de hardware, generación de interfaces y generación de software. En **[Sir02]** la misma metodología explora la generación de código C, ya que el mapeado se hace a una arquitectura que no cuenta con compilador de C++. No obstante, la metodología, aunque similar en varios aspectos a la metodología que se presenta en este trabajo, presenta una serie de diferencias fundamentales tanto en el nivel de especificación como en la técnica de generación.

La metodología de especificación propuesta en **[SAB02] [Sir02]** soporta jerarquía a través de módulos y puertos. Así mismo, admite especificación concurrente, ya que la computación se puede dividir en procesos síncronos de reloj (*SC_CTHREAD*). La señal SystemC (*sc_signal*) es la primitiva básica de sincronización entre procesos. La comunicación entre procesos que implique transferencia de datos, se resuelve por medio de una variable compartida cuyo acceso, no obstante, debe ser sincronizado mediante un protocolo de dos señales (DATA_READY y ACK). De esta forma, la metodología de especificación se caracteriza por un MoC cercano al de diseño de hardware síncrono.

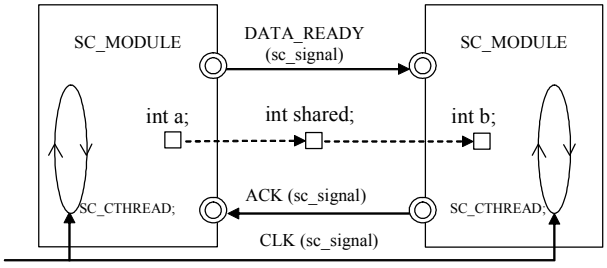



**Figura 3-6. Primitivas básicas de especificación SystemC en [SAB02] [Sir02].**

Una ventaja de que la metodología de especificación adopte este MoC, es que, al ser muy cercano al subconjunto sintetizable hardware de SystemC **[OSCI05]**, los pasos de refinamiento a hardware y, por tanto, el flujo de síntesis de hardware, se simplifican. La generación automática de software, se realiza traduciendo el código SystemC a estructuras C++ **[SAB02]** o C **[Sir02]** incluyendo un planificador propio, es decir, una librería de tiempo de ejecución o RTL (*Run Time Library*) específica de esta metodología. Esta RTL es una versión simplificada y, por tanto, ligera, del kernel de simulación de SystemC. Ese planificador específico, representado en la Figura 3-7, ranura o divide el tiempo en software para definir los instantes de actualización y periodos de validez del valor de la señal. La simplicidad de este esquema (Figura 3-7) se puede contrastar con el esquema de simulación de SystemC. En otras palabras, el modelo de ejecución síncrono de reloj CS en **[Jan04]**) es transportado o implementado en software. Este modelo se sigue en especificación y se preserva en implementación.

De este modo, esta metodología cumple el paradigma de fuente única tanto para hardware como para software, en tanto que las construcciones son semánticamente interpretables desde cualquiera de los módulos del sistema, independientemente de la partición elegida.

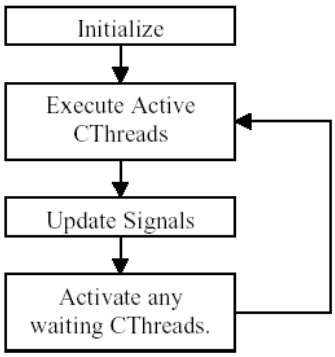



**Figura 3-7. Esquema básico del planificador en la metodología de generación de SW del VT.**

Sin embargo, esta metodología presenta algunas limitaciones. Aunque esta aproximación simplifica la síntesis HW, el mapeo a software se complica, por el hecho de exigir la traducción de la señal y el proceso síncrono de reloj (*SC_CTHREAD*) a software. Esta traducción no es trivial, ya que tiene implicaciones en la semántica de ejecución de los procesos. Por otro lado, en **[Sir02],** la implementación software, sustituye el planificador SystemC por un planificador que implementa el "ciclo software" de tal forma que la relación con la temporización del reloj del sistema en hardware no es inmediata, ya que la duración de cada ciclo software puede variar. Este ciclo software repercute en la eficiencia del código[5]. En **[SAB02]** se reporta que una implementación desde una versión SystemC de un ejemplo basado en un Vocoder GSM presenta un tiempo de ejecución un 10% superior al tiempo tomado por una versión codificada sin planificador, hilos y señales. En **[Sir02]**, llega al 20%. Los datos, no obstante son relativamente aceptables, si se tiene en cuenta la ganancia que supone la automatización del proceso, la posibilidad de movilidad de bloques de especificación entre hardware y software y que el código sufre pocas modificaciones, por tanto, conserva fundamentalmente la estructura original.

Una limitación más importante es la rigidez de la especificación de entrada. El usuario sólo puede emplear canales *sc_signal* y procesos *SC_CTHREAD* para especificar comunicación y cómputo. Se da una resolución a la generación de software para una especificación mono-MoC. Además, la adecuación de este MoC a una metodología de nivel de sistema es limitada. Es un estilo de modelado más cercano a la descripción hardware que, frente a la tendencia actual en las metodologías de nivel de sistema, resta abstracción. Implica el uso de un mayor nivel de detalle en los protocolos

[5] Una alternativa a esta solución hubiera sido finar una relación exacta entre el reloj software y el reloj hardware del sistema. Sin embargo, este caso presenta también sus dificultades, ya que el cómputo software se ve sujeto a las restricciones temporales que impone el modelo síncrono de reloj. Al igual que ocurren en hardware síncrono, sería preciso encontrar el camino crítico del cómputo software y asegurar que no excede el periodo de reloj software.

de comunicación (nivel de señal) y en el manejo del modelo de tiempo del sistema. Por lo tanto, se requiere un esfuerzo adicional al especificador que no produce rentabilidad en términos de software.

Otro inconveniente, tal y como ocurría en la técnica de síntesis de RTOS de la sección 3.2.3.2, tiene que ver con la robustez, ya que el kernel de ejecución propio (es decir, una librería de ejecución o *run-time* library específica de la metodología de generación) no cuenta con el mismo mantenimiento y soporte a nuevas plataformas que un RTOS comercial o abierto ampliamente extendido.

Por tanto, es necesaria una metodología de generación de software desde SystemC capaz de generar software desde niveles más abstractos, que sea capaz de abarcar más estilos de especificación o, al menos, complementar aproximaciones como las de **[SAB02] [Sir02]**. A la vez, es preciso que la metodología se adapte e integre el estado del arte en generación de software embebido, lo que incluye fundamentalmente la reutilización de código mediante el empleo de un eOS y *drivers* de dispositivos y la integración de/en herramientas y plataformas típicas de desarrollo cruzado.

## 3.3 Introducción a SWGen

La segunda línea de investigación de esta tesis consiste en el desarrollo de una metodología de generación automática de software embebido desde SystemC denominada *SWGen*. *SWGen* se encuadra dentro de una metodología de diseño de nivel de sistema, que también incluye a la metodología *HetSC*.

A continuación se exponen las características distintivas de la metodología de generación de software embebido *SWGen*.

**Desde un lenguaje estándar**. Se toma como entrada una especificación en SystemC. La normalización de SystemC (IEEE 1666) da estabilidad a la semántica del lenguaje de entrada, y en consecuencia, a la técnica de generación.

**Aproximación de Fuente Única.** Con la misma especificación se puede realizar una compilación nativa contra las librerías de especificación (*SystemC* y *HetSC*), para la simulación y verificación de nivel de sistema, y una compilación cruzada contra la librería *SWGen*, para generar el binario a cargar en la plataforma. Por lo tanto, se mantiene una única fuente para la especificación y para la implementación software.

**Basada en una librería metodológica**. La generación se basa en la sustitución de la librería *SystemC* (y *HetSC*) por la librería *SWGen*. La librería *SWGen* provee una implementación SW eficiente de las facilidades SystemC usadas en la especificación.

**Basada en plataforma HW/SW.** La metodología genera software para una plataforma hardware/software (HW/SW) que toma como componente un eOS con unos servicios básicos. La metodología es flexible, ya que es posible seleccionar entre varios eOS. Esto permite usar eOS comerciales o abiertos, en lugar de eOS ad-hoc o específicos de la metodología de generación, lo que dota de robustez, un soporte más continuado y mayor disponibilidad de herramientas relacionadas a la metodología.

**Generación Automática.** Se elimina la necesidad de un proceso de traslación manual desde SystemC al lenguaje de programación. Apenas un comando de consola y ediciones de *Makefile* son necesarias para generar el código software, fuente o binario. De este modo, el proceso de generación es más rápido y se elimina la introducción de errores propios de la traslación manual.

**Conservación de Estructura Modular y de Concurrencia.** En el software generado, la estructura de ámbitos de visibilidad y de disposición de hilos y sus respectivas comunicaciones se mantiene respecto a la especificación de sistema. Esto hace más fácil entender y localizar el código generado, haciendo su edición factible.

**Generación Eficiente.** Se eliminan todos los elementos del kernel de SystemC que, bien no son necesarios para la ejecución del software (depurado, etc), o bien ya están provistos por la plataforma HW/SW (soporte de concurrencia, temporización, etc). De esta forma el costo en tamaño de código y velocidad se mantienen reducidos.

**General.** Sin detrimento de que el empleo de un modelo de computación específico permita la aplicación de una técnica de generación o síntesis de software más eficiente, el método de generación es lo suficientemente general para permitir la síntesis sistemática de software desde cualquiera de los MoC abstractos cubiertos por *HetSC*.

**Escalable.** La librería de generación está estructurada en distintos paquetes de traslación, dependiendo del nivel de portabilidad del código, lo que permite una fácil extensión para nuevas plataformas objetivo.

**Diferentes posibilidades de validación.** La metodología habilita varias posibilidades de validación del código generado sin necesidad de tener que descargar el software a la plataforma objetivo.

En **[FHSV02]** se dieron los primeros avances de la metodología propuesta. Se puso en práctica la sustitución de código, de forma similar a **[GLMS02],** aunque con algunas diferencias. En **[GLMS02]** se ejemplifica la generación de C y el código que maneja el usuario consiste en dos versiones explícitas, una SystemC y otra C, separadas por sentencias de preprocesado. En cambio, en **[FHSV02]** se generaba código C++ y la generación se daba por sustitución de librerías, lo que permitía al usuario de forma práctica manejar el mismo código fuente de especificación, válido tanto para simulación como para generación de software. Asimismo, ya se utiliza como parte de la plataforma HW/SW un RTOS embebido (*eCos*), a diferencia de la aproximación de **[Sir02]**. Además, el uso de ese RTOS y de su API era trasparente al usuario y se asumía como parte de la plataforma de implementación, a diferencia de las metodologías de modelado de RTOS. Asimismo, tampoco se incluían pasos de refinado hasta obtener una versión intermedia que incluyera las llamadas al RTOS, como en otras metodologías de generación presentadas. En **[FHSV02],** la metodología de generación de software intercambia directamente los servicios de SystemC por los del eOS. También en **[FHSV02],** y en **[HSV02]**, se incidió en la parte de generación SW tocante a las interfaces HW/SW. La metodología madura en **[HPSV03] [CHPS03] [CFHS03]**, donde se establece la posibilidad de mapeo a distintas APIs de RTOS; se empieza a estructurar la librería *SWGen* en función de los distintos niveles de portabilidad y servicios de SystemC que es preciso soportar en la implementación (jerarquía modular, concurrencia, tipos de datos, etc); y se establecen niveles intermedios entre el nivel de especificación y de generación para la validación del código generado. En **[RPHF04]** aparece una visión de cómo se integra la metodología de generación de eSW en una metodología de diseño ESL de sistemas embebidos.

El estado actual de esta metodología se detalla en las secciones siguientes. Esa evolución se puede seguir también en **[SWG08]**.

## 3.4 Flujo de Generación

Esta sección da una visión del flujo de generación automático de eSW a partir de la especificación *HetSC*. También muestra la simpleza de los pasos que tiene que dar el usuario para seguir ese flujo. Para ello, se asume que el usuario ha instalado las herramientas típicas de desarrollo cruzado necesarias (herramientas de desarrollo cruzado, RTOS embebido, etc). Otra asunción básica actual de la metodología es que la partición software del sistema se ejecutará en un único procesador.

La Figura 3-8 muestra la metodología *SWGen* incrustada en una metodología de diseño ESL basada en SystemC. En la parte superior derecha de la Figura 3-8 se representan las actividades de especificación y verificación de nivel de sistema. La especificación se hace bajo la metodología *HetSC*. Esta especificación sirve para

simular y realizar, al menos, una verificación funcional del sistema. La aplicación de otras metodologías desarrolladas por el GIM sirven para la estimación de rendimiento directamente a partir de especificación HetSC (usando la librería *Perfidy* **[HPSV04])** y tras la generación de software (usando la librería *PERFidiX* **[PASV06]**).

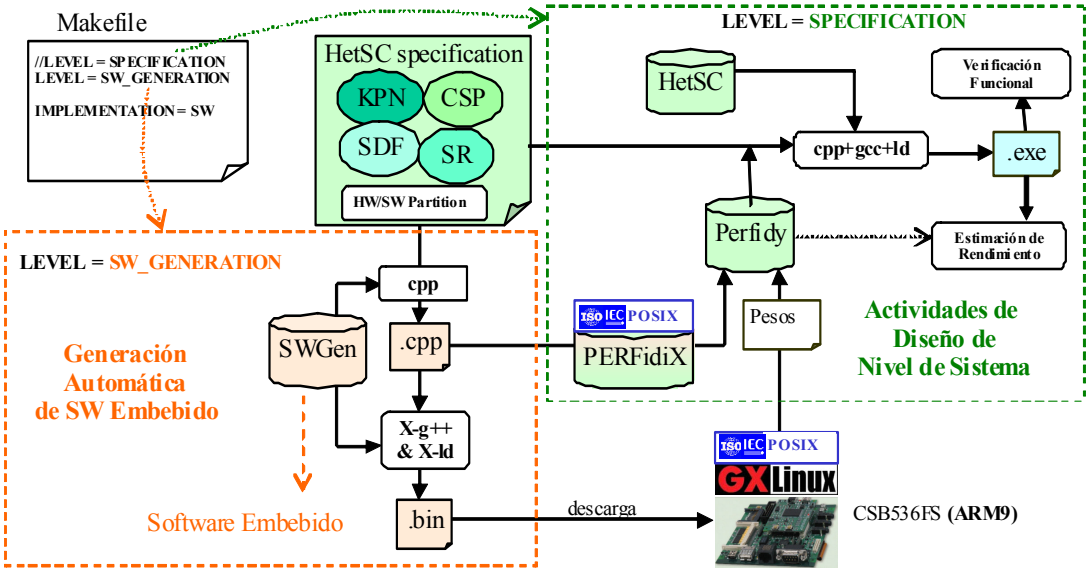



**Figura 3-8. Metodología SWGen integrada en una metodología de Diseño de Nivel de Sistema.**

En el nivel de generación, representado en la parte inferior izquierda de la Figura 3-8, el flujo de generación automática de software toma como entrada la misma especificación *HetSC*, que no precisa ser modificada.

Para indicar si la especificación se usa en el nivel de sistema o de generación, existen diversas alternativas. En la Figura 3-8 se ilustra el uso de un único *Makefile*. En ese caso se usa la variable LEVEL dentro del fichero *Makefile*. El valor de esta variable afecta al preprocesado de la especificación. La variable LEVEL puede tomar tres valores, SPECIFICATION, COSIMULATION y GENERATION.

En el caso en que LEVEL=SPECIFICATION, los Makefiles fuerzan una compilación nativa de la especificación contra la librería núcleo de SystemC (y la librería *HetSC* si se usa). En este nivel, se pueden incluir otras librerías de nivel de sistema, como las de estimación de rendimiento desarrolladas por el GIM, mostradas en la Figura 3-8, u otras, como por ejemplo, la SCV.

En el caso en que se defina LEVEL=SW_GENERATION, el código fuente de la especificación *HetSC* se usa para generación de software. En la Figura 3-9 (variación de la Figura 3-2) se representa en más detalle el flujo de generación SW esquematizado en la Figura 3-8. El flujo de generación de software precisa la información de la partición HW/SW. Actualmente esa información se incrusta en la especificación de sistema, aunque sin tener implicaciones en la funcionalidad. En la sección 3.5.2 se precisa con detalle cómo se provee esa información en forma de variables de preprocesado, que permiten realizar una selección del código de especificación perteneciente a la partición software. De esa forma, sólo el código fuente en el ámbito en el que la variable de preprocesado SW_SECTION está definida se pasa al compilador cruzado.

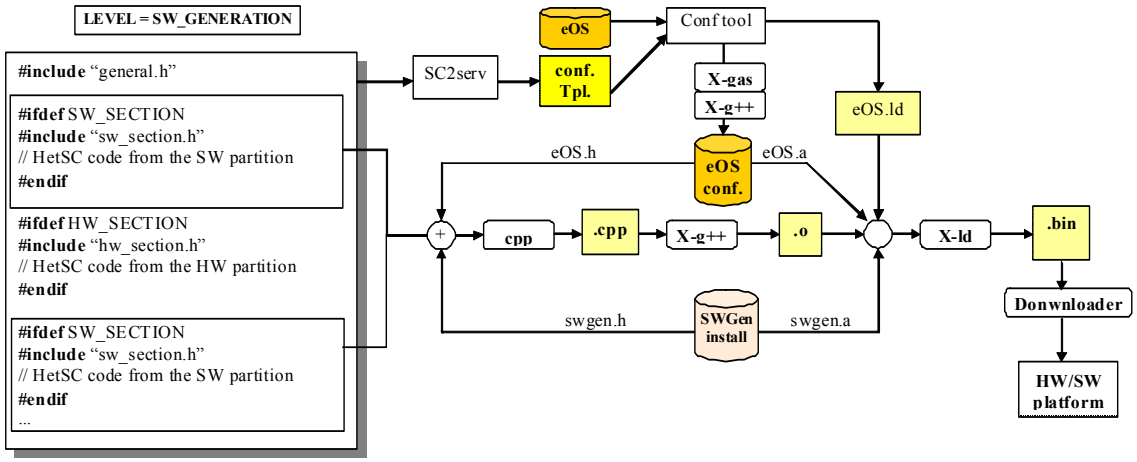



**Figura 3-9. Flujo de generación de software *SWGen* detallado.**

Además, también se da una sustitución de librería metodológica. En lugar de incluir la librería SystemC (y otras librerías de nivel de sistema, como *HetSC*, Perfidy, etc), se incluye la librería *SWGen*. La inclusión de esta librería sustituye las llamadas a los servicios del kernel de SystemC por las llamadas a los servicios de un eOS. Por lo tanto, la metodología asume que existe un eOS o RTOS como parte del SW de la plataforma HW/SW, independientemente de que éste sea de tiempo real o no, comercial o de libre distribución. El RTOS se trata como un componente más de la plataforma de implementación que puede ser cambiado dependiendo de la aplicación y las restricciones de rendimiento. Esto diferencia a esta metodología de aproximaciones como las de las secciones 3.2.3.2 y 3.2.4.2, y permite aprovechar el estado del arte en desarrollo de software embebido

Otra variación introducida cuando LEVEL=SW_GENERATION es la sustitución de la aplicación de un flujo de compilación nativo por uno de compilación cruzada. Por ejemplo, en la Figura 3-8 (parte inferior derecha) se presenta una plataforma HW/SW basada en un procesador ARM9, que cuenta con una distribución de Linux Embebido. En las figuras y nomenclatura usadas en este trabajo, para representar de forma genérica la aplicación de las herramientas de desarrollo cruzado se usa como prefijo la "X". De este modo, X-gcc, X-g++, X-ld, representan la compilación C/C++ y enlazado para un arquitectura objetivo X. Así, si la arquitectura de la plataforma es ARM, entonces X=*arm-elf* y se hablaría de las herramientas *arm-elf-gcc*, *arm-elf-g++* y *arm-elf-ld*.

En la metodología *SWGen*, las herramientas de desarrollo cruzado se suponen dadas. Por lo tanto, un requisito necesario es el de contar con un compilador cruzado de C++. Las arquitecturas más típicas en sistemas embebidos (ARM, MIPS, OpenRISC, etc) cuentan con soporte de compilación C++. Asimismo, cada vez más RTOS embebidos soportan C++. En **[RPAG00]** se reporta el soporte de C++ para los RTOS embebidos *Linux*, *RTLinux*, *RTEMS*, *QNX*, *VxWorks* y *LynxOS*, además de *eCos*. Por otro lado, diversos trabajos **[CPPP06] [KoMo99] [STL05]** demuestran que el uso de C++ frente a C no produce un impacto apreciable en el rendimiento.

La Figura 3-9 se puede cotejar con la Figura 3-2 para identificar las variaciones de la metodología de generación propuesta con respecto al flujo de diseño tradicional. En primer lugar, se parte de una especificación en SystemC, en lugar de partir de una

especificación escrita en C o C++. En segundo lugar, se requiere la librería *SWGen*. Esta se aplica tanto en la fase de compilación, como librería fuente (*swgen.h*), como en la fase de enlace, como librería objeto (*swgen*.a). Por otro lado, la herramienta que genera la plantilla de configuración del RTOS a partir de la aplicación, en lugar de extraer los servicios requeridos del RTOS desde el código de aplicación (*app2serv* en la Figura 3-2), lo debe hacer desde el código SystemC (*SC2serv* en la Figura 3-9). Esta herramienta no es indispensable. Prueba de ello es que no ha sido objeto de desarrollo en el ámbito de este trabajo. No obstante, es deseable, ya que agilizaría la tarea de configuración del sistema operativo y eliminaría posibles errores en tiempo de enlazado por omitir la inclusión en la configuración de un servicio requerido por la especificación o ineficiencias por incluir servicios no necesarios.

A continuación, se muestra la simpleza de los pasos que el usuario debe dar para, a partir de la especificación *HetSC*, generar automáticamente el software embebido. Una vez que se han configurado los *Makefiles* apropiadamente, el flujo de generación SW es transparente al usuario. Tal y como se representa en la Figura 3-10, el usuario solo tiene que seleccionar el nivel mediante el valor de la variable LEVEL en el *Makefile* y ejecutar el comando *make* en la consola de la plataforma de desarrollo. En el nivel de especificación se produce el fichero ejecutable (.exe) para simulación, y en el de generación software, el fichero binario (.bin) a descargar en la plataforma objetivo.

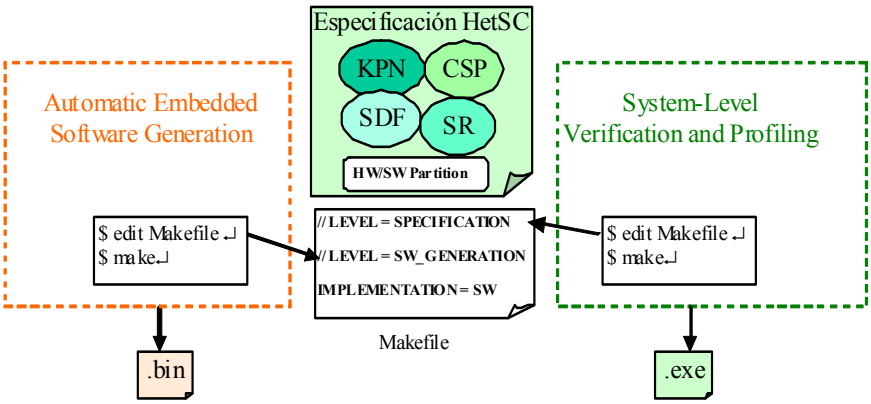



**Figura 3-10. La generación de software es automática desde la especificación *HetSC*.**

Por medio de algunas variables internas del *Makefile* se puede controlar fácilmente la conmutación entre las distintas plataformas objetivo. A la aplicación de *SWGen* para obtener binarios para diferentes plataformas objetivo se le denomina aquí *portabilidad objetivo*. De esta forma se distingue la *portabilidad objetivo* de la *portabilidad nativa* (o simplemente, *portabilidad*) que define la posibilidad de usar *SWGen* en distintas plataformas de desarrollo (en un PC con Linux, en un PC con Windows, en una estación de trabajo con Unix, etc).

Una variable del *Makefile* (**HOST**) debe definir el tipo de plataforma de desarrollo. Actualmente se define exclusivamente por el sistema operativo. Por tanto, ejemplos de las plataformas de desarrollo en *SWGen* son *Linux*, *Unix* o *Windows*. Cuantas más posibilidades, mayor es la portabilidad nativa de *SWGen*.

El *Makefile* de desarrollo también tiene una serie de variables para definir la plataforma objetivo. Cuantas más combinaciones se soporten de estas variables, mayor es la portabilidad objetivo de la metodología de generación. Estas variables son:

**OS_API:** Tipo de API para la que se genera el software embebido. Esta API debe estar soportada por el eOS de la plataforma. Se ha probado la generación de la librería para diferentes APIs: *POSIX*, API-C de *eCos y µC/OS-II*.

**OS:** Sistema operativo embebido. Ejemplos son *µC/OS-II, eCos*, *GX-Linux*, etc.

**ARCHITECTURE:** Tipo de arquitectura software objetivo (sección 3.2.1.1). Ejemplos son *ARM*, *OpenRISC*, *PowerPC* o *MIPS.*

**DEVICE:** Dispositivo de implementación. Ejemplos de dispositivos son *Carmen* de *SidSA*, *EPXA1* de *Altera* o la *CSB536FS* de *Cogent*.

Estas variables sintetizan cómo se modela la plataforma HW/SW de cara a la generación de software en la metodología propuesta. La Figura 3-11 lo esquematiza y ejemplifica gráficamente.

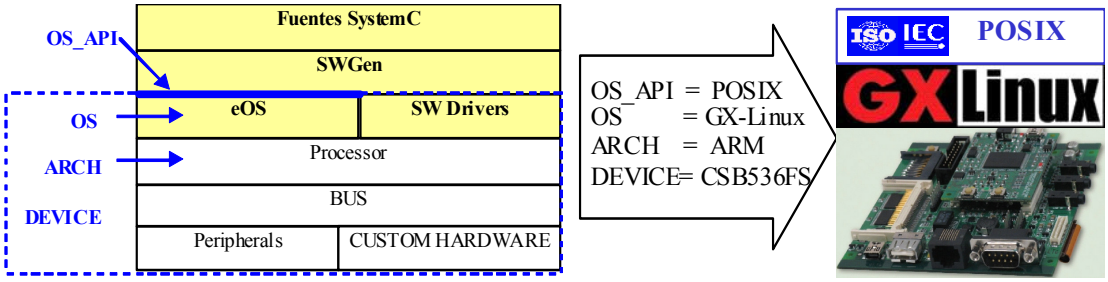



**Figura 3-11. Modelo de la plataforma HW/SW para generación de software.**

Como se muestra en la Figura 3-11**, e**l usuario sólo tiene que definir el RTOS empleado en la implementación y qué API se utiliza para acceder a sus servicios. No es preciso modificar la especificación SystemC, ni conocer el API de programación, ni qué servicios están disponibles en el RTOS. El manejo de servicios como la concurrencia se ha hecho ya en la especificación de una forma estándar y homogénea, mediante procesos SystemC. *SWGen* resuelve la implementación de la concurrencia por medio de los servicios del RTOS específico, ahorrándole al usuario ese proceso de traslación.

La variable DEVICE controla la selección de código *SWGen* que incluye llamadas a software dependiente de hardware de plataforma (*Hardware Dependent Sofware* o HdS). Este código incluye llamadas a controladores que "no pasan por" el eOS, es decir, que en aras de optimizar su rendimiento, realizan llamadas dependiente de la plataforma de implementación concreta, en lugar de llamadas al sistema del eOS empleado. En esta categoría se incluyen las interfaces HW/SW.

En *SWGen*, la variable LEVEL admite un tercer valor, denominado COSIMULATION (LEVEL=COSIMULATION). Este es un nivel de generación que permite acelerar la validación del resultado de generación sobre la plataforma de desarrollo. En la Figura 3-12 se representan distintas opciones de validación sobre la plataforma de desarrollo de la generación realizada mediante *SWGen*.

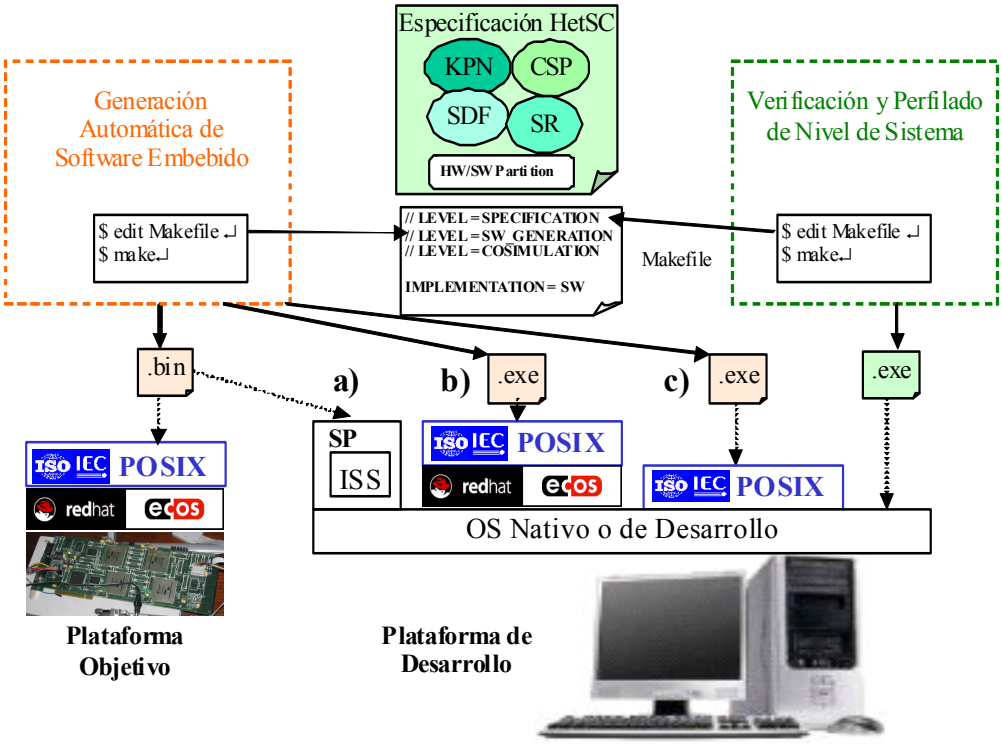



**Figura 3-12. Posibilidades de validación de la generación en la plataforma de desarrollo.**

En la Figura 3-12a se representa una técnica de validación en la que se realiza la configuración con LEVEL=SW_GENERATION y se realiza una generación para la plataforma objetivo. Una vez obtenido el binario, en lugar de descargarlo a la plataforma objetivo, se ejecuta sobre la plataforma de desarrollo mediante un simulador de plataforma (SP), que incluirá un ISS y un modelo de las llamadas de I/O. Esta es la opción de validación en plataforma de desarrollo más completa y, por tanto, más precisa. Sin embargo, es también la más costosa en tiempo de simulación.

La opción de generación con LEVEL=COSIMULATION puede reducir en uno o dos órdenes de magnitud los tiempos de ejecución obtenidos con el esquema de la Figura 3-12a. Cuando se maneja una aplicación relativamente grande, la aceleración del desarrollo se hace notable. En la Figura 3-12b se representa una posibilidad que consiste en lo siguiente. El RTOS objetivo se compila sobre la plataforma nativa. Esto es posible en un RTOS como *eCos*. Supóngase que el desarrollo se hace sobre un PC con OS nativo *Linux* (HOST=LINUX). Entonces se puede configurar *eCos* para la plataforma *i386*. De esa manera, lo que se compile contra esa configuración de *eCos* puede ejecutarse en el PC con Linux. Posteriormente, se configura *SWGen* para el API del RTOS y el RTOS objetivo, es decir OS=*ECOS* y OS_API=POSIX. Al aplicar *SWGen* a la especificación SystemC, se obtiene un ejecutable nativo, que ejercita la aplicación de la librería *SWGen*, en concreto, las llamadas al API del RTOS objetivo (API-C de eCos en este caso) y el propio RTOS objetivo (eCos en este caso). De esta forma, se puede comprobar muy rápidamente el efecto tanto de la implementación del RTOS, como de los detalles de implementación introducidos por la librería *SWGen* y relativos al RTOS, tales como la inclusión de prioridades, etc.

En la Figura 3-12c se representa un caso de uso del nivel COSIMULATION en el que OS_API = POSIX y DEVICE= HOST. Es decir, se genera un ejecutable nativo que utiliza directamente los servicios del OS de la plataforma de desarrollo o nativa. De esta forma se ejercita únicamente la aplicación de *SWGen*, sin necesidad de instalar una emulación del RTOS objetivo sobre la plataforma nativa. Esto permite una comprobación funcional muy rápida. También permite, por ejemplo, la detección de un soporte deficiente del mismo API en el eOS. En algunos ejemplos realizados, la generación en el nivel SW_GENERATION para un API POSIX producía un binario que fallaba al ejecutar en la plataforma objetivo o al realizar la validación de la Figura 3-12b. En cambio, al realizar una validación del tipo de la Figura 3-12c, el binario resultante ejecutaba sin problemas en la plataforma nativa. El análisis posterior revelaba que una llamada al sistema no estaba soportada (o, si lo estaba, que su implementación era deficiente) en el eOS. Esta detección aprovechaba el hecho de que el OS de la plataforma nativa (*Linux* en la Figura 3-12) soporte el mismo API (POSIX en la Figura 3-12) que el eOS de la plataforma objetivo (*eCos* en la Figura 3-12), pero de forma más completa y refinada. Estas limitaciones en el soporte de un API de un eOS no son extrañas dada la celeridad, especificidad y restricciones de tamaño en su desarrollo.

Por último, reseñar que en la última versión de *SWGen* se emplean varios *Makefiles*, en lugar de algunas variables del *Makefile* único. Por ejemplo, para la compilación de nivel de sistema se emplea un fichero *Makefile.sys*. Para la generación de software, se emplea un fichero del tipo *Makefile.arch-os_api* para cada combinación de arquitectura software y API de RTOS (por ejemplo, *Makefile.arm-linux* o *Makefile.host-posix*). De esta forma, se reduce la complejidad de los *Makefiles* y se desacopla y facilita la extensión del soporte de nuevas plataformas de implementación.

## *3.5 Librería de Generación Software SWGen*

### 3.5.1 Introducción

En el flujo de generación de software *SWGen,* la librería *SWGen* es un elemento esencial. El conocimiento de su estructura e implementación facilita la comprensión de la metodología. La librería *SWGen* provee la implementación de las facilidades de especificación SystemC mediante construcciones C/C++ y llamadas a los servicios de un RTOS embebido a través de una API específica. Permite obtener de forma automática una implementación eficiente para la plataforma objetivo configurada.

La implementación provista es al menos más eficiente que una que incluya todo el código de librería SystemC. Por ejemplo, en términos de velocidad, la implementación de *SWGen* es más rápida ya que la librería núcleo SystemC incluye un kernel de simulación de eventos discretos, que a su vez se ejecuta sobre la plataforma nativa. Además, las librerías de especificación de nivel de sistema (librerías SystemC y *HetSC*) se optimizan para las plataformas nativas sobre las que se ejecuta (que utilizan sistemas operativos como Unix, Linux y Windows, y arquitecturas software como Intel o Sparc). En cambio, *SWGen* se codifica para el eOS y se adapta a las herramientas cruzadas propias de la plataforma objetivo.

Además, la librería SystemC (así como la librería *HetSC*) incluye también elementos dedicados al chequeo y verificación de reglas (conexión de puertos, número

de procesos accediendo a un canal, etc). Una vez estos chequeos se realizan en la simulación de nivel de sistema, reiterar su ejecución en la implementación software deja de ser necesario en la mayoría de los casos. Dado que constituyen una sobrecarga en velocidad y tamaño, *SWGen* elimina ese tipo de código.

La librería *SWGen* es escalable. Es decir, su estructura esta diseñada para facilitar la extensión de sus capacidades y el soporte de nuevas APIs y plataformas objetivo. En la figura Figura 3-13 se presenta el árbol o estructura de la librería *SWGen*.

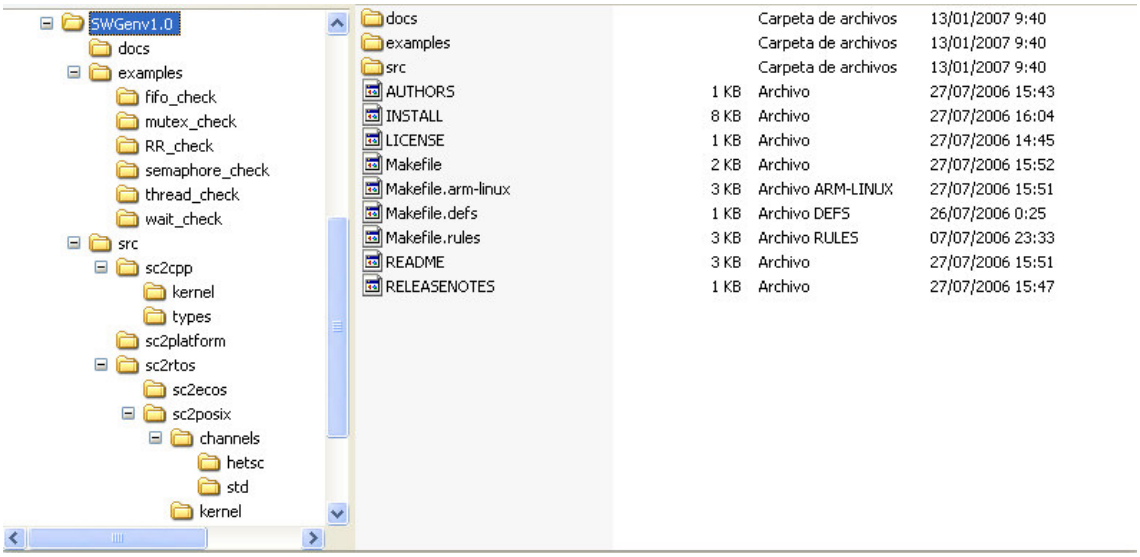



**Figura 3-13. Estructura de carpetas y archivos de la librería *SWGen*v1.0.**

En su primer nivel, la librería presenta una estructura típica, con un directorio de documentación (*/doc*), otro de ejemplos (*/examples*) y un tercer directorio de fuentes (*/src*). Cuenta también con ficheros con instrucciones de instalación (*README*), notas de versiones (*RELEASENOTES*), etc. Como se explicó al final de la sección anterior, existen varios *Makefiles,* uno por cada plataforma objetivo. De esta forma, el flujo de generación para una plataforma HW/SW concreta requiere una instalación de la librería *SWGen* con el *Makefile* específico asociado a esa plataforma HW/SW. Esa instalación implica una compilación cruzada de la librería *SWGen* para la plataforma destino, que incluye las cabeceras de la configuración del RTOS objetivo, para generar la librería objeto (*swgen.a*) y la librería fuente (*swgen.h*). La Figura 3-14 detalla el flujo de generación de software que se mostró en la Figura 3-9 y da una idea más precisa de cómo se incrusta la librería en el flujo de generación.

Se puede entender cada instalación de *SWGen* como una "configuración" de *SWGen* para una plataforma objetivo. Una configuración de *SWGen* emplea, a su vez, una configuración del RTOS. En la librería *SWGen v1.0* se provee, por ejemplo, un Makefile para plataforma basada en arquitectura ARM con eOS GX-Linux y API POSIX (*Makefile.arm_linux*). El usuario puede crear su propio *Makefile.plataforma* para otra combinación de plataforma, basándose en los disponibles.

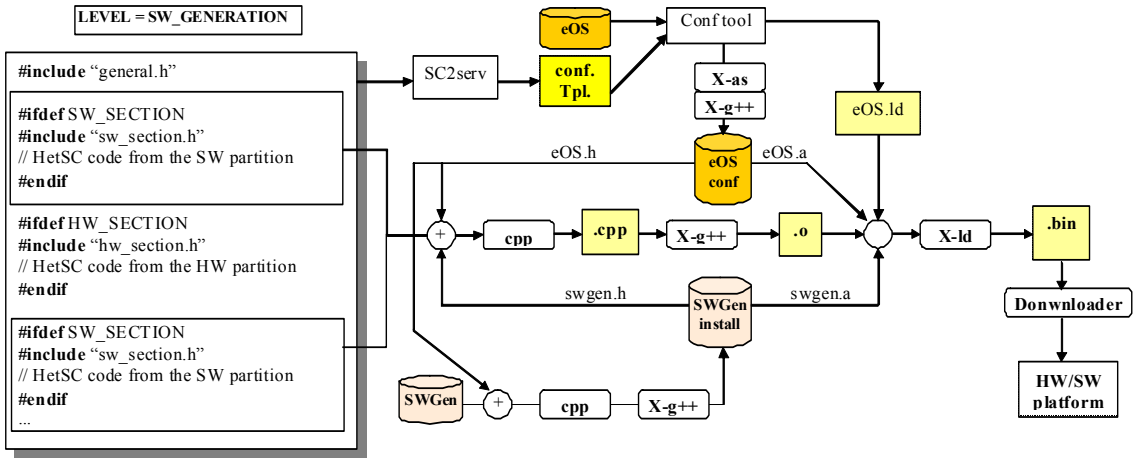



**Figura 3-14. Flujo de generación SW.**

La librería *SWGen* se divide en *paquetes de código*. Existen tres tipos de paquete, que reflejan un grado distinto de reutilización del código de librería para nuevas plataformas objetivo. Esta estructura permite sistematizar y agilizar la portabilidad objetivo de la librería *SWGen*. Retornando a la Figura 3-13, dentro del directorio de fuentes (*"/src"*), en un segundo nivel de jerarquía la librería se separa en tres directorios: */sc2cpp*[6] (de SystemC a C/C++), */sc2rtos* (de SystemC a API de RTOS) y */sc2platform* (de SystemC a plataforma). Cada uno de estos directorios se corresponde con los tres tipos de paquetes de código de *SWGen* para la generación de software: *sc2cpp*, *sc2rtos* y *sc2platform* (Figura 3-15).

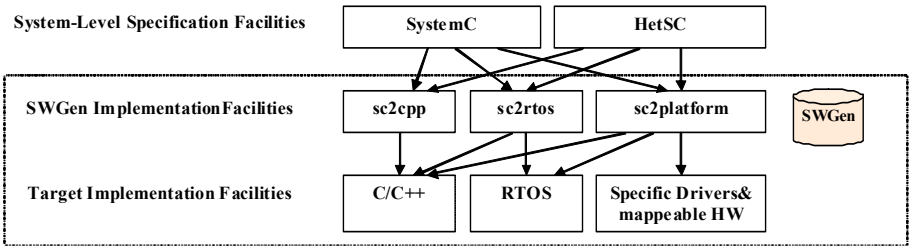



**Figura 3-15. La librería *SWGen* se divide en paquetes según la reutilizabilidad del código.**

Para realizar una configuración de *SWGen* y, por tanto, para generar código para una plataforma HW/SW concreta, se requiere un paquete de cada uno de esos tres tipos. A continuación se explica el contenido de cada tipo de paquete de código:

***sc2cpp*:** Contiene código que, para implementar facilidades SystemC (y *HetSC*) como software embebido eficiente, usa únicamente facilidades de C/C++.

***sc2rtos*:** Contiene código que, para implementar facilidades SystemC (y *HetSC*) como software embebido eficiente, usa, además de facilidades de C/C++, los servicios de un RTOS embebido a través de una de las APIs soportadas.

***sc2platform*:** Contiene código que permite implementar facilidades SystemC (y *HetSC*) como software embebido eficiente y que para ello requiere, además de usar

[6] Notar que aquí se usa cpp con el sentido "C++", en tanto que en la Figura 3-14, en la Figura 3-9 y en la Figura 3-2, cpp tomaba el sentido de "PreProcesador de C".

facilidades C/C++ e invocar a través de una de las APIs soportadas los servicios de un RTOS embebido, acceder a servicios que no pertenecen al RTOS (o que se acceden fuera de cualquiera de las API soportadas) y que, en general, dependen de la arquitectura de la plataforma (memoria, periféricos, etc) y su mapa de memoria.

Existe un solo paquete de tipo *sc2cpp*, denominado de igual forma. El código del paquete *sc2cpp* es el del directorio */sc2cpp*. Todo el código del paquete *sc2cpp* es común y reutilizado para cualquier plataforma HW/SW objetivo.

Existen tantos paquetes de tipo *sc2rtos* como APIs de RTOS se soporten. Es decir, el soporte de las APIs C de *eCos*, de *µC/OS-II* y POSIX implican que la librería *SWGen* cuente con tres paquetes de código de tipo *sc2rtos: sc2ucosii*, *sc2ecos* y *sc2posix*. Estos paquetes están contenidos en el directorio */sc2rtos*. Cada paquete de código está contenido en su directorio correspondiente (*/sc2ucosii, /sc2ecos* y */sc2posix*). Los servicios soportados por esta parte de la librería *SWGen* son los requieren que la plataforma objetivo cuente con, además de las herramientas de desarrollo cruzado C/C++, también con un RTOS embebido que soporte al menos una de las APIs de programador entre todas las soportadas por *SWGen*. Nótese que el requisito se refiere a una API específica y no a un RTOS específico. Si bien algunas APIs de RTOS están muy ligadas a un RTOS embebido concreto (por ejemplo, el API C de *eCos*), el empleo de otras APIs más genéricas, como POSIX, permite una generación para múltiples RTOS que soportan esa API. Por ejemplo, tanto *eCos* como *GX-Linux* (una distribución de Linux embebido) ofrecen acceso a sus servicios a través de un API POSIX. Por lo tanto, SWGen se puede usar para ambos RTOS una vez que ya soporta el API POSIX.

Es importante notar también que cada paquete de tipo *sc2rtos* no requiere al eOS un soporte exhaustivo del API. Por ejemplo, el paquete *sc2posix* no invoca todas las posibles llamadas al API POSIX, sino un subconjunto de llamadas correspondiente a un número limitado de servicios relativamente comunes. Esta acotación, por un lado, muestra que la extensión de *SWGen* a otras APIs de RTOS exige un esfuerzo limitado y abordable en corto plazo. Por otro lado, posibilita un rango amplio de plataformas objetivo. Aunque muchas plataformas objetivo pueden soportar un RTOS determinado, no todas lo hacen con la misma completitud. Sin embargo, es de esperar que una gran mayoría soporte al menos los servicios básicos que requiere *SWGen*.

| POSIX services | POSIX section | Concurrency &delays | Communication | |
|---|---|---|---|---|
| | | | SW/SW | HW/SW |
| Synchronization | 11 | used | | used |
| Clocks and Timers | 14 | used | | |
| Message Passing | 15 | | used | used |
| Thread Management | 16 | used | | |

**Figura 3-16. Servicios POSIX requeridos para el soporte del paquete sc2posix de la librería *SWGen*.**

Los servicios requeridos por la librería *SWGen* están identificados. En la Figura 3-16 se indican las secciones del API POSIX involucradas en el soporte de este API en *SWGen*. Si un RTOS embebido soporta estas secciones POSIX, entonces ese RTOS embebido es inmediatamente soportado como parte de la plataforma objetivo. Es decir, la librería *SWGen* es capaz de generar el código de SW embebido para los servicios asociados al paquete *sc2cpp* y al paquete *sc2posix*. Esta identificación, en el caso

POSIX se ha refinado incluso al nivel de llamada al sistema, habiéndose visto que tampoco se requiere al eOS un soporte completo de cada una de las secciones.

En resumen, la acotación e identificación de los servicios requeridos por *SWGen* al eOS agiliza y sistematiza la aplicación de *SWGen* a nuevos RTOS y a nuevas APIs de RTOS. En cualquier caso, un soporte amplio de POSIX por parte del RTOS embebido es conveniente, ya que puede ampliar las posibilidades de selección entre diferentes implementaciones en un futuro. Por ejemplo, el soporte de la sección 3 de POSIX (relativo a *Primitivas de Proceso*), podría permitir configurar la librería *SWGen* para mapear los procesos SystemC, bien a procesos SW, o bien a hilos en SW, en función de la arquitectura de la plataforma (monoprocesador o multiprocesador), partición HW/SW, requerimientos de velocidad, etc.

El paquete de tipo *sc2platform* provee la implementación de los servicios que requieren, bien llamadas a controladores sin pasar por el API de RTOS, o bien la visibilidad de la arquitectura hardware (mapa de memoria, acceso a periféricos, interfaces HW/SW, etc). En general, eso requiere un soporte específico por cada definición de la plataforma objetivo. Por tanto, existirá un paquete por cada plataforma concreta. Por ejemplo, para las plataformas *HSDT100* de *SidSA* y *CSB536FS* de *Freescale*, los paquetes *sc2hsdt100* y *sc2csb536fs*, cuyo código está contenido en sus directorios respectivos (por ejemplo, */sc2hsdt100* y */sc2csb536fs*), serán los que proveerán el código de implementación específico de esas plataforma respectivamente.

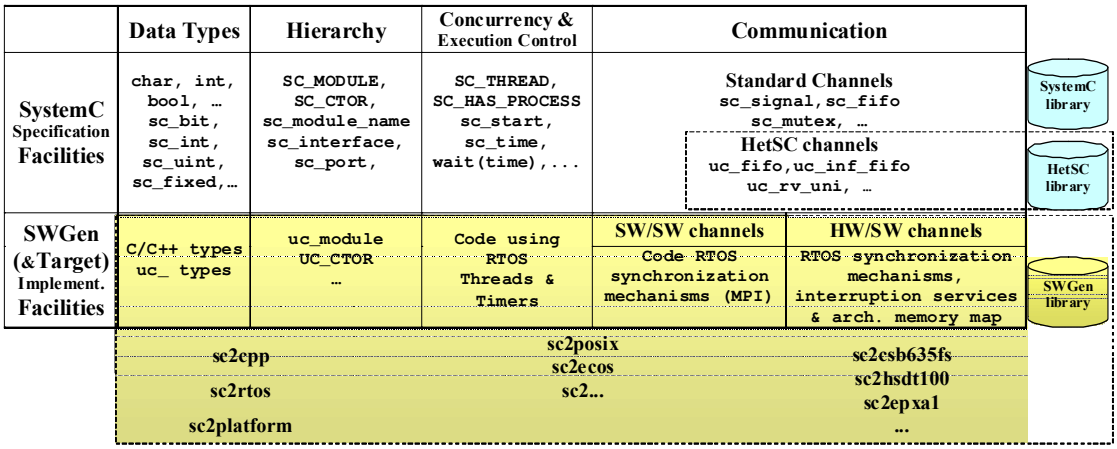



**Figura 3-17. Estructura de la Librería de Generación de SW según los servicios implementados.**

La Figura 3-17 identifica la facilidades de especificación y de implementación SW en función del tipo de servicio que soportan en la especificación (en columnas). Se han distinguido cuatro grupos de servicios: Tipos de Datos, Jerarquía, Concurrencia y Control de la Ejecución, y Comunicación. En filas se distingue el nivel de abstracción en la metodología de diseño. En la fila superior se refleja el nivel de especificación (LEVEL=SPECIFICATION), por tanto, en ella aparecen las facilidades SystemC (y *HetSC*) usadas en la especificación de sistema para cada tipo de servicio. Estas facilidades cuentan con una "implementación de sistema", es decir, la implementación SystemC que permite que la especificación sea ejecutable por el simulador SystemC. En la fila inferior, se muestra el nivel de generación (LEVEL=GENERATION). Por lo

tanto, en esa fila aparecen las facilidades provistas por la librería *SWGen* para la implementación eficiente de cada de servicio.

Por ejemplo, el soporte de distintos tipos de datos se resuelve mediante tipos C/C++ y SystemC en especificación. Estos, en cambio, son trasladados a tipos C/C++ o bien implementaciones específicas realizadas por la librería *SWGen* (tipos *uc_*). En el servicio de comunicación se contemplan tanto las comunicaciones entre procesos de la especificación como la entrada y salida del sistema. La entrada y salida se entiende como un caso particular de comunicación, en concreto, la comunicación del sistema con el entorno. En el nivel de especificación, la entrada salida puede ser a través de puertos o *exports*. En el primer caso, el sistema solo tiene la interfaz de acceso. En el segundo caso, el sistema incluye los canales de entrada/salida, de manera que el módulo de sistema y entorno se pueden comunicar directamente. En el nivel de generación de SW, esto se traduce en distintas implementaciones de canal de I/O, que se encontrarán dentro de las distintas categorías de canales de implementación (canales SW/SW y SW/HW).

La Figura 3-17 recoge en su parte inferior la correspondencia entre los tipos de servicios, las facilidades de implementación de la librería *SWGen* y los paquetes en las que se integran. El contenido del paquete *sc2cpp* se ha agrupado en dos subpaquetes, *types* y *kernel*, cuyo código se incluye en los subdirectorios */types* y */kernel* respectivamente (ver Figura 3-13). Cada uno de los subpaquetes se corresponde con el tipo de servicio. El subpaquete *types* contiene clases con una implementación SW para lo que eran tipos SystemC en especificación. El paquete *kernel* contiene fundamentalmente clases con la implementación software de estructuras referidas a la introducción de jerarquía en la especificación (puerto, módulo, etc).

Los paquetes de tipo *sc2rtos* también se estructuran en subpaquetes. Por ejemplo, el paquete *sc2posix* contiene dos subpaquetes: *kernel* y *channels*. El subpaquete *kernel* contiene el código de implementación software de facilidades de nivel de especificación que requieren servicios relativos a la concurrencia y a la temporización mediante el reloj del sistema (por ejemplo, bloqueo de la tarea durante cierto tiempo). El subpaquete *channels* contiene otros dos subpaquetes, uno con código de generación SW para los canales estándar SystemC y otro de generación para los canales provistos por la librería *HetSC*. El que el código de implementación de canales *HetSC* esté en un paquete aparte proporciona mayor modularidad a la librería *SWGen*. De esa forma, es posible aplicar *SWGen* a una especificación SystemC que respete la metodología de especificación general de HetSC que no necesite la instalación de la librería *HetSC*.

### 3.5.2 Especificación de la partición HW/SW

La metodología de generación de SW precisa, además de la propia especificación, información acerca de la partición HW/SW. Esta información es necesaria también en actividades de diseño tales como el perfilado de nivel de sistema. En la Figura 3-18 se da una representación gráfica de una partición HW/SW de una especificación *HetSC*. Actualmente, la metodología *SWGen* añade la información de la partición HW/SW a la especificación en el mayor nivel de jerarquía. Los módulos de ese nivel no pueden verse atravesados por la partición. En cambio, la partición puede atravesar canales. Una partición puede contener o no un canal que la conecta con la otra partición.

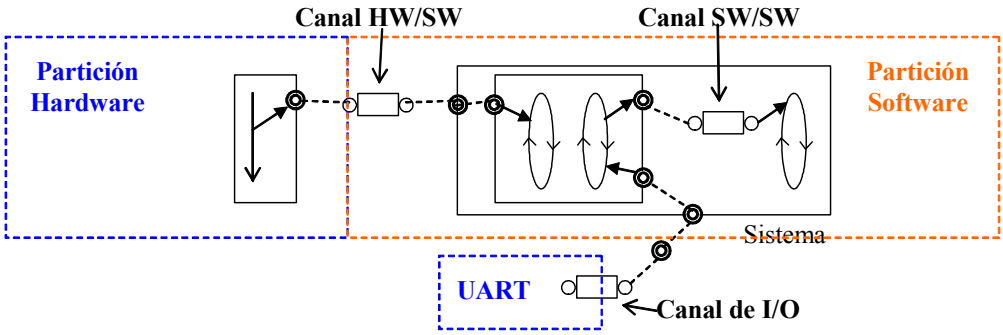



**Figura 3-18. Canales HW/SW, SW/SW y de Entrada/Salida.**

Para especificar la partición HW/SW se utilizan cláusulas de preprocesado y la inclusión de ficheros de cabecera. El código correspondiente a la partición SW se tiene que incluir entre la cláusula *#ifdef SW_SECTION* y la cláusula *#endif*. Además, detrás del *#ifdef*, es preciso incluir una cabecera correspondiente a la sección. Es decir, dentro de un *SW_SECTION*, hay que realizar un *#include "sw_section.h"*. Como se ejemplifica en la Figura 3-19, es posible dividir el código de cada partición en secciones de HW, de SW y de entorno.

```
#include "general.h"

#ifdef SW_SECTION
#include "sw_section.h"
// HetSC code from the SW partition
#endif

#ifdef HW_SECTION
#include "hw_section.h"
// HetSC code from the HW partition
#endif

#ifdef SW_SECTION
#include "sw_section.h"
// HetSC code from the SW partition
#endif
...
```

**Figura 3-19. Especificación de la partición HW/SW mediante cláusulas de preprocesado.**

### 3.5.3 Sustitución de Librería Metodológica

Como ya se ha adelantado, la sustitución de librería metodológica se realiza a través de *Makefile*, cambiando el valor de la variable LEVEL. El fichero *general.h* contiene definiciones tanto para el nivel de especificación como para el nivel de generación de software. Es el que se encarga de incluir las librerías de nivel de sistema (SystemC y *HetSC*) cuando LEVEL=SPECIFICATION, mientras que, al realizar el preprocesado con la variable LEVEL=SW_GENERATION, las construcciones del código de especificación encuentran su implementación en la librería *SWGen*, en lugar de en las librerías SystemC y *HetSC*. El fichero *"sw_section.h"* solo tiene efecto en el nivel de generación de software, incluyendo una serie de redefiniciones de nombres, que sustituye nombres de macros, clases y plantillas SystemC (prefijos "sc_" o "SC_") y HetSC (prefijos "sc_" o "SC_"), por nombres de macros, clases y plantillas de

implementación (que suelen presentar los prefijos "*uc_*" y "*UC_*" y/o sufijos "*_SS*", "*_SH*" o "*_SS*").

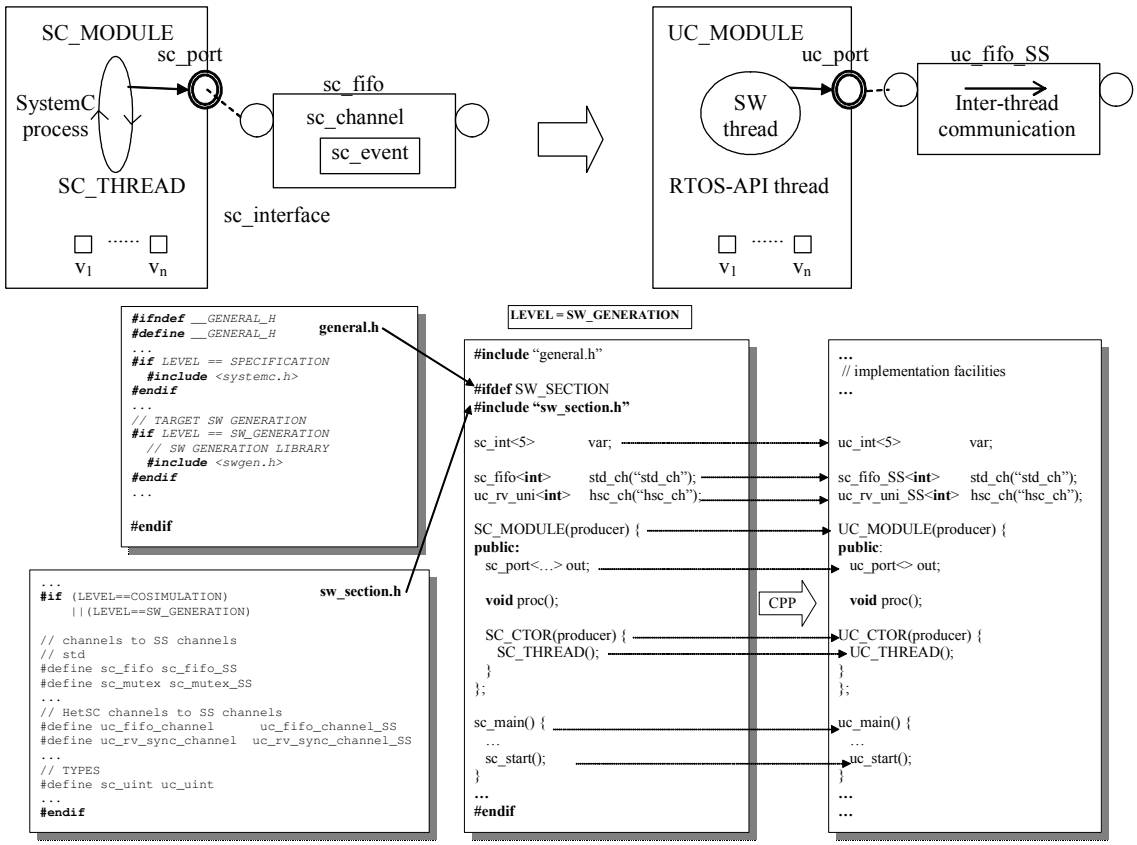



**Figura 3-20. Sustitución de las facilidades SystemC por facilidades de implementación *SWGen.***

De esta forma, cuando LEVEL=SW_GENERATION, el preprocesado con la inclusión de los ficheros *general.h* y *sw_section.h* produce una versión como la que se muestra en la parte derecha de la Figura 3-20, que usa facilidades de implementación *SWGen*. Aunque muchas de las sustituciones de nombres definidas en "*sw_section.h*" no son necesaria para la generación, éstas hacen más distinguible, a la vez que reconocible, el producto de la generación con respecto al código de especificación. Por ejemplo, en la librería *SWGen*, se podría haber mantenido el nombre *sc_start* para la clase de implementación correspondiente. Sin embargo, la función *sc_start* es sustituida por una función denominada *uc_start*, declarada e implementada por la librería *SWGen*. En cualquier caso, la implementación *SWGen* de esa función es distinta de la provista por la librería SystemC para la función *sc_start*.

Las secciones siguientes mostrarán de forma más detallada cómo se realiza la implementación software por medio de las distintas facilidades de implementación de la librería *SWGen*. Esta exposición se dividirá según grupos de servicios.

## 3.5.4 Implementación de Tipos de Datos

En esta sección se reseña como *SWGen* afronta la implementación software de tipos de datos. La especificación de nivel de sistema puede incluir tanto tipos C/C++

(por ejemplo, *int*, *float*, *double*, etc), como tipos específicos de SystemC (*sc_uint*, *sc_int*, *sc_fixed*, etc).

Los tipos C/C++ suelen ser una entrada válida tanto para el compilador nativo, con el que se genera la especificación ejecutable, como para el compilador cruzado disponible para la plataforma objetivo. El compilador cruzado trata de forma óptima esos tipos, teniendo en cuenta la anchura de palabra de la arquitectura objetivo. Sin embargo, los tipos SystemC no son reconocidos como tipos por el compilador cruzado. La librería *SWGen* esta al cargo de proveer una implementación automática y eficiente en SW de los tipos específicos de SystemC.

Una solución posible para automatizar la traslación de los tipos específicos es reutilizar el código implementado por la librería núcleo SystemC. Por ejemplo, para implementar en SW el tipo *sc_uint* se puede reutilizar la implementación que se encuentra en los ficheros fuente de la librería núcleo de SystemC (tomando código de ficheros como *sc_uint.h*, *sc_uint_base.h*, *sc_value_base.h*, etc de los directorios */src/sysc/datatypes/misc* y */src/sysc/datatypes/int*).

La primera cuestión a considerar en esa reutilización se refiere a que el código de implementación de nivel de sistema está adaptado a la plataforma de desarrollo nativa y no necesariamente a la plataforma objetivo. Este inconveniente es fácil de evitar si el código de implementación de los tipos SystemC es lo suficientemente genérico y parametrizable mediante variables que modelen la plataforma de implementación. Por ejemplo, a continuación se muestra la definición de una serie de tipos empleados internamente en la librería SystemC para la implementación de tipos SystemC visibles:

```
typedef implementation-defined int_type;
typedef implementation-defined uint_type;
typedef implementation-defined int64;
typedef implementation-defined uint64;
```

Es decir, dependiendo de la plataforma nativa en la que se instala la librería SystemC, se emplea una definición u otra. Por ejemplo, para una plataforma nativa Windows de 32 bits:

```
typedef long long int_type;
typedef unsigned long long uint_type;
typedef int64 int64;
typedef uint64 uint64;
```

La idea es sustituir la definición de esos tipos para la implementación en plataforma nativa, por una definición acorde con la de la plataforma objetivo en generación de software. Esta definición puede no ser trivial. Guardar la equivalencia entre la implementación de nivel de sistema y la implementación software requiere conocer el tamaño de la implementación del tipo básico en la plataforma nativa y el tamaño en la plataforma objetivo. En el primer caso, lo fija la configuración de SystemC para esa plataforma y el compilador nativo empleado. En el segundo caso, lo fijaría la librería SWGen y el compilador cruzado empleado. Habría que tener en cuenta además que los rangos en los que se pueden mover las plataformas nativas (de 32 o 64 bits), y las plataformas objetivo (usualmente desde 8 a 32 bits). Esto requiere un estudio específico que determine una configuración equivalente para cada combinación plataforma-nativa/compilador-nativo/plataforma-objetivo/compilador-cruzado.

Otra cuestión que plantea *SWGen* es si, una vez demostrado que la implementación SystemC es reutilizable para la implementación software en la plataforma objetivo, ésta reutilización fidedigna es la más eficiente. Es decir, una vez provista una implementación objetivo equivalente en términos de precisión y funcionalidad, se plantea si existen implementaciones del tipo más eficientes en tamaño, velocidad y consumo, y aceptables en términos de grado de equivalencia con la implementación del sistema. Más aún, en algunos casos, podría ser interesante sacrificar una equivalencia del 100% entre la implementación SW y la de sistema. Por ejemplo, se podría perder precisión fuera de cierto rango, en aras de mejorar parámetros de rendimiento.

Por tanto, la implementación de tipos es un problema de compromiso entre los siguientes factores:

- el grado de equivalencia entre la implementación del tipo de nivel de sistema y la implementación SW.
- la eficiencia de la implementación en tamaño de código y velocidad.

En *SWGen* se da una variación de este compromiso cuando se pasa del nivel de especificación al de generación. La estructura de clases que implementa los tipos específicos de SystemC es compleja y pesada, lo que afecta al rendimiento. Por ejemplo, en el nivel de sistema, la clase *sc_uint*, que es una clase primitiva del lenguaje, hereda la clase de implementación *sc_uint_base*, que a su vez, hereda la clase *sc_value_base*. Además, por definición, el manual de referencia de SystemC obliga a que los tipos internos tengan un tamaño mínimo de representación de 64 bits (tipos *int_type* y *uint_type*) y exacta de 64 bits (tipos *int64* y *uint64*). En contraste, las plataformas objetivo usualmente no superan los 32bits.

La cuestión de la precisión de los tipos ilustra que existen varias posibilidades en la implementación de tipos. En este caso, se contemplan dos posibilidades en las que se gana eficiencia en tamaño de código y velocidad de ejecución a costa de una pérdida de equivalencia en la implementación software.

Por un lado, es posible que la generación conlleve una disminución de la precisión. En efecto, un caso posible en el flujo de diseño es contar con una plataforma de desarrollo de 64 bits y que se quiera generar software para una plataforma embebida de 32 bit. En este caso, para aplicaciones en las que interese asegurar la equivalencia, una solución es admitir un paso de refinamiento del código. Por ejemplo, en un primer paso se especifica el algoritmo con tipos *unsigned int de C/C++*. Imagínese que se prueba la simulación en una plataforma de desarrollo basada en un AMD Athlon 64, con sistema operativo con soporte de 64 bits. La compilación de la librería SystemC ya manejará internamente esos tipos. Sin embargo, los resultados podrían diferir al generar SW para una plataforma objetivo basada en ARM (32bits). Por tanto, una vez comprobado el algoritmo, se genera una versión de nivel de sistema refinada, en la que todas las declaraciones de *unsigned in*t se sustituyen por sc*_uint<32>*. Si tras ese refinamiento, el funcionamiento del algoritmo es satisfactorio, es de esperar que, funcionalmente, la implementación SW, que implementa el sc*_uint<32>* como un *unsigned int* del ARM 32 bits, lo sea dado que la precisión interna manejada por la clase de implementación es la misma que la de la clase de especificación. En esta solución lo

que se hace es mover el refinamiento a la fase de especificación de sistema. En muchos casos, este tipo de refinamiento es considerado una actividad propia de la fase de especificación.

Por otro lado, es posible que la generación de software implique un aumento de la precisión. En general, esta inequivalencia no se considera un problema. Imagínese que en una especificación se plantea reutilizar la especificación de un filtro. Ese filtro está escrito directamente como una especificación que en otro momento se utilizó para un DSP de 16 bits, donde los coeficientes y otras variables tienen una precisión limitada. Por tanto, en la especificación los coeficientes se declararon como *sc_uint<16>*. Supóngase que en la fase de exploración de diseño, para la aplicación actual hay requisitos relajados de velocidad y se concluye que la implementación más eficiente de ese filtro es en SW, donde la plataforma objetivo está, por ejemplo, basada en arquitectura software ARM de 32 bits. Por lo tanto, una opción es refinar el código y cambiar las declaraciones *sc_uint<16>* por *unsigned int,* que en la arquitectura objetivo es de 32 bits. En la mayoría de los casos, que los coeficientes tengan una mayor anchura de palabra, y por tanto mayor precisión, no será un problema funcionalmente, ya que no afecta a la estabilidad del filtro, en tanto que otorga eficiencia al código generado. No obstante, con las facilidades de implementación provistas por *SWGen* es también posible preservar la precisión, de tal manera que la clase de implementación *uc_uint<16>* emplee exactamente ese tamaño verificado en especificación. Esto podría ser interesante si se pretende usar un banco de pruebas de precisión de bit. En cualquier caso, una implementación alternativa a la reutilización de código SystemC es proveer una implementación menos pesada. Esa implementación puede eliminar código de chequeo, disminuir precisión al tipo en implementación respecto a su versión en especificación o eliminar métodos de acceso y, por tanto, parte de las operaciones soportadas para ese tipo, es decir, funcionalidad.

En resumen, la pérdida o ganancia de precisión, así como la pérdida de funcionalidad suponen la pérdida de equivalencia entre el tipo de especificación y el de implementación. Sin embargo, en muchos casos esto puede ser interesante en aras de objetivos de prestaciones. *SWGen*, de hecho, sacrifica por defecto algunas de esas características de los tipos SystemC en la obtención de un software eficiente. Además establece una jerarquía en la tipología de código que se puede eliminar en la implementación SW y que está presente en el nivel de sistema. Se presenta a continuación, comenzando por el tipo de información antes eliminable en la implementación software:

1) Código de chequeo en el uso del tipo.
2) Código que afecta a la semántica del tipo. Dentro de esta tipología, se distingue el código que afecta:
   A) A la precisión de manejo interno del tipo.
   B) A la semántica de saturación.
3) Código que afecta a la funcionalidad del tipo. Con esto se refiere a la posible eliminación en implementación del soporte de determinados métodos de acceso y operaciones (selección de bit, selección de parte, desplazamientos, suma, etc).

Es decir, en *SWGen* se preserva la equivalencia funcional antes que la precisión, y la precisión antes que el código de chequeo. Es posible, en cualquier caso, que en determinadas aplicaciones sea de interés un orden de prioridades distinto en la preservación/eliminación de las características de los tipos. Por ejemplo, en una aplicación puede ser vital la precisión de manejo de un tipo y, sin embargo, no importar si se mantienen todos los operadores disponibles en el nivel de sistema, ya que sólo se usan dos en la especificación (por ejemplo, suma y resta). En general, las configuraciones óptimas que controlan la transformación de la implementación de los tipos en generación de software cambian, dependiendo del tipo de aplicación.

Por lo tanto, la idea es que *SWGen*:

- Debe ser suficientemente flexible como para admitir una configuración de la generación de tipos que permita establecer si se da equivalencia en la implementación de tipos, si se sacrifica precisión, si se omiten los chequeos, etc.
- Fija una configuración por defecto de esa generación, que fija qué se elimina, qué se preserva y cómo se implementa. Específicamente, por defecto, se elimina el código de tipo 1), el de tipo 2.A) se transforma a la precisión propia de la anchura de palabra de la plataforma y el resto, de 2.B) y 3) se respeta.
- Permite fijar una implementación SW equivalente en términos de precisión y/o funcionalidad.
- Admita un flujo de síntesis, que con o sin restricciones (por ejemplo, equivalencia funcional), encuentre una configuración óptima en rendimiento de velocidad u otros parámetros de rendimiento.

### 3.5.5 Implementación de Jerarquía

La Figura 3-21 esquematiza qué facilidades de implementación SW de la librería *SWGen* implementan las facilidades de especificación visibles que caen dentro de la partición software de la especificación del sistema. Entre ellas están los canales y los procesos, tratados en secciones posteriores. En ésta sección se trata la generación de una implementación software de las facilidades que soportan una estructuración jerárquica de ámbitos de visibilidad, es decir, los módulos, los puertos y los exports.

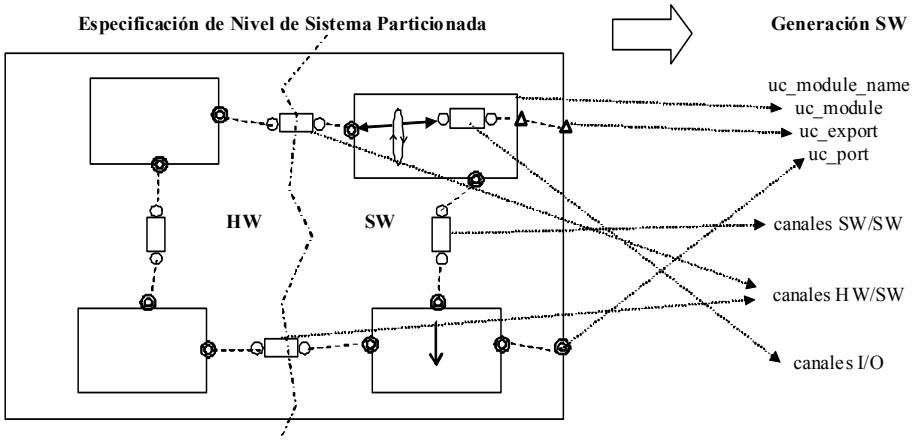



**Figura 3-21. Facilidades de implementación SW a partir de la partición SW de la especificación.**

Para soportar dicha jerarquía, la librería SystemC provee clases y macros específicas. El siguiente extracto de código de la librería SystemC muestra la declaración de las macros SC_MODULE y SC_CTOR y de la clase *sc_module*.

```
#define SC_MODULE(user_module_name) \
struct user_module_name : ::sc_core::sc_module

#define SC_CTOR(user_module_name)    \
            typedef user_module_name SC_CURRENT_USER_MODULE; \
 user_module_name( ::sc_core::sc_module_name )

class sc_module : public sc_object, public sc_process_host {
    friend class sc_module_name;

... // friend classes (not shown)

public:
    sc_simcontext* sc_get_curr_simcontext() { ... }

... // rest not shown

private:

... // rest not shown

};
```

En la declaración de la clase *sc_module* hay una parte visible que está fijada por el estándar, en tanto que otra es propia de la implementación que da la librería OSCI SystemC. Nótese que parte del código puede ser público (y, por tanto, accesible desde una aplicación SystemC) pero, sin embargo no estar fijado por el estándar. Ese código puede variar en las diferentes implementaciones de SystemC. Esto incluye a las herramientas SystemC provistas por desarrolladores diferentes y a las sucesivas versiones de la librería SystemC proporcionada por la OSCI. Por ejemplo, en la versión 2.2 de SystemC, la clase *sc_module* (clase estándar) hereda la clase *sc_object*. En cambio, esta clase no se utiliza en la implementación de la versión 2.1.

La librería *SWGen* declara, mantiene e implementa los mismos elementos que aparecen estándar SystemC, teniendo en lo demás libertad para realizar la implementación SW. Para la implementación de las facilidades SC_CTOR, SC_MODULE, sc_module, etc, *SWGen* provee facilidades de implementación SW que, básicamente se limitan a ofrecen un servicio de encapsulamiento. De esta forma, mantienen los ámbitos de visibilidad definidos en la especificación SystemC con un costo mínimo en tamaño de código. A continuación se muestran extractos de esas facilidades de implementación eSW. En el fichero *swgen.h*:

```
#define sc_module_name uc_module_name
#define SC_MODULE(user_module_name) UC_MODULE(user_module_name)
#define SC_CTOR(user_module_name) UC_CTOR(user_module_name)
```

De los ficheros *uc_module_name.h* y *uc_module.h* correspondientes al paquete de código *sc2cpp*:

```
class uc_module_name {
public:
  uc_module_name(const char* ) {;};
```

```
  uc_module_name(const uc_module_name&) {;};
};

class uc_module {
  friend class execution_context;
};

#define UC_MODULE(user_module_name) \
struct user_module_name : public uc_module

#define UC_CTOR(user_module_name)    \
       typedef user_module_name UC_CURRENT_USER_MODULE; \
       user_module_name( uc_module_name )
```

La macro UC_MODULE es similar a la clase SC_MODULE y permite declarar ámbitos de módulos en la implementación software por medio de la clase *uc_module*, que sustituye a la clase *sc_module*. Sin embargo, la clase *uc_module* es una clase casi vacía, que no necesita heredar ninguna clase adicional, ni implementar un conjunto tan extenso de métodos como la clase *sc_module*, tales como *sc_get_curr_simcontext()*, *before_end_of_elaboration()*, *construction_done()*, etc. Muchos de esos métodos se refieren al control del simulador de SystemC, que es una parte del código que se elimina en la generación de SW. La clase *uc_module* únicamente declara a la clase *execution_context* como clase amiga, que es una facilidad de implementación interna de la librería *SWGen* que se explicará más adelante.

La librería *SWGen*, provee una implementación básica para la clase *uc_module_name*. De esta manera, es posible declarar la clase de implementación del módulo (*uc_module*) con la misma estructura que la clase módulo SystemC (*sc_module*), que requiere un parámetro de tipo *string* en construcción para identificar la instancia del módulo. En la implementación SW, esos nombres, en principio, no se usan para nada y, de hecho, la clase de implementación *uc_module* no utiliza ni reserva espacio en memoria para ellos. Esto refleja la filosofía de la librería *SWGen*, que añade elementos para automatizar la generación de SW directamente desde el código de especificación, pero minimizando y limitando el empleo de recursos a lo necesario en la implementación eSW.

Actualmente, también se elimina en la implementación eSW el uso de modificadores de ámbito. En las últimas versiones de SystemC, muchas estructuras de código se encierran en ámbitos. Por ejemplo, la clase *sc_module_name* se encuentra en el ámbito *sc_core*. Actualmente, la librería *SWGen* no emplea ámbitos. No obstante, la librería podría estructurarse en ámbitos en un futuro.

La librería *SWGen* también provee una implementación eSW para el código SystemC utilizado para interconexión entre módulos, proveyendo una implementación concisa de puertos (*sc_port*) y exports (*sc_export*). El siguiente extracto de código ejemplifica la complejidad de las clases y el volumen de código que puede acarrear la implementación SystemC de una clase que representa un concepto tan simple y compacto como el puerto:

```
class sc_port_base : public sc_object {...
    ...
    virtual void before_end_of_elaboration();
```

```
    virtual void end_of_elaboration();
    virtual void start_of_simulation();
    virtual void end_of_simulation();
    void report_error( const char* id, const char* add_msg = 0) const;
    ...
};

template <class IF> class sc_port_b: public sc_port_base {
public:
    typedef sc_port_base  base_type;
    typedef sc_port_b<IF> this_type;
public:
    ...
    IF* operator -> ();
    const IF* operator -> () const;
    ...
};

template <class IF, int N = 1> class sc_port: public sc_port_b<IF> {
    typedef sc_port_b<IF> base_type;
    typedef sc_port<IF,N> this_type;
public:
    // constructors
    sc_port() : base_type( N ) {}

    explicit sc_port( const char* name_ ) : base_type( name_, N ) {}
    explicit sc_port( IF& interface_ ) : base_type( N ) {
    ...
    virtual const char* kind() const
        { return "sc_port"; }
private:
   ...
};
```

Como puede comprobarse, la implementación SystemC de la plantilla de clase *sc_port<IF,N>* hereda otra plantilla de clase, *sc_port_b*, que a su vez, hereda otra clase, *sc_port_base*, que a su vez hereda la clase *sc_object*. Si además se analiza el contenido de esas clases, se puede observar que algunas proveen métodos (tales como *before_end_of_elaboration*, *end_of_elaboration*, etc) que, como ocurría con la clase *sc_module*, sólo tienen interés de cara al control de la simulación. Análogamente, otros métodos ofrecen información estructural o de la especificación. Por ejemplo, el método *kind()* en la plantilla de clase *sc_port* permite obtener el tipo de puerto. En definitiva, la implementación SystemC de la plantilla de clase *sc_port<IF,N>* tiene un conjunto de código que es prescindible en la implementación SW.

Por eso, de nuevo, *SWGen* sustituye las facilidades SystemC, en este caso, la implementación SystemC de plantilla de clase *sc_port* por una implementación SW menos pesada, la clase *uc_port*, que contiene aún aquellos métodos de acceso y posibilidades de uso *básicos* en el nivel de especificación. Por usos básicos se pueden entender la conexión y los accesos a puerto desde los procesos. A continuación se muestra un extracto del código de la clase *uc_port* (tomado de los ficheros *swgen.h*, *uc_port.h* y *uc_port.cpp* correspondientes al paquete de código *sc2cpp*).

```
#define sc_port  uc_port
template<class IF> class uc_port{
public:
  // binding among the same hierarchy
  void  operator()(IF& channel_interface_port);

  // binding between child and parent port
  void  operator()(uc_port<IF> &parent_port);

  IF *access();
  IF *operator->();
private:
  IF *port_interface;
  uc_port<IF> *upper_port;
  bool attached_to_a_parent_port;
};

...
template<class IF> IF *uc_port<IF>::access() {
  if (attached_to_a_parent_port) {
    return upper_port->access();
  } else {
    return port_interface;
  }
}
...
```

Esta clase de implementación SW elimina la necesidad de transformar el código de los procesos de la especificación SystemC, ya que no es preciso sustituir los accesos a las instancias de puerto por llamadas al RTOS para sincronización y comunicación. El ejemplo mostrado es un soporte básico, ya que no muestra la implementación de métodos de conexión de array de puertos[7]. En el extracto de código se reproduce la implementación del método más costoso en cómputo de la clase (*access*), como muestra del reducido tamaño de código y nivel de optimización en términos de rendimiento que presenta esta solución. Un manejo parecido se puede extrapolar a la clase *sc_export*.

### 3.5.6 Implementación de Concurrencia y Control de la Ejecución.

La concurrencia y el control de la ejecución son servicios básicos que la librería SystemC ofrece al diseñador. Hay varias facilidades básicas del lenguaje dedicados en la metodología de especificación al soporte de la concurrencia. Las macros estándar SC_THREAD y SC_METHOD sirven para declarar procesos. Otras facilidades de especificación estándar se utilizan para controlar la simulación. Estas facilidades son las funciones *sc_main()*, que encierra la instanciación del módulo de sistema y los de entorno, y la función *sc_start()*, que permite arrancar la simulación de la especificación. Todas estas facilidades encuentran una interpretación en software mediante la librería *SWGen*.

[7] La implementación actual de la librería *SWGen* tiene un carácter de demostrador. La completitud de todas las funcionalidades propuestas y descritas en este trabajo de tesis requiere un esfuerzo adicional que se justificaría en un marco de desarrollo industrial de esta herramienta.

Hay una serie de macros y clases de implementación que tienen una asociación directa con las macros y clases SystemC estándar (por ejemplo, la macro UC_THREAD para la implementación software de la macro SC_THREAD). Además, *SWGen* contiene una estructura de facilidades de implementación SW internas, entre la que la más importante es el objeto *execution_context*, de clase *exec_context*. A continuación se muestra un extracto de la clase *exec_context*.

```
exec_context *execution_context;

class exec_context {
public:
   exec_context();
   void register_thread_process(UC_ENTRY_POINT fn,
                                uc_module* mod,
                                char *name,
                                unsigned int stack_size);
   ...
   void start_threads();
   void wait_threads();
private:
   ...
}

void exec_context::register_thread_process() {
   ...
   // add thread to system thread list
   // fill thread creation fields
   ...
}
```

El objeto *execution_context* se genera al llamar a la función de entrada de la aplicación SW embebida. La función de entrada depende del API de RTOS. Para el API POSIX, la librería *SWGen* implementa (en el paquete *sc2posix*) una función *main*, pero para el API C de *eCos*, la librería *SWGen* implementa (en el paquete *sc2eCos*) la función *cyg_user_start*. A continuación se muestra un extracto de la función de entrada (función *main*) en el paquete *sc2posix*.

```
// in uc_main.cpp
...
int main(int argc, int *argv[]) { // entry point
   ...
   execution_context = new exec_context;
   ...
   uc_main(argc,argv);
}
```

En la función de entrada, las dos acciones principales son la creación del objeto *execution_context* y el arranque de la función *sc_main* de la especificación (convertida en *uc_main* por la librería de generación en la fase de preprocesado).

El objeto *execution_context* es una facilidad de implementación software que se puede equiparar al objeto de implementación *sc_simcontext*, presente en las últimas versiones de implementación de la librería SystemC. El objeto *sc_simcontext* contiene casi la totalidad del simulador que es embebido en la aplicación ejecutable SystemC.

Almacena toda una serie de objetos, entre ellos las listas de procesos ejecutables en cada delta de simulación, así como otros objetos que, en definitiva, controlan la ejecución de nivel de sistema, esto es, la simulación SystemC. El objeto *execution_context* tiene una función similar, ya que controla y gestiona la ejecución de la especificación en un contexto de implementación software. A grandes rasgos, la estructura de implementación de esta clase es similar a la de la clase *sc_simcontext*. Cuenta con una lista de procesos en la que cada elemento guarda los datos de gestión del hilo, y un conjunto de funciones miembro (como *register_thread_process* o *start_threads*), que son llamadas desde las macros y funciones que implementan en software la declaración de procesos y su arranque (como UC_THREAD o *uc_start*).

No obstante, siguiendo la línea de la metodología *SWGen*, la clase *exec_context*, es más simple y ajustada a las necesidades de una implementación software que la clase *sc_simcontext*. En efecto, cada nivel tiene distintas necesidades. En el nivel de simulación SystemC, el control de la ejecución es más complejo. Por ejemplo, la ejecución de nivel de sistema tienen dos fases bien diferenciadas, elaboración y simulación, pero luego cada una se desglosa en más. En concreto, hay 3 fases de elaboración y 4 de simulación. Además, una de las fases de la simulación se desglosa a su vez en una repetición iterativa de 4 subfases (evaluación, actualización, notificación en delta y notificación temporal). Todo esto requiere que el objeto *sc_simcontext* presente un amplio conjunto de facilidades dedicadas o relacionadas a ese complejo control de la simulación. Ejemplos de esas facilidades son la declaraciones y uso de constantes que reflejan el estado de la simulación (SC_SIM_OK, SC_SIM_ERROR, etc), de tipos como *sc_enum_mode*, métodos como *init*, *clean, initialize, simulate, stop, end, reset*, etc, y de funciones relacionadas como *sc_set_stop_mode*, *sc_get_stop_mode*, *sc_cycle*, etc. Sin embargo, en el nivel de generación de SW, la clase *exec_context* se ciñe a prestar un servicio de concurrencia, manteniendo una lista de procesos y permitiendo la declaración, el arranque y sincronización de terminación de los mismos. La clase *exec_context*, no necesita implementar el cambio de contexto o la política de planificación, ya que esto es resuelto por debajo por el RTOS embebido.

La semántica temporal o de de ejecución SW no se corresponde exactamente con la semántica temporal de nivel de sistema. Una de las razones es, precisamente, que una semántica de ejecución en delta en implementación SW no tiene mucho sentido por ineficiente. Por otro lado, la implementación SW requiere una incorporación de detalle en términos de semántica de ejecución. En efecto, la implementación SW requiere determinar la política de planificación, que no se define ni se asume en el nivel de especificación. Finalmente, *SWGen* asume que la implementación SW de los procesos puede ejecutarse indefinidamente y no devolver el control de la ejecución. En cambio, la simulación SystemC retorna el control al sistema operativo de la plataforma de desarrollo cuando no hay más eventos planificados para un delta de simulación futuro.

Muchas de estas inequivalencias son asumibles y controlables en pos de conseguir la eficiencia del software embebido generado. En cualquier caso, como se verá en las siguientes secciones, la librería *SWGen* trata de mantener la semántica funcional de la especificación en la implementación software para las distintas APIs de RTOS soportadas. Para ello, ofrece una implementación de canales y procesos que trata de mantener el determinismo de la ejecución tras el proceso de generación. El

determinismo es entendido como la conservación del orden parcial entre ciertos eventos del sistema (fundamentalmente los referidos a accesos a canal).

Referente al soporte de concurrencia, la metodología *SWGen* contempla dos posibilidades. Los procesos SystemC (o procesos de nivel de sistema) pueden ser implementados como hilos de un único proceso (implementación monoproceso), o como hilos pertenecientes a distintos procesos (implementación multiproceso). En esta discusión es importante hacer una distinción de términos. Por un lado, en el nivel de sistema se habla únicamente de proceso. Esta denominación puede resultar confusa, ya que, de hecho, la implementación de la librería SystemC se hace a través de hilos que se ejecutan en la plataforma nativa[8]. En cualquier caso, se usa para modelar una tarea concurrente y en la comunidad SystemC se ha asumido y normalizado el término *proceso* para ello. Por otro lado, en la implementación software, hay que tener en cuenta la diferencia entre hilos y procesos. Una distinción fundamental es que el proceso es una tarea concurrente que no comparte el mapa de memoria (y por tanto, su ámbito de visibilidad de variables) con otra, en tanto que un hilo lo comparte con otros hilos que pertenezcan al mismo proceso.

Las implementaciones actualmente provistas por la librería *SWGen* son monoproceso. Este tipo de implementación es más eficiente que la implementación multiproceso en arquitecturas monoprocesador, ya que todos los hilos, que implementan los procesos del sistema comparten el mismo espacio de memoria y, los cambios de contexto entre hilos son menos pesados que los cambios de contexto entre procesos. Muchos RTOS embebidos soportan una gestión de la concurrencia únicamente a través de hilos, dados los requerimientos de reducida huella (tamaño de código generado), poca carga computacional requerida, y a que la mayoría de arquitecturas embebidas actuales son monoproceso. Por esta razón, este tipo de mapeo cubre una gran parte de las necesidades de implementación de SW embebido.

El soporte de mapeo de procesos SystemC a procesos software, en lugar de a hilos, habilitará la aplicación de *SWGen* a sistemas embebidos de arquitecturas más complejas. En concreto, se podrán soportar arquitecturas de redes en chip (NoC) y multiprocesadoras (MPSoCs). Por ejemplo, es previsible que un MPSoC en el que cada elemento procesador (PE) cuente con su propio subsistema de memoria requiera al menos un proceso por PE. No obstante, el desarrollo de *SWGen* para soporte del mapeo a procesos no ha sido objeto del presente trabajo.

En la Figura 3-22 se muestra un esquema del soporte de concurrencia basado en hilos. En una capa superior las macros y funciones estándar presentes en el código de la especificación realizan llamadas a las facilidades de implementación de *SWGen*, situadas en el paquete de código de tipo *sc2rtos*, correspondiente al API de RTOS correspondiente. A su vez, estas facilidades de implementación *SWGen*, utilizan los servicios del RTOS embebido a través de ese API.

[8] No en vano, la macro usada en especificación para especificar un proceso SystemC se denomina SC_THREAD (hilo SystemC).

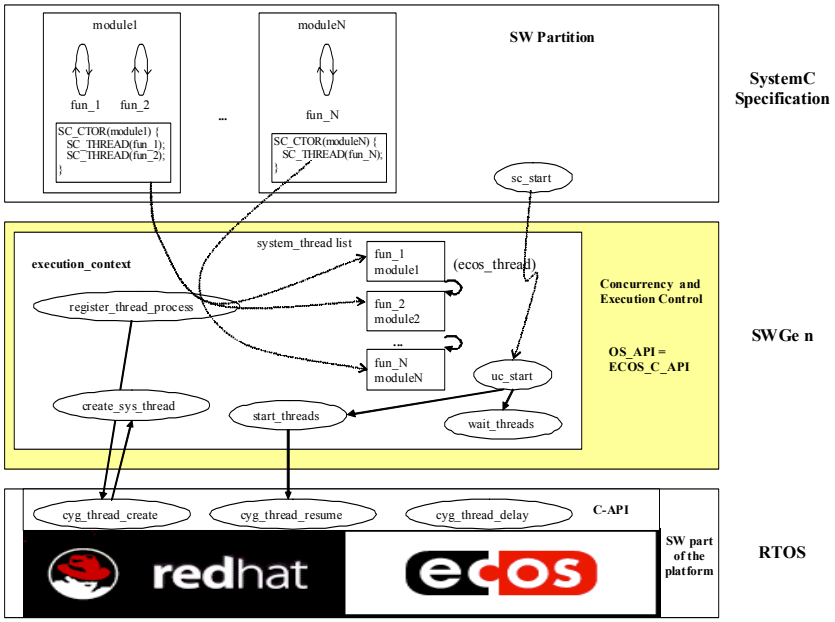



**Figura 3-22. Soporte de concurrencia y control de la ejecución de *SWGen* para el API C de *eCos*.**

Específicamente, en la Figura 3-22, la gestión de concurrencia se resuelve con llamadas a la API-C del sistema operativo *eCos*. La ejecución de la macro SC_THREAD (UC_THREAD tras el preprocesado) implica la ejecución de un método del objeto *execution_context* denominado *register_thread_process*. Este método añade a la lista de hilos del sistema mantenida por *SWGen* (*system_thread_list*) un nuevo objeto de clase *eCos_thread*. La clase *system_thread_list* contiene diversos campos necesarios para la posterior declaración de un hilo *eCos*. Entre los más importantes, contiene un puntero a función miembro. Este puntero apuntará a la función miembro del módulo declarada como proceso y pasada como parámetro en la macro SC_THREAD. Dicha función miembro contiene la secuencia de instrucciones que será ejecutada como un hilo *eCos*. La clase *eCos_thread* también contiene un puntero al módulo al que pertenece dicha función miembro. La ejecución del método *register_thread_process* también provoca la llamada a *cyg_thread_create*. Ésta es la llamada al sistema empleada en el RTOS *eCos* para declarar un hilo nuevo y pasarle toda la información que el RTOS *eCos* precisa para su lanzamiento y ejecución. Entre otra información, un hilo *eCos* precisa un manejador, el tamaño de pila asociado, un string para asociar un nombre al hilo, y parámetros si los hubiere. En el caso de *SWGen* caso, dado que los procesos SystemC no reciben parámetros, el método *register_thread_process* no invoca *cyg_thread_create* con parámetros ni, por tanto, reserva campos para ello en la clase *eCos_thread*.

En la Figura 3-23, se muestra la gestión que hace *SWGen*, si se mapea utilizando una API distinta, en este caso, POSIX. Una diferencia respecto a la Figura 3-22 es que los objetos de la lista de hilos del sistema son de clase *posix_thread*. Esta es una estructura que contiene un conjunto de campos similares, pero diferentes de los de la clase *eCos_thread*. De esta forma, estos campos se corresponden con los tipos necesarios para alimentar las llamadas de creación de hilos POSIX. También, a diferencia de la Figura 3-22, la llamada a *register_thread_process* no genera ninguna

llamada a función de declaración el hilo SW correspondiente al proceso. La razón es que en POSIX, la declaración y el arranque del hilo se realizan mediante la misma llamada (*pthread_create*). Esto es diferente en *eCos*, donde se requiere una llamada para declarar el hilo (*cyg_thread_create*) y otra para su arranque (*cyg_thread_resume*).

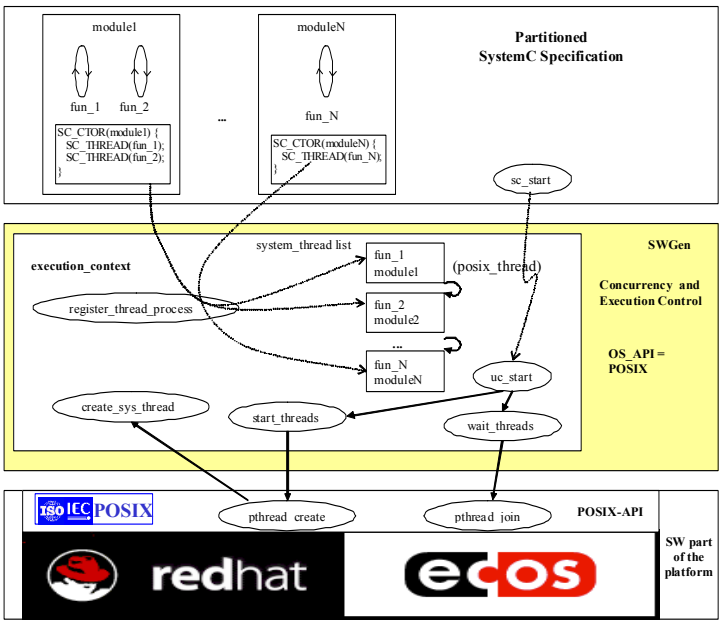



**Figura 3-23. Soporte de concurrencia y control de la ejecución de *SWGen* para el API POSIX.**

Por lo tanto, *SWGen* se encarga de reservar las estructuras de datos y realizar las gestiones adecuadas para cada API de RTOS, que como se ha mostrado, varían en cada caso. Por ejemplo, el paquete *sc2eCos* realiza dos llamadas al API-C de *eCos* para declarar y arrancar un hilo, en tanto que el paquete *sc2posix* realiza una llamada.

La Figura 3-24 muestra cómo la flexibilidad y estructura de *SWGen* provee múltiples posibilidades. *SWGen permite* implementar una especificación concurrente SystemC en *eCos* haciendo uso bien del API-C de *eCos*, bien del API POSIX de *eCos*.

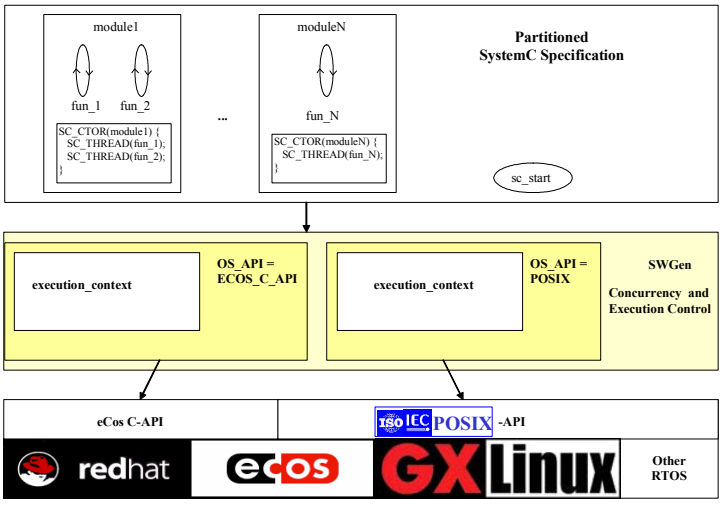



**Figura 3-24. Soporte de concurrencia con distintos APIs de RTOS para distintos RTOS embebidos.**

La Figura 3-24 también muestra las ventajas del soporte de la implementación SW usando APIs de RTOS estándar, tales como POSIX. Configurando el uso de la API POSIX, la librería *SWGen* puede generar código tanto para una plataforma con el RTOS embebido *GXLinux*, como para otra con el RTOS embebido *eCos*. En ambos casos se usa el mismo paquete de código de generación software, *sc2posix*.

En *SWGen*, por control de la ejecución se entiende la gestión del arranque de los procesos del sistema y su sincronización de terminación. Asúmase que la especificación de sistema tiene N procesos. En el paquete *sc2eCos* se arranca el sistema mediante N llamadas a la función *cyg_thread_resume*. En el paquete *sc2posix* se arranca mediante N llamadas a la función *pthread_create*. Como ya se ha visto, en este último caso, declaración y arranque son la misma cosa. Por esta razón, en el paquete *sc2posix*, la propia declaración del hilo está pospuesta al momento del arranque y, por tanto, al momento de la llamada a *sc_start* (convertido a *uc_start* en el preprocesado).

La sincronización de terminación en *SWGen*, consiste en que el contexto de ejecución fuerza una sincronización de todos los hilos de sistema antes del fin de la simulación. Esto es necesario para evitar que ningún hilo de sistema, incluido el propio hilo *sc_main* (*uc_main* tras el preprocesado), destruya sus recursos antes de que termine el resto. Esta sincronización no es necesaria para todas las APIs y/o RTOS. Por ejemplo, en *eCos* no es necesaria, en tanto que sí lo es en el paquete *sc2posix*. En este caso, el método *wait_threads* de la clase *execution_context* realiza N llamadas a *pthread_join* y asegura que los hilos POSIX de la implementación software se sincronicen con esa llamada (declarándolos de forma que sean *JOINABLE*).

A continuación se muestra un extracto de código (mezclado con pseudo-código) de la función *uc_start* y de los métodos *start_threads* y *wait_threads* del contexto de ejecución (*execution_context*) del paquete *sc2posix*:

```
// in uc_start.cpp
void uc_start(int) {
  execution_context->start_threads();
  execution_context->wait_threads();
}

// in exec_context.cpp
void exec_context::start_threads() {
   ...
   system_thread = first_system_thread;
   while(system_thread!=NULL) { // foreach registered system thread

      ...
      pthread_create(&system_thread->handler,
                     &system_thread->attr,
                     create_sys_thread,
                     (void *)&system_thread);
      ...
      system_thread = system_thread->next;
      ...
   }
   ...
}
```

```
void exec_context::wait_threads() {
   ...
   system_thread = first_system_thread;
   while(system_thread!=NULL) { // foreach registered system thread
      ...
      pthread_join(&system_thread->handler,
                   (void **)&thread_return_value);
      ...
      system_thread = system_thread->next;
      ...
   }
   ...

}
```

La implementación software incluye más elementos de sincronización entre hilos del sistema, pero estos están dedicados a la implementación de la semántica de comunicación de los canales SystemC y *HetSC*. Esto se trata en secciones posteriores.

### 3.5.7 Implementación de Especificación Temporal

La librería *SWGen* permite la implementación automática en SW de estamentos SystemC que portan información temporal estricta. Específicamente, la librería es capaz de implementar el estamento *wait* cuya declaración se expone a continuación:

```
void wait(double value, sc_time_unit tu);
```

Como se ve, la primitiva requiere un argumento de tipo *double* y otro del tipo *sc_time_unit*. Éste tiene la siguiente declaración en SystemC.

```
enum sc_time_unit {SC_FS=0,SC_PS,SC_NS,SC_US,SC_MS,SC_SEC};
```

En simulación, la llamada a este estamento provoca el bloqueo del proceso durante *value tu*. Es decir, si se ejecuta un estamento *wait(3,SC_SEC)* el proceso deja de ejecutarse durante 3 segundos de tiempo de simulación SystemC.

La librería *SWGen* provee una facilidad de implementación SW *wait* con la misma declaración, que reutiliza la declaración del tipo de unidades temporales (*sc_time_unit*) de SystemC. Para conseguir una implementación eficiente, el código de implementación SW transforma ligeramente la semántica inicial. En la implementación software el retardo significa el tiempo físico medido por el sistema objetivo que el hilo SW estará bloqueado hasta que pase a estar disponible para la ejecución. El RTOS realiza la gestión de los elementos temporizadores. De esa manera, el periodo de retardo podría ser mayor si no hay al menos un procesador libre para la ejecución y el hilo es menos prioritario que los que actualmente se ejecutan o la planificación es no expulsora. Esta transformación semántica es un refinado necesario que usualmente se daría en un flujo de traslación manual. Queda patente que la implementación SW necesita considerar parámetros que no son considerados en especificación, tales como el número de procesadores, política de planificación y prioridades.

Del mismo modo, como se mostrará posteriormente, la implementación SW tiene un rango de validez debido a las diferencias entre la plataforma de desarrollo y la plataforma objetivo. Nótese que estas limitaciones no son propias del esquema de generación de SW propuesto, sino de la plataforma de implementación subyacente.

La implementación de la sentencia *wait* en *SWGen* (en el paquete *sc2posix*) se realiza mediante llamadas a los servicios de temporización del RTOS. Por ejemplo, para el API POSIX, la librería *SWGen* implementa la función *wait* mediante una llamada a la función *nanosleep* de POSIX. La declaración de esta función es como sigue:

```
int nanosleep(const struct timespec *req, struct timespec *rem);
```

Esta función retarda la ejecución del programa durante al menos el tiempo especificado en **req*. La función puede regresar antes si el proceso recibe una señal mientras está bloqueado. En ese caso, devuelve -1, pone *errno* a EINTR, y escribe el tiempo restante en la estructura apuntada por *rem* a menos que *rem* sea NULL.

La función *wait* del paquete *sc2posix* adapta el par valor de tipo *double* y unidad de tiempo SystemC al tipo *timespec* requerido por la función *nanosleep*. El tipo *timespec* es un estructura declarada en <*time.h*> de la siguiente manera:

```
struct timespec {
  time_t   tv_sec;          /* segundos */
  long     tv_nsec;         /* nanosegundos */
};
```

La implementación SW dada por *SWGen* trata de minimizar la generación de señales para reducir las posibilidades de que la función *nanosleep* retorne antes de haber consumido todo el tiempo explicitado por la sentencia *wait* en la especificación. En cualquier caso, para ajustar la implementación frente a estas situaciones, la implementación *sc2posix* del *wait* repite la llamada a *nanosleep* hasta que se completa la espera planificada. Para ello, emplea el valor del parámetro **rem*. A continuación se muestra un extracto del código de implementación que refleja la previsión de ese caso:

```
...
while(nanosleep(&time_value_req,&time_value_rem)<0) {
    time_value_req.tv_sec = time_value_rem.tv_sec;
    time_value_req.tv_nsec = time_value_rem.tv_nsec;
#ifdef SWGEN_DBG
    printf("Retrial of nanosleep: new delay values: \n");
    printf("tv_sec = %d\n", time_value_req.tv_sec);
    printf("tv_nsec = %d\n", time_value_req.tv_nsec);
#endif
}
...
```

La implementación SW de esta primitiva tiene un rango de validez. Ese rango se define como aquel en el que el retardo temporal en términos de tiempo de simulación de la especificación se traduce en un retardo físico equivalente en la implementación. Existe un límite superior y otro inferior en el rango de validez de la implementación del retardo temporal. A efectos prácticos, normalmente ambas limitaciones las impone la plataforma objetivo. En la Figura 3-25 se representa el rango de validez de la implementación software del *wait* para valores típicos en plataformas nativa y objetivo.

Nótese que el rango de validez es mucho más pequeño en la plataforma objetivo, aunque en la representación de la Figura 3-25 se han aproximado las dimensiones de los rangos por claridad.

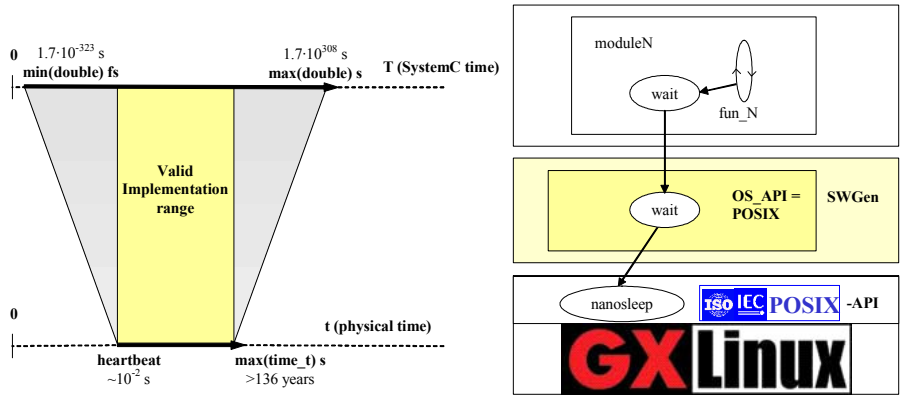



**Figura 3-25. Rango de validez de la implementación SW del retardo temporal.**

Por ejemplo, una plataforma objetivo de 32 bits que asigne 32 bits a *time_t*, podrá manejar un retardo de $2^{32}$ = 4Mseg (algo más de 136 años). Este límite superior será usualmente menor que el rango que puede ofrecer la simulación SystemC en plataforma de desarrollo, que utilizará un valor *double* con unidad de tiempo en segundos (*sc_time_unit* = *SC_SEC*). En cualquier caso, es razonable asumir que el límite superior no afectará a la mayoría de aplicaciones.

En cuanto al límite inferior, la plataforma objetivo es también normalmente el factor limitante. Ese límite lo impone el latido del sistema, es decir, la interrupción del reloj del sistema que controla todas las facilidades de temporización (contadores, alarmas, llamadas de espera, etc) del RTOS embebido. Un valor típico suele ser 10ms, aunque éste se va reduciendo conforme las frecuencias de reloj de los procesadores embebidos crecen. Es previsible que esta reducción se haga cada vez menos importante. Hoy día, en el mercado de PCs, la carrera en frecuencias ha frenado y dado lugar a otra de arquitecturas, siendo muy posible que este efecto extrapole a los sistemas embebidos. En cualquier caso, la reducción en el periodo del latido del sistema no podrá ser tan importante como para equipararse al mínimo retardo implementable en SystemC, es decir, *min(double) fs*. Por ejemplo, en una plataforma nativa que asigne 8 bytes al double, el mínimo retardo representable por SystemC es de $1.7 \cdot 10^{-308}$ fs. Para retardos muy pequeños el código de implementación del *wait* de sc2posix en *SWGen* necesita realizar la conversión del par (*double*, *sc_time_unit*) al par (*time_t*,*long*). Este código de conversión se ha optimizado, de modo que solo precisa un producto, una resta y un par de conversiones explícitas.

En definitiva, la metodología *SWGen* asume que el latido del sistema es la mínima resolución de tiempo estricto manejable en software. Una implementación de la primitiva *wait* más precisa para el límite inferior es posible, pero requeriría un uso directo del o los temporizadores de la plataforma y los servicios de interrupciones. Requiere como mínimo un temporizador por procesador en el sistema y, deseablemente, por proceso. La principal desventaja de esta implementación es que sería directamente dependiente de la arquitectura HW de la plataforma. Además, el latido del sistema seguirá probablemente presente para la implementación de otros servicios, con lo cual,

muchas de las sincronizaciones del proceso retardado con el resto de procesos software se harían efectivas en función del latido, lo que, en general, puede desvirtuar la precisión obtenida con una implementación de los retardos que use los temporizadores directamente. Por lo tanto, *SWGen* opta por una implementación que sólo depende del API de RTOS, lo que aumenta la portabilidad objetivo de la librería de generación.

### 3.5.8 Implementación de Canales

En el Capítulo 2 se ha visto que los canales son elementos importantes de la metodología de especificación heterogénea *HetSC*. Estos contienen gran parte de la semántica del modelo de computación empleado. En la metodología *HetSC*, los canales vienen dados, pero el usuario ha de conocer su sintaxis y su semántica.

La metodología *SWGen* se encarga de proveer una implementación SW de estos canales cuando corresponde. La Figura 3-21 de la sección 3.5.5 y la Figura 3-18 de la sección 3.5.2 distinguen los distintos tipos de implementación SW que es preciso generar, dependiendo de la partición HW/SW. A estas implementaciones se las denomina canales de implementación software o canales *SWGen*. Hay tres tipos de canales *SWGen*: canales SW/SW, canales HW/SW y canales de Entrada/Salida (o I/O).

Los canales *SWGen* implementan interfaces *SWGen*. Estas interfaces son prácticamente idénticas a las interfaces SystemC implementadas por el canal de sistema (SystemC o *HetSC*) correspondiente. Una diferencia es que las interfaces *SWGen* no heredan la clase *sc_interface*. Asimismo, los canales *SWGen* heredan interfaces *SWGen*, pero no la clase *sc_channel* o *sc_prim_channel* de SystemC. Es decir, se elimina la herencia y uso de clases SystemC innecesarias en la implementación software.

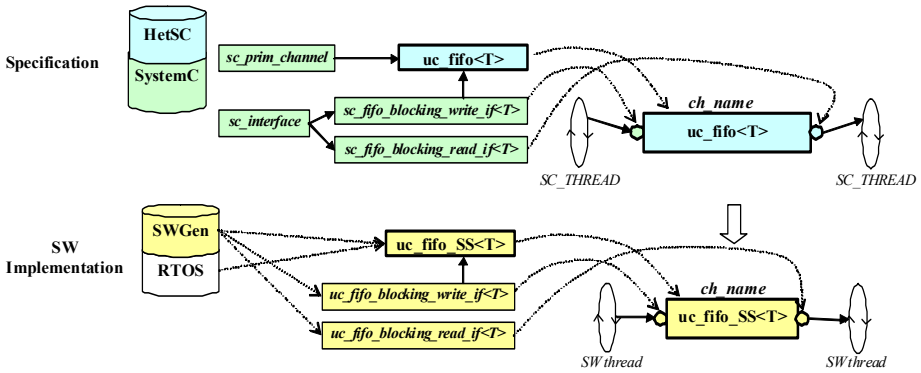



**Figura 3-26. El canal SystemC se implementa en SW mediante un canal de implementación.**

La implementación de la interfaces y canales *SWGen* se realiza mediante facilidades C/C++ y llamadas a la librería de RTOS embebido. La implementación trata de proveer una semántica equivalente (ver sección 3.6), que respete los supuestos y propiedades del MoC. No obstante, también refina la especificación de sistema, es decir, toma decisiones que hacen factible, sistemática y eficiente la implementación en SW.

En la Figura 3-20 se muestra como se sustituye automáticamente una declaración de canal de tipo *sc_fifo* por un canal *SWGen* denominado *sc_fifo*_SS. Esta sustitución es la que se refleja en la Figura 3-26 (aunque para un canal HetSC similar, el *uc_fifo*).

El código de especificación en el que se declara e instancia el canal *uc_fifo* sería el siguiente:

```
#include "general.h"
#include "sw_section.h"

// i.e. declaration in the body of a module to be implemented in SW…
uc_fifo<int>   *ch_name;

// … instance in the constructor of the module to be implemented in SW
ch_name = new uc_fifo<int>("ch_name",10);
```

En el nivel de especificación, la cabecera *"general.h"* incluye las librerías SystemC y *HetSC* y se utiliza el canal *uc_fifo* de *HetSC*. En el nivel de especificación, la cabecera *"sw_section.h"* no tiene impacto alguno. Sin embargo, en el nivel de generación de SW, el preprocesado de ese extracto de especificación, suponiendo una implementación SW/SW del canal (canal SW/SW) da lugar a un código del siguiente estilo:

```
// unfold of "general.h" without substituting the #include clauses
#include "Swgen.h"
#include "rtoslib.h"

// unfold of the #include "sw_section.h" statement substituting
// the clause #define uc_fifo uc_fifo_SS

// i.e. declaration in the body of a module to be implemented in SW…
sc_fifo_SS<int>   *ch_name;

// … instance in the constructor of the module to be implemented in SW
ch_name = new sc_fifo_SS<int>("ch_name",10);
```

### *3.5.8.1 Canales SW/SW*

Cuando un canal comunica dos o más procesos y todos ellos se hayan dentro de la partición software, entonces, la implementación de ese canal es íntegramente realizada en software. En la metodología de generación *SWGen*, el canal de sistema (provisto por la librería SystemC o la *HetSC*) se implementa entonces como un canal SW/SW.

Teniendo en cuenta la posibilidad de mapeo de procesos SystemC a hilos o a procesos (sección 3.5.6), la metodología de generación contempla los siguientes tipos de canales SW/SW:

- Tipo 1: Los que implementan una comunicación entre hilos.
- Tipo 2: Los que implementan una comunicación entre procesos. En este caso hay dos posibilidades de comunicación:
  - Local: los procesos involucrados no requieren infraestructura de red.
  - Remota: los procesos involucrados requieren infraestructura de red.

  En la Figura 3-27 se presentan estas posibilidades:

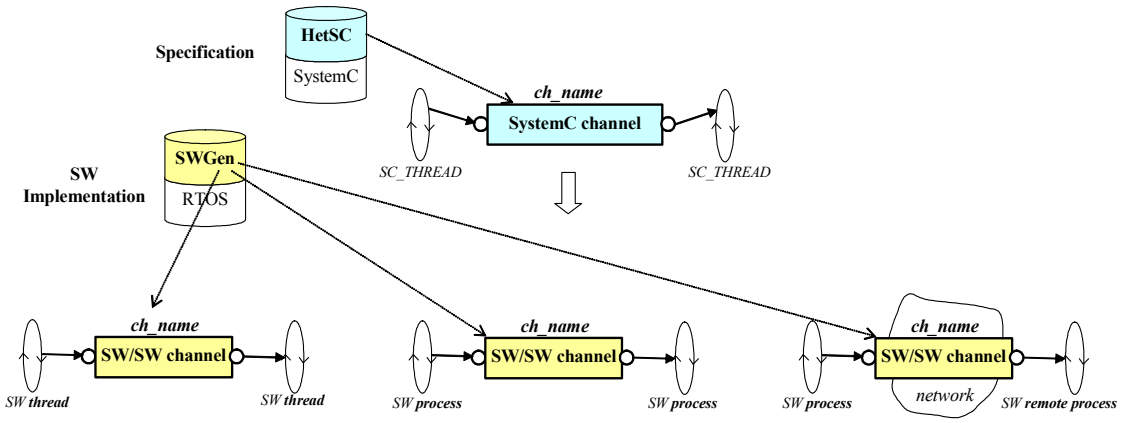



**Figura 3-27. Posibilidades de implementación de canales SW/SW.**

El segundo tipo de canales SW/SW debe emplear IPCs (*Inter Process Calls*), por ejemplo, con llamadas tipo *socket*. Este caso incluye la posibilidad de que los procesos deban comunicarse mediante una red. En tal caso, la implementación de los canales SW/SW utilizará llamadas a servicios de red como, por ejemplo, llamadas tipo *socket IP* si la red es IP y el RTOS embebido soporta una pila IP. Dado que, como se comentó en la sección 3.5.6, el mapeo a procesos SW no es objeto de este trabajo, tampoco se trata aquí la implementación de canales con llamadas IPC. Por ello, en lo que sigue, se hará referencia al primer tipo de canales SW/SW.

En este trabajo se ha resuelto el primer tipo de canales SW/SW. Para entender la implementación SW/SW del canal, es conveniente entender las semejanzas y diferencias existentes con respecto a la implementación SystemC del canal de nivel de sistema. En la Figura 3-28 se representan el canal de nivel de sistema *uc_fifo* (parte superior) y el correspondiente canal de implementación SW/SW *sc_fifo_SS* (parte inferior) para comunicación entre hilos (Tipo 1).

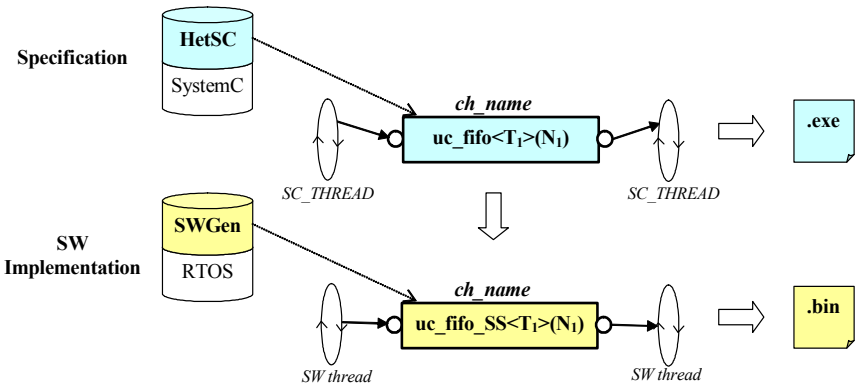



**Figura 3-28. Ejemplo de canal SW/SW.**

Adicionalmente, la Figura 3-29 permite comparar las declaraciones del canal de sistema y el canal de implementación. Como se puede comprobar, las declaraciones son muy parecidas. Las interfaces (*sc_fifo_blocking_in_if* y *sc_fifo_blocking_out_if* son idénticas en nombre y en los métodos declarados. Sin embargo, en la librería *SWGen*, no heredan la clase *sc_interface*.

```
template <class T>
inline void uc_fifo<T>: public sc_fifo_blocking_in_if<T>,
                        public sc_fifo_blocking_out_if<T> {
public:
    // constructors
    explicit uc_fifo( int size_ = 16);
    ...
    // blocking read
    ...
    // blocking write
    void write( const T& );
    ...
private:
    ...
};
```

```
template <class T>
class uc_fifo_SS<T>: public sc_fifo_blocking_in_if<T>,
                     public sc_fifo_blocking_out_if<T> {
public:
    // constructors
    explicit sc_fifo_SS( int size_ = 16);
    ...
    // blocking read
    ...
    // blocking write
    void write( const T& );
    ...
private:
    ...
};
```

**Figura 3-29. Extracto de declaraciones del canal uc_fifo y de su clase de implementación software.**

La implementación SystemC difiere claramente de la implementación del canal SW/SW. En primer lugar, como se mostró en la Figura 3-26, el canal de implementación SW/SW no hereda *sc_channel* o *sc_prim_channel*, ni hace uso de los servicios de SystemC. La Figura 3-30 muestra la implementación de nivel de sistema del canal estándar *sc_mutex* (parte izquierda), así como su implementación de canal SW/SW en los paquetes sc2ecos y sc2posix (parte derecha).

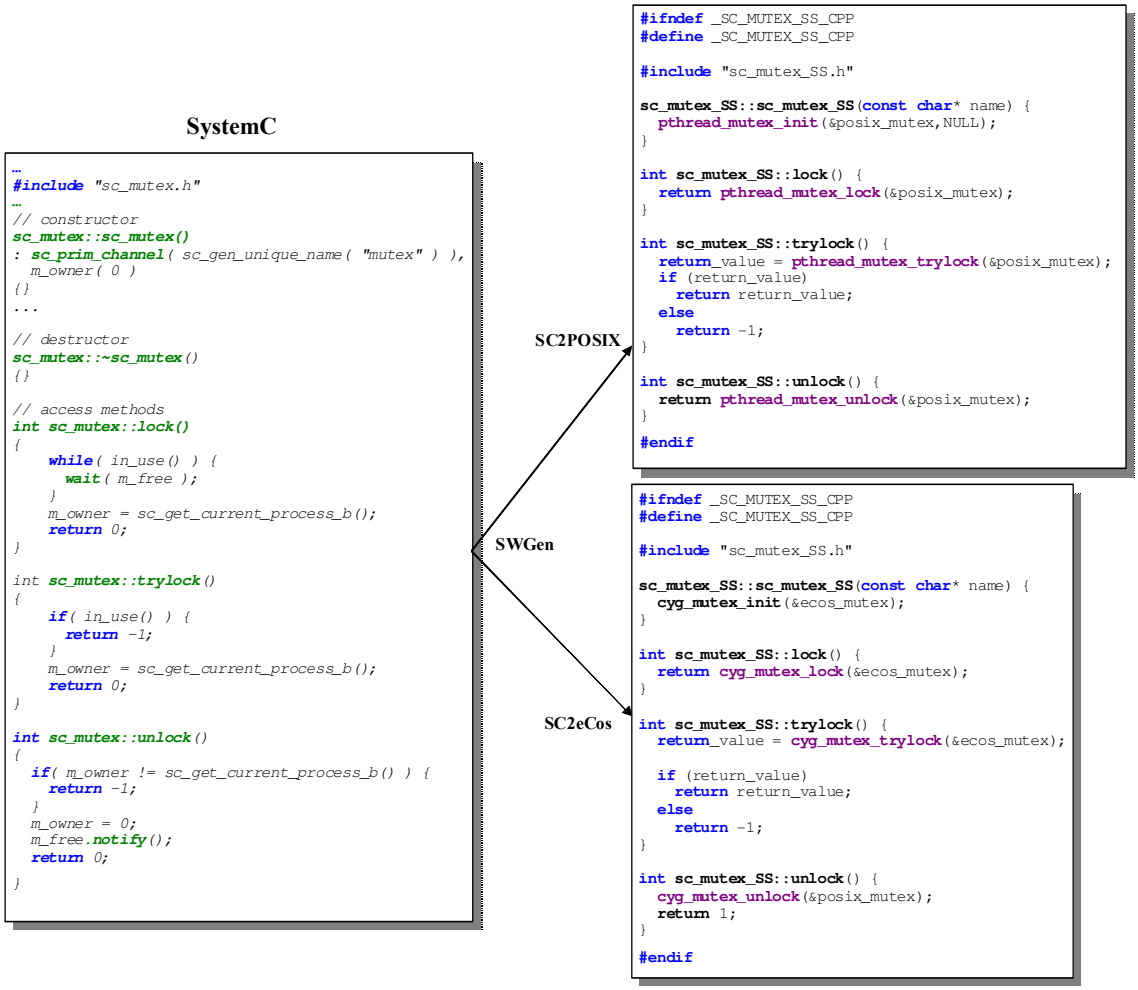



**Figura 3-30. Implementación SystemC y SW/SW del canal sc_mutex.**

Este es un caso sencillo en el que se implementa prácticamente toda la interfaz que aparece en el nivel de sistema, excepto el destructor. Obsérvese el uso de las primitivas SystemC *notify* y *wait* en la implementación de sistema, en tanto que las implementaciones SW utilizan las APIs POSIX (con llamadas al sistema con prefijo *pthread_*) y eCos (con llamadas al sistema con prefijo *cyg_*).

La Figura 3-30 también ilustra cómo la adaptación de la librería *SWGen* para nuevas APIs de RTOS es casi sistemática en algunos casos. Por ejemplo, la adaptación para el canal mutex SystemC suele ser sencilla, ya que la mayoría de RTOS cuentan con una primitiva de sincronización de ese tipo y con una interfaz similar. En estos casos, la librería *SWGen* ofrece un envoltorio que automatiza la generación desde el API SystemC y adapta los tipos de parámetros de entrada y de retorno. Por ejemplo, la llamada *unlock* del mutex eCos no retorna ningún valor. Por ello, la implementación SW/SW de *unlock* tiene que incluir al final la sentencia "*return* 1;" (Figura 3-30).

En otros casos la adaptación es menos inmediata, por ejemplo, porque, el RTOS embebido no provee primitivas similares a las de sistema. Entonces, la implementación de los métodos de acceso se complica. En la Figura 3-31 se muestran extractos del código de implementación SystemC (izquierda) e implementación SW/SW POSIX (derecha) del método *write* del canal *HetSC uc_fifo*.

**SystemC (HetSC)**

```
template <class T>
inline void uc_fifo<T>::write( const T& val_ ) {
   // dynamic rule checking
#ifdef DYNAMIC_CHECKING_UC_FIFO_CHANNEL
   check_write();
#endif
   while( (m_size - m_num_readable -
m_num_written)==0 ) {
     wait( m_data_read_event);
   }
   m_num_written ++;
   buf_write( val_ );
   request_update();
}
```

```
template <class T>
inline void uc_fifo<T>::update() {
   if( m_num_read > 0 ) {
      m_data_read_event.notify(SC_ZERO_TIME);
   }
   if( m_num_written > 0 ) {
      m_data_written_event.notify(SC_ZERO_TIME);
   }
   m_num_readable = m_size - m_free;
   m_num_read = 0;
   m_num_written = 0;
}
```

**SWGen (SC2POSIX)**

```
template <class T>
inline void uc_fifo_SS<T>::write( const T& val_ ) {

#ifdef _WITH_LOCK
  pthread_mutex_lock(&writer_mutex);
#endif

  sem_wait(&room_to_write);

  // dump of the data
  buffer[write_index] = val_;

  // buffer write index update
  if(write_index>=(size-1))
    write_index=0;
  else
    write_index++;

  sem_post (&data_to_read);

#ifdef _WITH_LOCK
  pthread_mutex_unlock(&writer_mutex);
#endif

}
```

**Figura 3-31. Extractos de código del canal uc_fifo y su canal de implementación SW/SW.**

En este caso, aunque técnicamente se implementan los mismos métodos que en el nivel de sistema, la implementación software deja vacíos algunos de ellos. En el nivel de especificación, el canal declara una serie de métodos y funciones que, si bien son de utilidad en dicho nivel, en el de generación SW, donde interesa minimizar el tamaño del código y generar sólo el código funcional, son innecesarios y, por tanto, no se implementan. Por ejemplo, el canal *HetSC uc_fifo* declara e implementa el método *trace()*. Este método realiza un volcado de los datos del canal a un fichero. Esto sirve para depurar y comprobar los resultados. En cambio, la implementación *SWGen* de ese método se deja vacía.

Se implementan al menos (de forma parecida al caso de los puertos) los métodos de acceso, que sirven para comunicar y sincronizar los hilos, que implementan en software los procesos de sistema. Por ejemplo, el canal de implementación *uc_fifo_SS* implementa los métodos de acceso bloqueantes *read* y *write*.

Otro ejemplo de eliminación de código que no se implementa en SW tiene que ver con la realización de chequeos en el nivel de especificación. Por ejemplo, en la implementación del canal *uc_fifo* se declara e implementa el método *register_port()*. Este método implementa un chequeo estático que da soporte al MoC BKPN realizado durante la conexión del canal al puerto, en tiempo de elaboración, para evitar que se conecten varios lectores y/o escritores a un canal *uc_fifo*.

Análogamente, en la implementación de nivel de sistema, especialmente en *HetSC*, existe código de chequeo dinámico. En la parte superior izquierda de la Figura 3-31 se muestra un extracto del método *write* del canal *uc_fifo*. Como se puede ver, ese método incluye un método de chequeo, *check_write* (utilizado para chequear el cumplimiento de las condiciones del MoC BKPN), el cual, a su vez, tiene asociado toda una infraestructura de código que no se presenta en la figura por cuestión de espacio. En cambio, la implementación del método *write* del canal SW/SW *uc_fifo_SS,* presentada en la parte derecha de la Figura 3-31, no incluye dicho método de chequeo. La implementación software gana así en eficiencia en tamaño, velocidad y consumo. La implementación sigue ateniéndose a las condiciones del MoC, puesto que el procedimiento de generación de software no genera nuevas conexiones del canal a otros procesos. Por lo tanto, mantener el chequeo hubiera sido innecesario e ineficiente. El flujo de diseño arranca en la fase de especificación y en esa fase ya se ha asegurado que no hay varios lectores ni escritores. En la sección 3.6 se trata en más detalle cómo la metodología de generación preserva las propiedades del MoC.

Como en la Figura 3-30, otra diferencia patente entre la implementación de sistema y la implementación SW proviene del tipo de infraestructura de ejecución. En tanto que la implementación SystemC del canal *uc_fifo* se apoya en el kernel de simulación de eventos discretos, la implementación SW se apoya en un RTOS de tiempo real. El código SystemC del canal *uc_fifo* utiliza esperas y notificaciones a eventos SystemC (parte izquierda de la Figura 3-31) para implementar la semántica de sincronización del canal. La Figura 3-31 muestra también que la implementación SystemC del canal *uc_fifo* emplea el método *update()*, reflejando el uso del modelo de ejecución temporal, basado en ciclos delta y en el mecanismo de evaluación y actualización en el que éstos se dividen. Como se mencionó, este modelo es ineficiente en implementación software. La implementación SW utiliza otros elementos: las llamadas de sincronización entre hilos del RTOS para el API correspondiente, acorde con el modelo temporal empleado en la implementación (ver sección 3.6).

Aunque en casos como el del canal *uc_fifo* se complica un poco la implementación SW, es posible encontrar cierta estructura en el código de implementación y, por tanto, que se pueda sistematizar la misma a la hora de portar *SWGen* a otras APIs de RTOS. En la Figura 3-32 se contrastan la implementación del método *write* del canal *HetSC uc_fifo* en los paquetes sc2ecos y sc2posix. Está claro que las dos implementaciones utilizan APIs diferentes y primitivas de sincronización diferentes. En el primer caso, la implementación se resuelve mediante tres tipos de

primitivas de sincronización de hilos eCos siguientes: un mutex, una bandera y un semáforo. En el segundo caso, se usan dos tipos de primitivas de sincronización de hilos POSIX: dos mutexes y dos semáforos.

**SC2eCos**

```
template <class T>
inline void sc_fifo_SS<T>::write(const T& data) {

#ifdef _WITH_LOCK
  cyg_mutex_lock(&writer_mutex);
#endif

  cyg_semaphore_wait(&channel_sem);

  // dump of the data
  buffer[write_index] = data;

  // buffer write index update
  if(write_index>=(size-1))
    write_index=0;
  else
    write_index++;

  cyg_flag_setbits(&channel_flag,THERE_IS_DATA);

#ifdef _WITH_LOCK
  cyg_mutex_unlock(&writer_mutex);
#endif

}
```

**SC2POSIX**

```
template <class T>
inline void uc_fifo_SS<T>::write( const T& val_ ) {

#ifdef _WITH_LOCK
  pthread_mutex_lock(&writer_mutex);
#endif

  sem_wait(&room_to_write);

  // dump of the data
  buffer[write_index] = val_;

  // buffer write index update
  if(write_index>=(size-1))
    write_index=0;
  else
    write_index++;

  sem_post (&data_to_read);

#ifdef _WITH_LOCK
  pthread_mutex_unlock(&writer_mutex);
#endif

}
```

**Figura 3-32. Implementación SW/SW sc2ecos y sc2posix del método write del canal uc_fifo.**

Pese a esas diferencias, se puede apreciar una estructura repetitiva en la implementación. En primer lugar se bloquea el acceso a la llamada para que ningún otro hilo pueda acceder a ella en tanto no se haya completado, es decir, se convierte la llamada *write* en una llamada atómica. Esto se solventa en ambos casos mediante un *mutex*, que es una primitiva de sincronización sencilla que suele proveer cualquier API. Posteriormente se comprueba si la memoria intermedia del canal está llena. De ser así, la llamada ha de bloquear el hilo llamador. En ambos casos se resuelve mediante un semáforo. El valor de inicialización del semáforo es el del tamaño de la memoria intermedia del canal, en tanto que la variable de cuenta interna asociada al semáforo es la que refleja el número de unidades de datos presentes en la memoria intermedia interna. Esa inicialización se realiza en el constructor de la clase de implementación. Una vez que se comprueba que hay espacio en el canal, se añade el dato a la memoria intermedia interna y se actualiza el índice de escritura, teniendo en cuenta que se trata de una memoria circular. Después, se notifica la presencia de datos en el canal, por si hubiera un hilo lector bloqueado a la espera. Esto se resuelve en SC2eCos mediante una notificación de una bandera, en tanto que en SC2POSIX se notifica a un semáforo. Finalmente, al haber completado las operaciones necesarias, se desbloquea el mutex de acceso. En la Figura 3-44 se ilustra otra posible implementación POSIX en la que se pueden apreciar algunas diferencias de implementación, sutiles en cuanto a restricciones, y más explícitas en cuanto a código involucrado y primitivas empleadas.

### *3.5.8.2 Canales HW/SW*

Cuando un canal de la especificación comunica dos procesos y uno se haya dentro de la partición software en tanto que el otro se haya dentro de la partición hardware, entonces, su implementación comprende tanto una parte software como otra hardware. En lo que respecta a la librería *SWGen*, a la parte de software específica de la

implementación de canal se la denomina canal HW/SW o controlador software de canal. La implementación de canales HW/SW requiere cierta visibilidad y conocimiento de la plataforma, fundamentalmente, del mapa de memoria. Con respecto a la hardware de la implementación de canales HW/SW, en este documento se introduce la solución propuesta y explicada en detalle en **[FHSV02]**.

Un canal de especificación puede requerir más de un canal HW/SW. En función de la partición (HW o SW) en la que se emplacen cada una de las interfaces de la instancia de canal, uno u otro canal HW/SW será el adecuado para la implementación SW. En la Figura 3-33 se representan dos posibilidades de implementación HW/SW, es decir, dos canales HW/SW, correspondientes a un canal *uc_fifo*. En función de la dirección de transferencia, se utiliza un canal de implementación HW/SW u otro: *uc_fifo_H2S*, cuando el canal transfiere datos desde hardware a software y un canal *uc_fifo_S2H* si el sentido de transferencia de datos va de software a hardware.

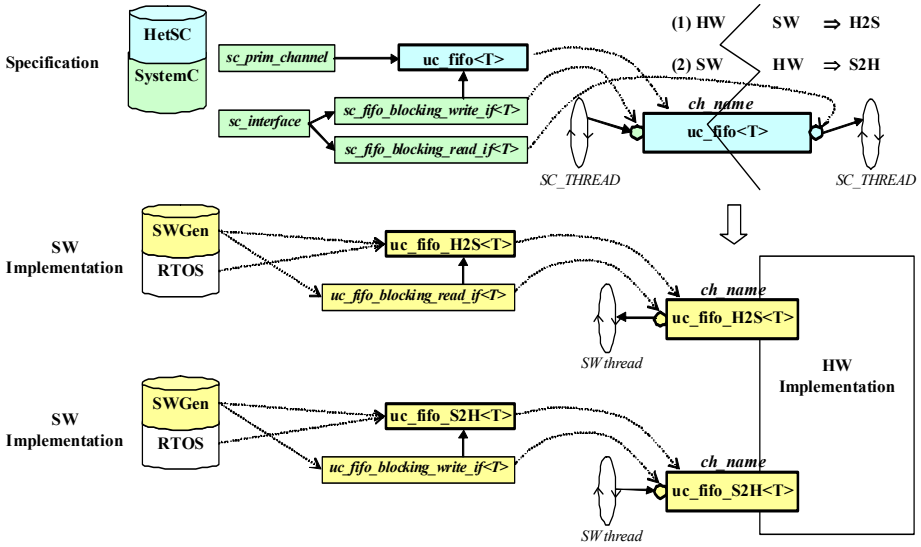



**Figura 3-33. Canales de implementación HW/SW para el canal uc_fifo.**

En cualquier caso, tal y como se representa en la Figura 3-33, el canal HW/SW es y se comporta como un controlador software, que es invocado desde el código de un hilo software y da acceso al hardware que implementa los procesos de la partición hardware a través del bus del sistema.

Entre el hilo software y el proceso hardware existen una serie de elementos tanto software como hardware, que complementan al canal HW/SW y que permiten sistematizar la implementación. En la Figura 3-34, estos elementos están representados como hardware/software de comunicación de propósito general. La parte SW está particularizada para un RTOS eCos y un bus AMBA-APB. Esta solución se basa en una estructura que separa las dos funciones principales de un canal (inalteradas, bien se trate de su implementación HW/SW, de su implementación SW/SW o de la implementación SystemC): la transferencia de datos y la sincronización entre procesos.

Para la transferencia de datos, en la Figura 3-34, se representan unos bloques HW (HW_SW_DATA y SW_HW_DATA) que contienen unos registros que implementan la memoria intermedia interna del canal y que están mapeados en memoria para permitir

su acceso desde el canal HW/SW. Esta es una solución buena para canales con memoria intermedia interna pequeña. De hecho, la solución de la Figura 3-34 se utilizó para probar la implementación del canal *uc_simple_channel* **[HSV02]**, básicamente el canal *sc_fifo* con semántica de acceso bloqueante y tamaño 1. Para canales que necesiten una memoria intermedia interna de longitud apreciable se recurrirá a memorias, también mapeadas directamente en memoria.

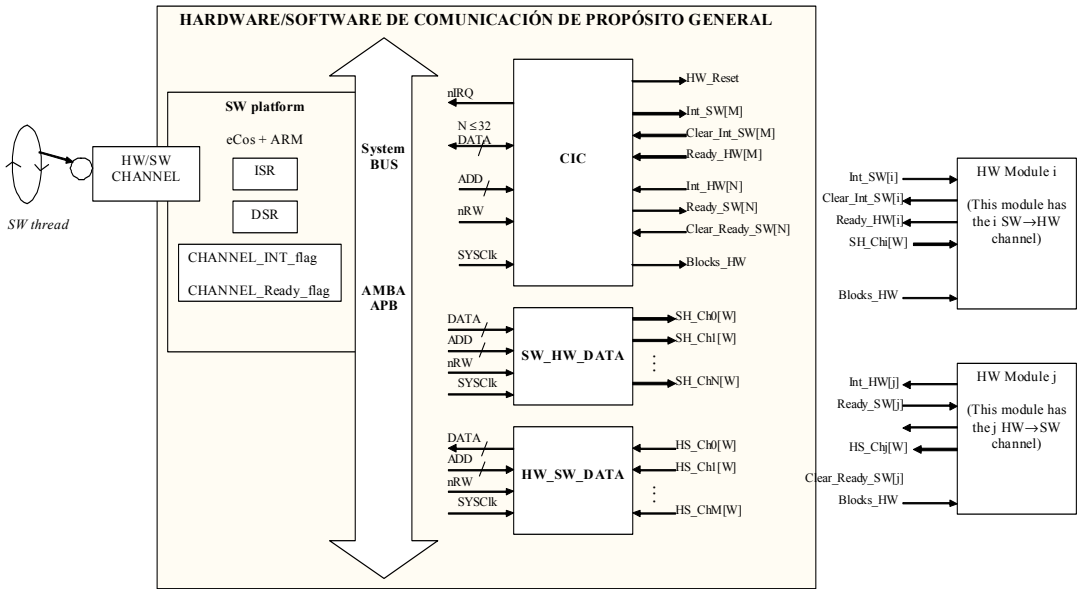



**Figura 3-34. Software y hardware de interfaz para la implementación de canales HW/SW**

La otra cuestión importante en los canales HW/SW es cómo se resuelve la sincronización entre los hilos SW y los procesos HW. Exista transferencia de datos o no, y en su caso, sea la transferencia unidireccional o bidireccional, existe una necesidad de sincronización de doble sentido. Es decir, hay que habilitar la implementación de:

1. que un proceso hardware pueda bloquearse y que un hilo software pueda desbloquearlo. Eso se denomina *notificación SW→HW*.
2. que un hilo software pueda bloquearse y ser despertado por un proceso hardware. Esto se denomina *notificación HW→SW*.

En la arquitectura propuesta, existe un módulo que se encarga de toda la gestión intermedia entre canal HW/SW (que es software) y los procesos HW para la resolución de las necesidades de sincronización: el controlador de interrupciones de canal o CIC, representado en la Figura 3-35. El CIC contiene un juego básico de 3 registros mapeados en memoria para las notificaciones de software hacia hardware (SW→HW) y de hardware hacia software (HW→SW). Cada registro de la Figura 3-35 está formado por una serie de banderas. El registro superior contiene las banderas (*flags*) de notificación SW→HW, usadas para los dos sentidos de transferencia de datos. La mitad son de tipo *Int_SW* y se reservan para la implementación de canales con transferencia de datos de SW a HW (S2H). Con esas banderas el SW puede notificar al HW que va a enviar o que ha enviado datos. La otra mitad de banderas son de tipo *Ready_SW* y se reservan para implementar canales con transferencia de datos desde HW a SW (S2H). Con ellas, el SW puede notificar al HW que está listo para recibir más datos o que ya

los ha recibido correctamente. El significado específico de las banderas lo completa el canal HW/SW.

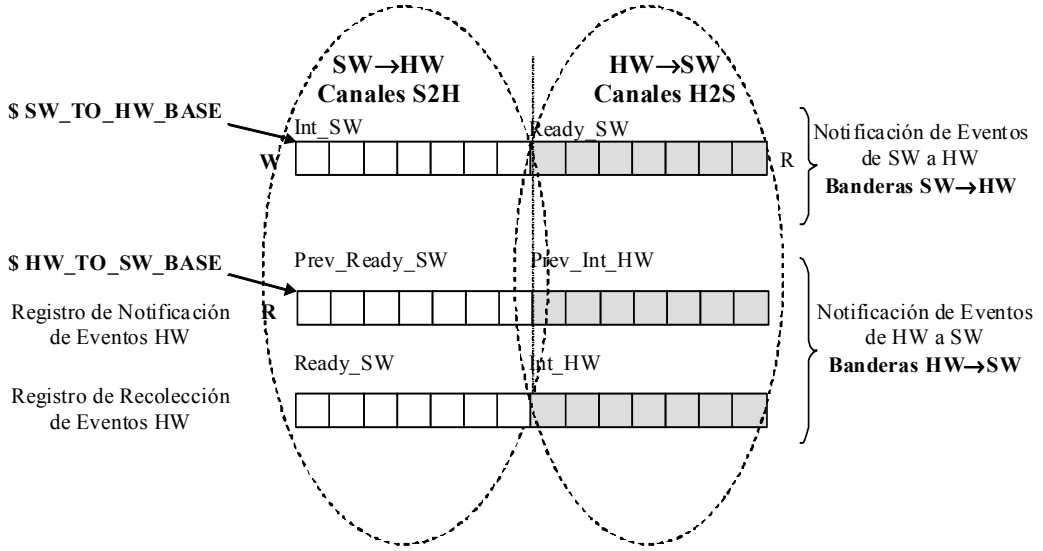



**Figura 3-35. Juego de registros para la notificación de eventos del CIC.**

La Figura 3-36 esquematiza la implementación HW de la bandera de notificación SW→HW.

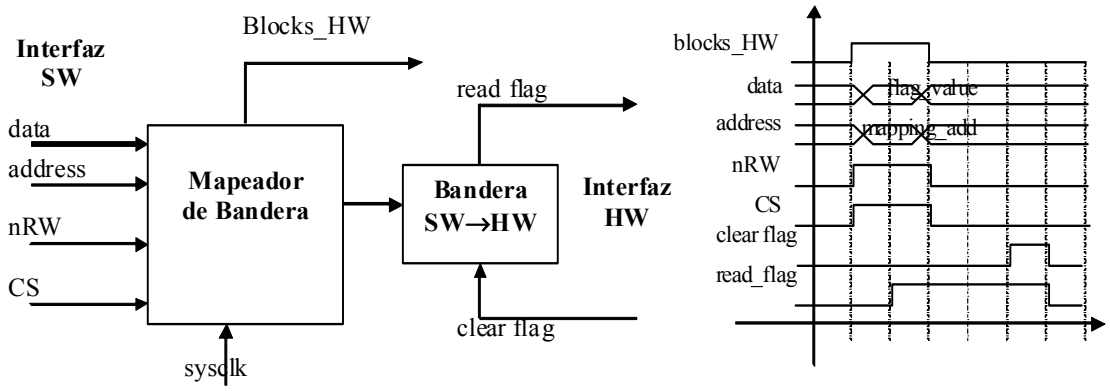



**Figura 3-36. Esquema básico de la bandera SW→HW.**

En la notificación SW→HW, el proceso hardware se "bloquea" encuestando indefinidamente una línea (*read_flag)* que se actualiza cada ciclo, reflejando el valor de la bandera correspondiente del registro de notificación SW→HW. De este modo, el proceso SW realiza una notificación simplemente escribiendo en la dirección de memoria del registro de notificación SW→HW y específicamente, a través de una máscara, en la bandera correspondiente. Entonces, la línea *read_flag* se habilita y el proceso hardware se "desbloquea" y continúa la ejecución hasta el siguiente "bloqueo" (que en realidad es una encuesta). En este caso, la encuesta del proceso hardware no implica el bloqueo de otros procesos hardware, ya que cada proceso hardware cuenta con sus propios recursos de ejecución.

Para la implementación de las banderas de notificación HW→SW, una posibilidad inmediata es que exista un hilo software adicional de alta prioridad o bien

una excepción temporizada que se dedique a leer los registros de notificación HW→SW, mapeados en memoria, y que representen que el hardware requiere notificar y/o despertar un hilo software.

Existe otra posibilidad, que se ha explorado en el ámbito de esta tesis. Esta solución se basa en utilizar una interrupción hardware, para minimizar el retardo entre la notificación hardware y el que esta se haga efectiva en el software. De este modo, la implementación de las banderas de notificación HW→SW se realiza mediante dos registros de eventos, un registro de recolección de eventos y otro de notificación de eventos (etiquetado con el prefijo *Prev_* en la Figura 3-35). Sólo el registro de notificación de eventos es mapeado en memoria, para su lectura desde el software. El CIC aserta una línea de interrupción *nIRQ_CIC*, y por tanto provoca una interrupción, siempre que dentro de un intervalo temporal mínimo haya aparecido algún evento de notificación desde HW hacia el SW (notificación HW→SW). A este intervalo se lo denomina intervalo mínimo de recolección de notificaciones o $T_{CIC}$.

Durante el intervalo de recolección, se van registrando en el registro de recolección todos los eventos que acaezcan. Una vez el intervalo expira, si ha habido al menos una notificación en el registro de recolección de eventos, la señal IRQ se eleva. Si no, el intervalo de recolección se alarga hasta la llegada de la primera notificación. Por tanto, los intervalos de recolección no tienen porqué ser periódicos. Esto queda reflejado en la Figura 3-37.

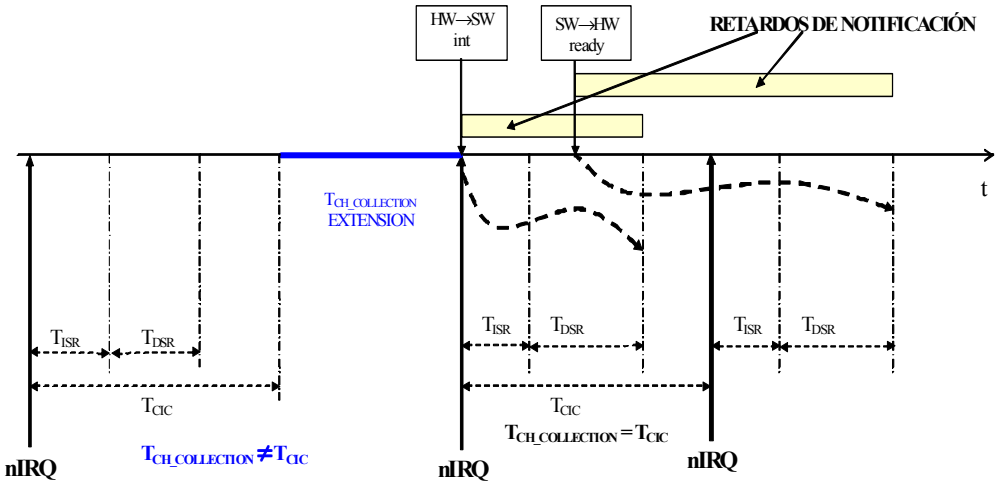



**Figura 3-37. Funcionamiento temporal "macroscópico" del CIC.**

Además de elevar la interrupción, el registro de notificación se carga con los valores del de recolección, y el de recolección queda borrado. De este modo, durante el siguiente intervalo, a la vez que desde el SW se puede leer el registro de notificación y atender las notificaciones realizadas en el intervalo previo, se pueden ir recogiendo las notificaciones del intervalo actual.

Al emplear el mecanismo de interrupciones, se han de manejar las funciones manejadoras de interrupciones. Estas quedan comprendidas en la parte de software de comunicación de propósito general mostrado en la parte izquierda de la Figura 3-34. La Figura 3-34 es específica para el RTOS embebido eCos, caso en el que se emplean dos

rutinas manejadoras: la rutina de atención a la interrupción (ISR) y la rutina de atención a la interrupción retrasada (DSR). La Figura 3-38 muestra el código de ambas.

```
// CHANNELS ISR
cyg_uint32 int_isr(cyg_vector_t vector,cyg_addrword_t data) {
   /* ISR notifies to the CIC that ISR has been executed */
   outi( HW_TO_SW_BASE,0x00 );  // What is written is not important
   /* ISRs must acknowledge the interrupt, or they might be invoked again */
   cyg_interrupt_acknowledge( CHANNELS_IRQ_VECTOR );
   return ( CYG_ISR_HANDLED|CYG_ISR_CALL_DSR );
 }
// CHANNELS DSR
void int_dsr(cyg_vector_t vector, cyg_ucount32 count, cyg_addrword_t data) {
   /* DSR reads HW to SW events register and updates its SW version */
  cyg_flag_setbits(&channel_events_reg,ini(HW_TO_SW_BASE));
}
```

**Figura 3-38. ISR y DSR para implementación de canales HW/SW en plataforma eCos.**

La aserción de la línea de interrupción provoca la ejecución de la ISR. La macro *outi* sirve para escribir un valor entero (segundo parámetro) en un registro hardware mapeado en la dirección apuntada por el primer parámetro. De esta forma, la ISR realiza un acceso de escritura al HW que le indica al CIC que la IRQ va a ser atendida, para que el CIC desaserte la línea de interrupción. Luego, la ISR realiza una llamada al sistema operativo requerida (*cyg_interrupt_acknowledge*) y retorna un código que indica que la ISR ha sido manejada y que requiere el disparo de la DSR. La ejecución posterior de la DSR actualiza una versión software del registro de notificaciones HW→SW. Para ello ha de leer primero mediante la macro *ini* el registro de notificaciones HW→SW. La macro *ini* lee un entero de un registro hardware mapeado en la posición de memoria apuntada por el único parámetro de la macro. La versión software del registro de notificaciones HW→SW es una primitiva de sincronización de eCos: una bandera eCos, que tiene la misma longitud en bits que el registro de notificaciones.

En general, lo que se hace es un proceso de conversión de notificaciones hardware en notificaciones software mediante el mecanismo de interrupciones, las primitivas de atención relacionadas y las primitivas de sincronización del RTOS. Dependiendo del API, la codificación de la ISR y de la DSR tiene que adaptarse a las primitivas de sincronización existente y a las restricciones de uso. El caso de eCos es sencillo, dado que existe una primitiva se sincronización tan sencilla como la bandera eCos, en la que cada bit se comporta de forma aproximada a una notificación en un registro hardware.

En principio, en eCos podría ejecutarse solamente la ISR. Sin embargo, para la implementación de los canales HW/SW es preciso disparar la DSR. Esto es porque en eCos sólo puede llamarse a un número limitado de primitivas del RTOS desde la ISR, lo que no incluye la llamada de sincronización de bandera eCos (*cyg_flag_setbits*, correspondiente a la primitiva de). En cambio, esta limitación no existe para la DSR. Este tipo de limitaciones es común para las rutinas de atención a la interrupción y son aspectos que hay que tener en cuenta para la implementación de canales HW/SW para una nueva API.

Con este mecanismo de notificación HW→SW se puede acotar el retardo desde que se realiza la notificación en las banderas HW y el momento en el que éstas se hacen patentes en la partición SW (en los elementos sincronización de tipo bandera en eCos). Este retardo de notificación se debe fundamentalmente al tiempo de recolección, a las latencias de la ISR ($T_{ISR}$) y de la DSR ($T_{DSR}$). El retardo de notificación se representa en la Figura 3-37, de una forma más "macroscópica", despreciando los tiempos de ejecución de la ISR y la DSR. Estos tiempos suelen ser pequeños, especialmente en el caso de la ISR. En la Figura 3-38 se pudo apreciar lo reducido del código de la ISR y la DSR. En la Figura 3-39 se muestra de forma más detallada la temporización involucrada en este mecanismo de notificación.

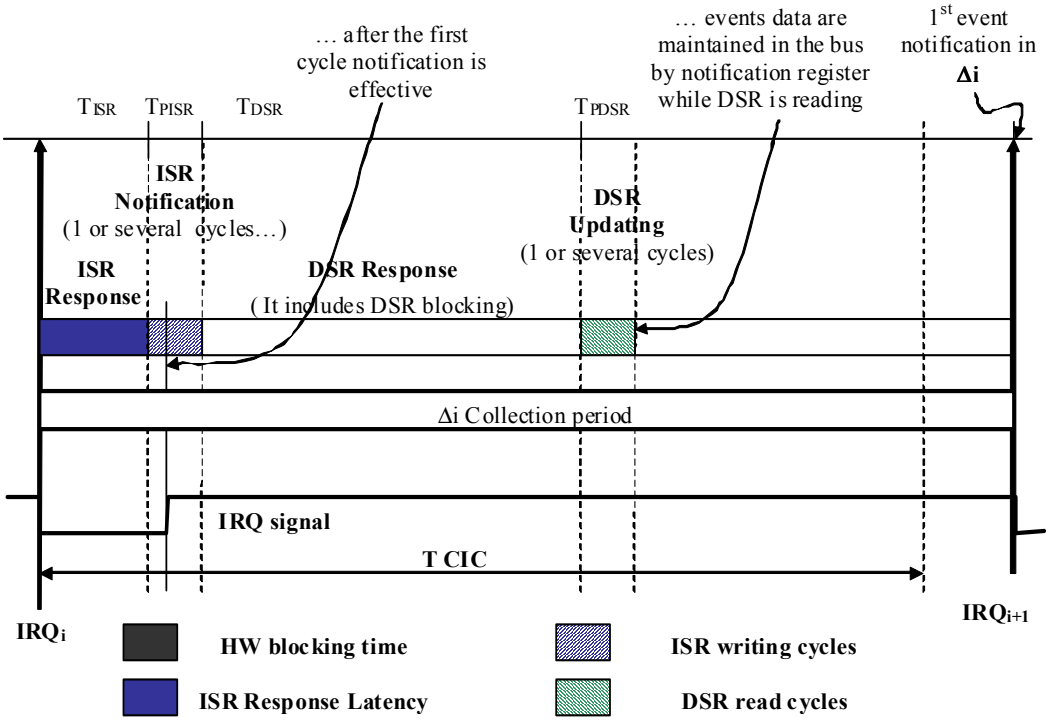



**Figura 3-39. Ciclo temporal detallado del CIC.**

En tanto que $T_{ISR}$ y $T_{DSR}$ son fijados por la plataforma, $T_{CIC}$ es programable. Es, de hecho, un parámetro de diseño del sistema. En una aproximación ruda, tiene una cota inferior determinada por $T_{ISR}$ y $T_{DSR}$ y una cota superior determinada por la mínima frecuencia de notificaciones (y, a la postre, de accesos al canal). En el diseño, existe un compromiso. Por un lado, interesa hacer $T_{CIC}$ tan grande como sea posible para no sobrecargar el sistema. Por otro lado, interesa hacer $T_{CIC}$ tan pequeño como sea posible para minimizar el retardo de notificación. En un diseño correcto de $T_{CIC}$, que cumple $T_{CIC} > T_{DSR} + T_{ISR}$, se cumple a su vez que el retardo de notificación es menor que $T_{CIC}$. En diversas pruebas realizadas con la placa HSDT100, plataforma que se introducirá en el siguiente capítulo, con un reloj a 29.4912MHz, se encontró un límite inferior para $T_{CIC}$ de aproximadamente 50μseg. Es decir, en un caso óptimo, existía un límite de 20.000 notificaciones por canal. Un análisis más detallado de las cotas de $T_{CIC}$ y de su programación se puede encontrar en **[FHSV02]**.

Como se ha comentado, toda esta infraestructura hardware común para la implementación de canales (CIC y registros o memorias intermedias para la transferencia de datos), están mapeados en memoria.

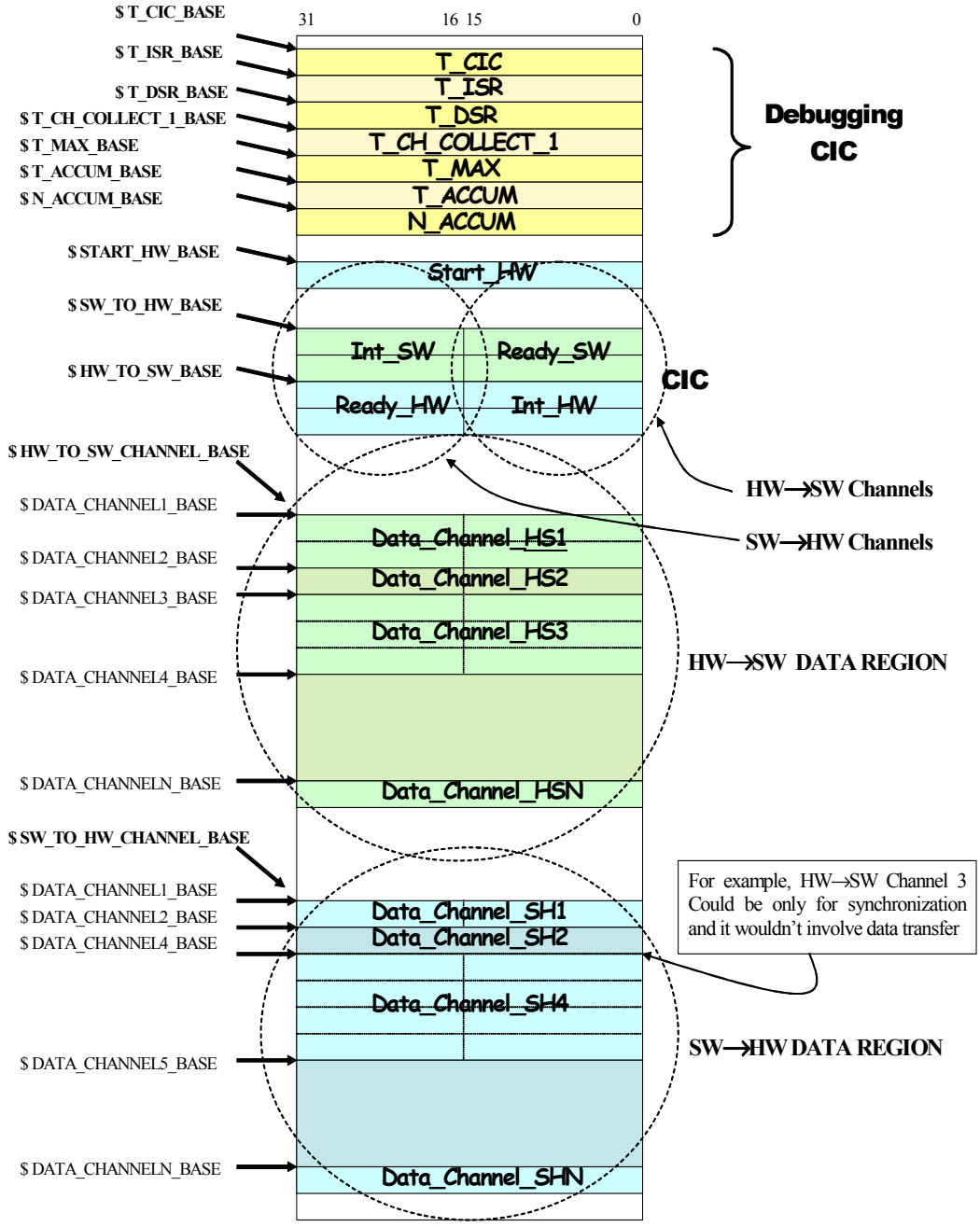



**Figura 3-40. Mapa de memoria para la implementación de canales HW/SW.**

Este mapa de memoria se basa en un plan de asignación de canales. La asignación de canales equilibra el número de canales dedicados a comunicaciones H2S y comunicaciones S2H. El registro de $T_{CIC}$ también se mapea para su programación en número de ciclos. Otros campos mostrados pertenecen al CIC de depurado, una versión de CIC que permite la medida de $T_{ISR}$, $T_{DSR}$, etc, de forma que permite un diseño más

preciso de $T_{CIC}$. Estos campos son opcionales y eliminables en el diseño final. En ese mapa de memoria aparece también un registro *Start_HW*, que permite la inicialización desde software de todo el hardware de interfaz.

Una vez se ha visto en qué consiste el software/hardware de comunicación de propósito general, se puede entender cómo se implementa el canal HW/SW o controlador de canal. En la Figura 3-41 se muestra un ejemplo sencillo para el canal *uc_simple_channel*. Se trata de la implementación de canal en el caso de que, tras la partición software, la interfaz de escritura quede en software y la de lectura en hardware. Por lo tanto, el canal HW/SW en este caso es un canal S2H, con transferencia de datos desde software a hardware que precisa, no obstante, el empleo tanto de notificaciones HW→SW como notificaciones SW→HW.

```
void write(const T& data) {

  // wait until Ready HW = '1'
  cyg_flag_wait(&channel_events_reg,SW_HW_pattern,CYG_FLAG_WAITMODE_OR);
  // to ensure them to return to '0' (false):
  cyg_flag_maskbits(&channel_events_reg,SW_HW_mask_pattern);

  // transfer of data. Valid for any data type
  sw_data_point    = (unsigned int *)&data;
  hw_channel_point = channel_data_base_address;

  // Code valid for 32 bits structures!!
  for(unsigned int i=0; i< num_mem_accesses; i++) {
    outi(hw_channel_point,*sw_data_point);
    sw_data_point++;
    hw_channel_point++;
  }

  // Int_SW=1 for channel= channel_number
  outi(SW_TO_HW_BASE,SW_HW_pattern);

}
```

**Figura 3-41. Implementación del método *write* del canal S2H para el canal *uc_simple_channel*.**

Como se puede ver, en primer lugar, se comprueba, mediante la llamada *cyg_flag_wait* a la bandera eCos (versión software del registro de notificación de canales), si el hilo se tiene que bloquear. Si el canal está lleno, es decir, tiene un dato, la llamada bloqueará el hilo, permitiendo la ejecución de otros hilos software. En el momento en que el proceso hardware lector consuma el dato, realizará una notificación HW→SW tal y como se ha descrito anteriormente, de manera que la notificación pasa del registro de recolección, al de notificación y de ahí a la bandera SW, a través de los disparos de la ISR y la DSR. Por lo tanto, la semántica de la notificación HW→SW, en este caso es que el HW ya ha consumido el dato anterior y la memoria intermedia está lista para ser escrita por el software. Una vez que esa notificación está hecha, el hilo que ha hecho la llamada *write* se desbloquea y el planificador del RTOS le puede pasar a estado de ejecución. Posteriormente se realiza la trasferencia de datos. En este canal, se trata de una única unidad de datos de tipo genérico (es decir, el tipo pasado a la plantilla de canal). En la Figura 3-41 se muestra un código de implementación genérico para el tipo de dato y asumiendo una longitud de palabra de 32 bits. Los registros del módulo SW_HW_DATA de la Figura 3-34 toman también esta longitud.

En ese esquema, *num_mem_accesses* es una variable que en el constructor del canal SW/HW se inicializa con el número de accesos de 32 bits necesarios para completar la transferencia de un dato del tipo que toma la instancia del canal. Este número se calcula en el constructor del canal HW/SW. Así, por ejemplo, si se trasfieren

unidades de datos de tipo *byte* o *unsigned int*, el valor será normalmente 1. Si se transfieren unidades de datos de un tipo que tome más de 32bits, el valor será mayor que 1. Esta implementación de la transferencia de datos en los canales HW/SW, no considera la estructura de los tipos de datos y realiza una transferencia genérica. No obstante, esta solución puede no ser simple para todos los casos, y una especialización de canales HW/SW en función del tipo puede ser conveniente y ofrecer soluciones óptimas. La razón se muestra en la Figura 3-42, donde dos declaraciones distintas de una estructura de datos con los mismos campos puede dar lugar a organizaciones y tamaños distintos en el mapa de memoria. En general, esta organización depende de la declaración del tipo, del compilador y de las opciones de optimización. La solución pasa por imponer reglas en la declaración de tipos y en fijar las opciones que fijan el alineamiento de datos en memoria.

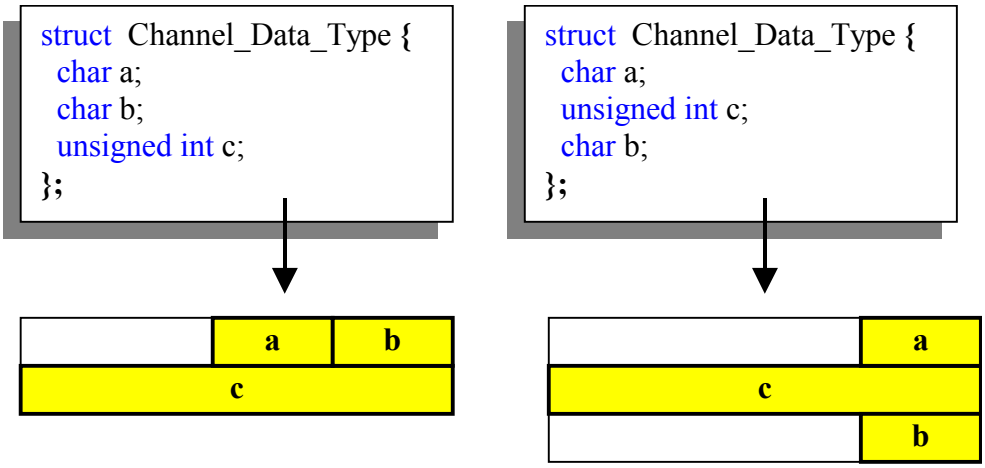



**Figura 3-42. Diferentes organizaciones de los campos de una estructura en memoria.**

Finalmente, el canal S2H de la Figura 3-41, realiza una notificación SW→HW, mediante una escritura directa en el registro de notificación SW→HW, mapeado en la dirección SW_TO_HW_BASE. La variable *SW_HW_pattern* lleva habilitado únicamente la bandera del registro de notificación SW→HW que ha sido asignada para la instancia de canal que realiza la llamada. La semántica de la notificación SW→HW es aquí la de que ya se ha realizado una escritura y existe una unidad de datos lista para ser leída.

Esta implementación de canal HW/SW se puede extrapolar de forma relativamente sencilla a canales algo más complejos como el *uc_fifo*, que se diferencia en que tiene una memoria intermedia de mayor longitud, lo que reduce el número de condiciones en las que se da notificación.

### *3.5.8.3 Canales de Entrada/Salida*

En especificación, los canales de entrada/salida, no se diferencian, en principio, del resto de canales, salvo por encontrarse en la frontera entre módulo de sistema y módulo o módulos de entorno.

La casuística de implementación de los canales de entrada/salida puede contemplar, en implementación de software varias combinaciones. Por ejemplo, puede ocurrir que tanto módulo de entorno como módulo de sistema acaben implementados en

software, por lo tanto, la implementación del canal de I/O se realiza mediante un canal SW/SW.

El caso más común será el que se ejemplifica en la Figura 3-43. Por restricciones de la aplicación, que a su vez fijan la plataforma de implementación, queda fijado el uso de una interfaz de entrada/salida. Por ejemplo, asúmase que se ha especificado un sensor que ha de enviar datos a una estación base, que se modela en SystemC como un módulo de entorno. La aplicación fija que sólo un enlace inalámbrico es viable. Entonces se elige una plataforma que provee este interfaz de I/O. Normalmente, las plataformas seleccionadas proveen el hardware básico de I/O (antenas, moduladores, registros mapeados en memoria, etc) y un software controlador. Por lo tanto, el canal de I/O es un canal HW/SW que maneja ese controlador. Normalmente es un canal HW/SW porque, aunque se realicen llamadas a las funciones del controlador de I/O, estas no pasan por el API genérico del RTOS embebido.

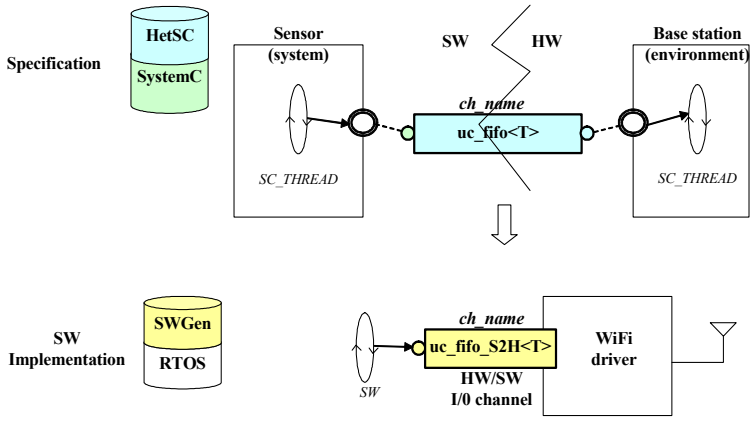



**Figura 3-43. Implementación de canal de I/O.**

## 3.5.9 Configuración de la Librería

La librería *SWGen* provee un fichero de cabecera que permite configurar la acción de la librería y así sintonizar o especificar más detalladamente la implementación en función de parámetros que tienen en cuenta detalles de la plataforma. Estos parámetros pueden tomar valores distintos en función de las decisiones de diseño. Entre estos parámetros se encuentran: la política de planificación, el tamaño de pila asociado a cada hilo de implementación, si éste se aplica homogéneamente o no, la implementación del *sc_main* como un hilo o no, etc.

Por defecto, estos parámetros están configurados para que las distintas soluciones de implementación sean factibles y funcionalmente equivalentes a la especificación de sistema. Así, por ejemplo, una implementación que funcione tanto para política de planificación expulsora como no expulsora se toma por defecto frente a una implementación más eficiente, pero que es válida solo para una política de planificación no expulsora.

No obstante, no toda configuración por defecto de estos parámetros, y de la librería en general, puede garantizar que un resultado de la generación SW factible y/u optimo. Es el caso típico del tamaño de pila. Por lo común, el tamaño de pila (bien

conjunto, bien asignado a cada hilo) es parte de la configuración del RTOS, que, a su vez, depende de la capacidad de memoria de la plataforma concreta.

La configurabilidad de estos parámetros prepara *SWGen* para un posible flujo de diseño, en el que una herramienta, deseablemente en el nivel de sistema, sea capaz de encontrar los valores factibles y óptimos de estos parámetros. Por ejemplo, actualmente, la librería configura por defecto un tamaño de pila que el usuario puede cambiar. Este tamaño de pila se aplica a todos los procesos del sistema. En la implementación actual, es posible también aplicar en especificación un parámetro de tamaño de pila que se utiliza sólo en generación de software (en especificación no tendría efecto). Para ello, la librería de generación de software implemente la llamada SystemC *set_stack_size*, actualmente comprendida en el estándar SystemC. No todos los procesos del sistema tienen las mismas necesidades de pila, que además, depende de datos en general. Una herramienta que calcule el tamaño de pila de cada hilo podría integrarse con *SWGen*. Esta podría situarse antes, realizándose en un nivel de sistema y luego admitir un proceso de refinamiento iterativo, requiriendo la aplicación de la librería *SWGen*. La afinación de este parámetro, como se muestra en la sección 4.3.2 puede llegar a reducir considerablemente la huella de memoria. Sin embargo, un ajuste demasiado pequeño puede llevar a un fallo en tiempo ejecución.

La optimización que se puede obtener en el caso del tamaño de pila parece intuitiva. El impacto que produce la elección de otros parámetros, tales como la política de planificación, puede parecer en cambio más cuestionable. Sin embargo, la decisión de este parámetro afecta a los requisitos y complejidad tanto de la plataforma como de *SWGen*. Por ejemplo, una planificación no expulsora permite la escritura de un código de SWGen más simple y no requiere de la plataforma un temporizador hardware (un contador de frecuencia conocida). De igual modo, la política de planificación determina qué optimizaciones (en forma de reducción de código fuente) se pueden hacer en el código de *SWGen*. Esto se podía apreciar en la Figura 3-32, tanto en la implementación de SC2eCos como en la de SC2POSIX. Ahí, la utilización del mutex de acceso está condicionada por una variable de preprocesado (_WITH_LOCK). Esta variable controla si se emplea o no el mutex de acceso. Siempre que en especificación se haya seleccionado la opción de admitir varios escritores y que la planificación del RTOS embebido sea expulsora, será necesario el uso de este mutex de acceso. En caso contrario, el uso del mutex puede ser eliminado y, por tanto, se aumenta la velocidad de acceso y se reduce el tamaño de código. La implementación con la inclusión del mutex de acceso funciona independientemente de la política de planificación seleccionada. Por ello, esta es la implementación por defecto.

Las posibilidades de optimización dependen del tipo de implementación. Esto se ilustra en la Figura 3-44, que muestra dos implementaciones SW/SW del método *write* de *uc_fifo*, realizadas bajo el mismo API (POSIX). La primera es la implementación SC2POSIX ya mostrada en la Figura 3-32, y que ahora se denomina implementación 1. La segunda es una nueva versión, funcionalmente equivalente, en la que el canal SW/SW se resuelve mediante un mutex y dos variables de condición, denominada implementación 2.

**SC2POSIX (IMPLEMENTATION 1)**

```
template <class T>
inline void uc_fifo_SS<T>::write( const T& val_ ) {

#ifdef _WITH_LOCK
  pthread_mutex_lock(&writer_mutex);
#endif

  sem_wait(&room_to_write);

  // dump of the data
  buffer[write_index] = val_;

  // buffer write index update
  if(write_index>=(size-1))
    write_index=0;
  else
    write_index++;

  sem_post (&data_to_read);

#ifdef _WITH_LOCK
  pthread_mutex_unlock(&writer_mutex);
#endif

}
```

**SC2POSIX (IMPLEMENTATION 2)**

```
template <class T>
inline void uc_fifo_SS<T>::write( const T& val_ ) {

  pthread_mutex_lock(&access_mutex);

  while (number_of_tokens==size) {// fifo_full
    pthread_cond_wait(&room_to_write,&access_mutex);
  }

  // dump of the data
  buffer[write_index] = val_;

  // buffer write index update
  if(write_index>=(size-1))
    write_index=0;
  else
    write_index++;

  number_of_tokens++;
  if(number_of_tokens==1) {
    pthread_cond_signal(&data_to_read);
  }

  pthread_mutex_unlock(&access_mutex);

}
```

**Figura 3-44. Dos implementaciones POSIX del acceso de escritura del canal uc_fifo.**

Como ocurría entre la implementación SC2eCos y SC2POSIX de la Figura 3-32, las dos implementaciones SC2POSIX de la Figura 3-44 presentan una estructura muy semejante. Como se ha mencionado, la diferencia se da en cuanto a las llamadas de sincronización realizadas. La implementación 2 emplea variables de condición tanto para la condición de espacio para escritura como para la notificación de datos en la cola. Eso quiere decir que la implementación 2 tiene que declarar y actualizar explícitamente dos variables para mantener el tamaño de la memoria intermedia y el número de unidades de datos en la cola (*size* y *number_of_tokens* respectivamente). Esto, además de expandir un poco el código de la implementación 2 frente al de la 1, hace que la implementación 2 necesite un mutex común tanto para el acceso de escritura como para el de lectura (*access_mutex*), a diferencia de la implementación 1, que independiza los accesos de escritura de los de lectura con dos mutexes, uno de escritura (*writer_mutex*) y otro de lectura. La necesidad del mutex común surge de que la variable *number_of_tokens* es actualizada tanto por el método de escritura (*write*) como por el método de lectura (*read*) y es, por tanto, una variable compartida.

Por lo tanto, dependiendo de la implementación, una optimización del código que depende de una opción de diseño de la plataforma (la política de planificación) estará disponible o no. Por otro lado, se debe notar que el que una implementación disponga de opciones de eliminación de código, esa eliminación no implica necesariamente la implementación óptima. Por ejemplo, aunque la implementación 2 en la Figura 3-44 tiene que usar dos variables más, la implementación que el sistema operativo haga de los semáforos será más pesada que la de la variable condicional, ya que, de hecho, los semáforos han de mantener internamente la variable de cuenta y de tamaño del semáforo. Por lo tanto, qué implementación es óptima, tanto en velocidad como en tamaño puede depender de cuan bien optimiza cada RTOS la implementación de esas primitivas. No obstante, la posibilidad de configuración dada por la estructura de *SWGen* abre la puerta a la exploración de implementaciones para distintas APIs, y para la misma API, jugando con otros parámetros como tamaño de pila, tipo de planificación, etc.

## 3.6 Preservación de propiedades del MoC en Síntesis de SW

En el nivel de especificación, el empleo de un MoC específico aportaba una serie de propiedades beneficiosas para el manejo de especificaciones concurrentes. En esta sección se explica cómo la metodología consigue que la implementación SW generada conserve las mismas propiedades que la especificación.

La solución más sencilla a este problema es que el proceso de generación no suponga refinado alguno. Es decir, que los supuestos y reglas del MoC que proporcionan en la especificación las propiedades deseadas se mantengan en el código software generado. En la metodología propuesta, esto se cumple para un número importante de las reglas que se mostraron en el capitulo de especificación. Por ejemplo, en el nivel de especificación, una de las reglas básicas es el paralelismo acotado y estático. El proceso de generación software preserva esta regla, ya que no añade procesos o hilos adicionales a los existentes en la especificación SystemC, ni de forma estática (en el proceso de generación) ni dinámica (introduciendo estamentos de generación dinámica de hilos). Asimismo, en el nivel de especificación, la estructura jerárquica y de comunicación es estática, lo que significa que, por ejemplo, en una especificación KPN, no es posible que aparezca una nueva conexión de un canal a un proceso productor o consumidor en tiempo de simulación. El proceso de generación propuesto preserva la estructura jerárquica y de comunicación de la especificación original y la implementación no resultará, por ejemplo, en que un canal SW/SW se conecte a nuevos hilos en tiempo de ejecución.

Sin embargo, en la metodología de generación *SWGen* existe una transformación de la plataforma de ejecución o implementación, cuyo modelo o caracterización se puede denominar MoC base. Esto es así toda vez que la metodología *SWGen* sustituye para la ejecución de la especificación un kernel de eventos discretos por un RTOS embebido. Por lo tanto, la implementación de las facilidades de especificación y su correspondiente versión en SW utilizan, además de llamadas a un API distinto, un modelo de ejecución distinto.

La diferencia de APIs entre la implementación de sistema y la implementación SW se ha mostrado en la sección 3.5.8.1 (Figura 3-30 y Figura 3-31). Por ejemplo, en la Figura 3-31, la implementación SystemC de un canal *uc_fifo* maneja eventos SystemC, esperas y notificaciones a los mismos, en tanto que la implementación de los canales SW/SW emplea llamadas al sistema a través de un API de RTOS.

Ahora, la Figura 3-45 resalta además las diferencias en el modelo temporal entre el kernel de simulación SystemC y la plataforma de implementación SW. En la parte superior de la Figura 3-45 se representa cómo la implementación SystemC hace uso de un modelo de tiempo que divide la ejecución en elaboración y simulación. Esta última divide el tiempo en avances temporales, que pueden constar de uno o más ciclos delta. El ciclo delta, a su vez se divide en una fase de evaluación y otra de actualización. Como se comentó en secciones anteriores, SystemC permite un gran control del comportamiento del canal en cada uno de esos pasos y en sus transiciones.

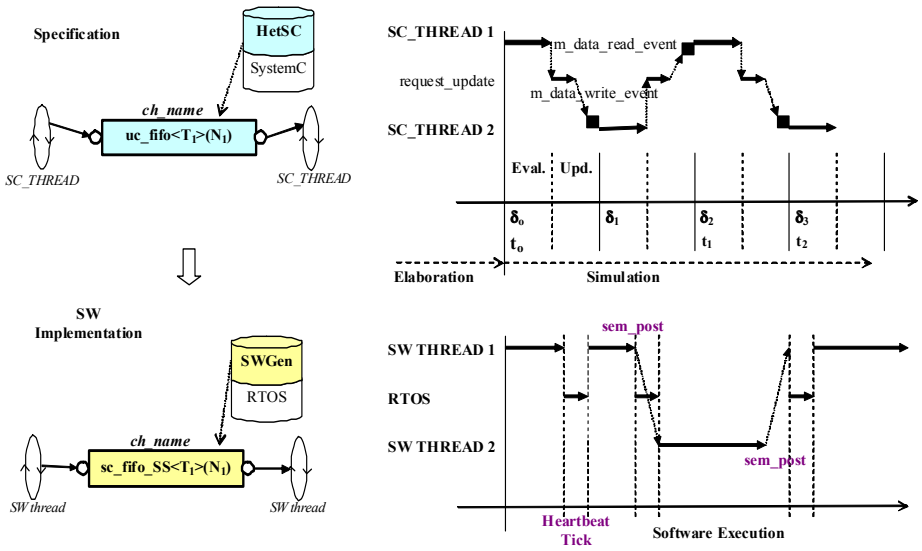



**Figura 3-45. Transformación del modelo temporal en la implementación SW.**

En cambio, en la parte inferior de la Figura 3-45, se representa el modelo de tiempo que maneja un programador de SW habitualmente. Es un modelo asíncrono, sin nociones de división temporal alguna ni de tiempo estricto, es decir, el instante absoluto en el que se sitúa cada evento a partir de un origen dado. Pese a que la ejecución del SW se realice realmente ciclo a ciclo por uno o más procesadores, esto es transparente desde el punto de vista de programador de SW.

En la metodología *SWGen*, la traslación a software se hace desde la semántica abstracta del canal SystemC (bien sea un canal primitivo o un canal *HetSC*). Tanto la implementación SystemC como la implementación *SWGen*, cumplen la semántica del canal, que proporciona las propiedades que caracterizan al MoC y que, por tanto, se quiere preservar. Esa traslación no supone un refinamiento complejo o poco intuitivo, ya que el refinamiento que tiene lugar en la traslación es el que va del MoC abstracto al que sigue la implementación. Esto se refleja con la flecha ancha de la Figura 3-46.

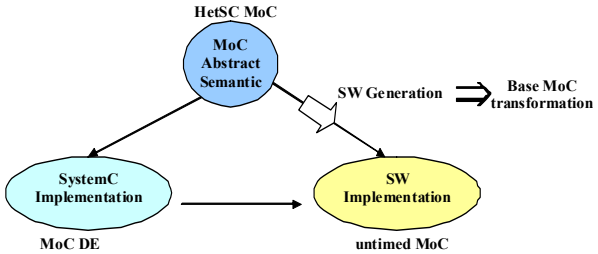



**Figura 3-46. La generación de SW supone un refinamiento desde la semántica abstracta del MoC.**

Por ejemplo, si el acceso de escritura a un canal fifo es bloqueante, bloqueando al proceso SystemC que accede si la fifo está llena, igualmente, en implementación SW, el canal de implementación ofrece una semántica bloqueante al acceso del hilo escritor (que implementa en software el proceso escritor SystemC) y la condición de escritura sigue siendo la misma (que la fifo esté llena). Para ese acceso de escritura, la diferencia en otros detalles semánticos temporales entre la implementación SystemC y la implementación SW es irrelevante.

Por lo tanto, en la metodología *SWGen*, el refinamiento no se realiza desde la semántica temporal exacta de la implementación SystemC. Es decir, no se implementa el ciclo delta en el software embebido (tal y como hace la metodología de la sección 3.2.4.2). Como se explicó en el Capítulo 2, el usuario *HetSC* no asume o maneja toda esa información, sino que maneja una semántica abstracta del canal. La implementación SystemC hace simulable esa semántica abstracta. Para ello la contiene, pero también necesita detallar más información, el mapeo al eje (t,δ), el cual no tiene porqué trasladarse a software.

En la Figura 3-47 se detalla la Figura 3-46 para el caso de la Figura 3-45. En la parte superior se representa la ejecución que el especificador espera a priori de la semántica abstracta de un canal *uc_fifo*. En ella se asume planificación no expulsora. Esta es la que SystemC explícitamente asume (tal y como se especifica así en el LRM) y permite un modelado más sencillo del sistema.

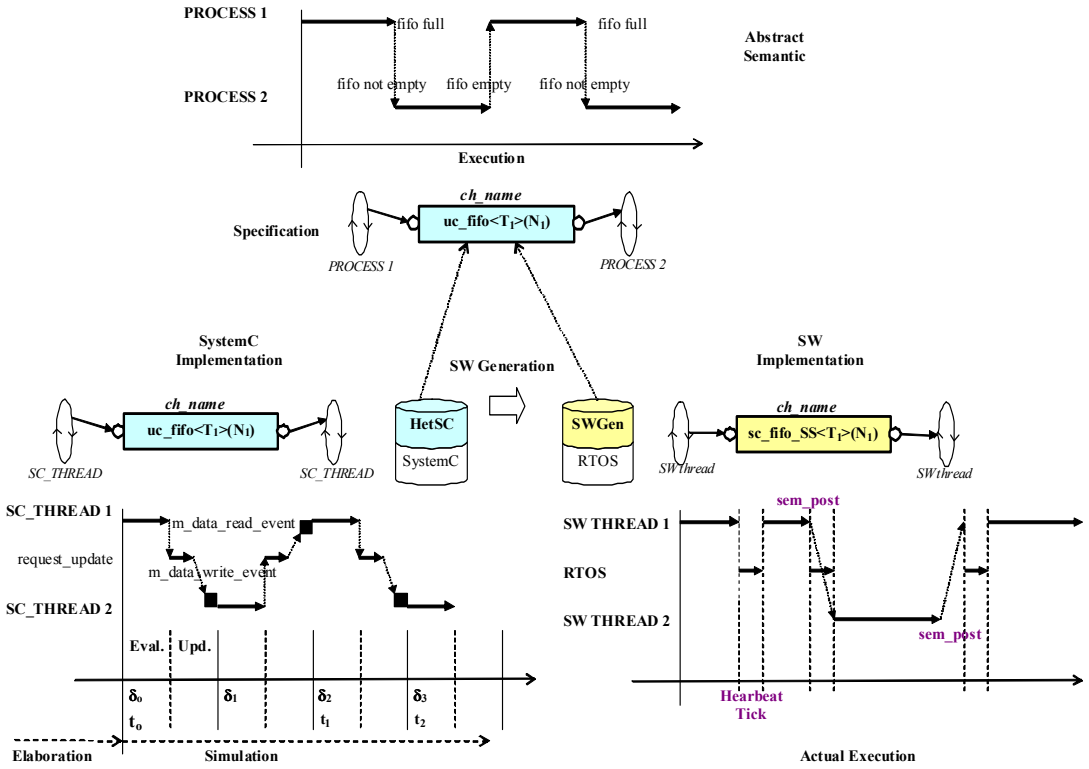



**Figura 3-47. Semántica abstracta y semánticas de implementación del canal uc_fifo.**

En la parte inferior izquierda se representa la semántica temporal que provee la implementación SystemC del canal. Esta contiene más información que la que realmente precisa el usuario para construir la especificación. Esa información hace ejecutable la especificación. Se ha representado un detalle de cómo el control pasa del proceso al código de los canales (*request_update*). Se han omitido otros detalles, como el paso del control al propio kernel de simulación de SystemC. En cualquier caso, lo importante es notar el nivel de detalle del modelo temporal de la implementación SystemC. En *HetSC*, se realiza una abstracción de la información temporal que maneja

el kernel de simulación DE y el usuario realmente no maneja ese detalle, sino el modelo abstracto de la parte superior de la Figura 3-47.

En la parte inferior derecha de la Figura 3-47 se representa la implementación SW. En ella aparecen las transiciones entre los hilos que implementan los procesos de la especificación y también la actuación del RTOS. Se ha representado una primera planificación debida a una expiración del latido del sistema. La implementación del canal, en este caso, garantiza que el control retorna al hilo productor (SWTHREAD 1) en tanto no termina la transferencia. Una vez ésta se completa, la implementación puede realizar un *sem_post* que posibilita que la segunda planificación pueda ceder el control al hilo lector. La segunda planificación puede aparecer por un nuevo *tick* del sistema o, por ejemplo, porque el hilo escritor realiza un *sem_wait* y se bloquea por llenado de la fifo. Como se puede comprobar, en este caso, la implementación SW, salvo por la consideración de las planificaciones del latido del sistema, es más cercana al modelo asíncrono propio del MoC atemporal empleado en la especificación que al modelo DE que hay que considerar para predecir la semántica de un canal SystemC considerando su implementación. De este modo, en este caso, mientras que la transición inferior de modelo DE a modelo SW supone una abstracción, la generación de software sin embargo, se puede concebir prácticamente como un mapeo o un refinamiento. En cualquier caso, *SWGen* produce una implementación software equivalente a la implementación SystemC en términos de preservación de la semántica abstracta manejada por el usuario *HetSC*.

Tanto la implementación SystemC como la implementación SW son refinamientos del MoC de especificación (el empleado por el usuario HetSC). Alternativamente, en el Capítulo 2, se mostraba el MoC de especificación como una abstracción realizada sobre el MoC DE de tiempo estricto. Esta discusión conduce a la conveniencia de la formalización de los refinamientos y abstracciones realizados tanto por la metodología de especificación como por la de generación de SW propuesta. Esto no es objeto del presente trabajo. Este aspecto está relacionado tanto con la metodología de especificación *HetSC* como con la de generación *SWGen*. El primer caso es objeto de estudio en el proyecto ANDRES **[AND08]**.

En resumen, la metodología de generación SW propuesta resuelve muchos de los aspectos de generación SW como un mapeo. También presenta facetas de síntesis, en tanto que elimina código innecesario en generación de SW y asociado a las actividades y chequeos propios del nivel de especificación. También ofrece una posibilidad de síntesis en el sentido de habilitar la elección entre distintas posibilidades de implementación. En ese sentido, la librería presenta distintas opciones de configuración.

# Capítulo 4

# Resultados Experimentales

*El desarrollo de las metodologías presentadas anteriormente ha requerido un desarrollo paralelo de pequeños ejemplos específicos y de algunos de mayor entidad, como un Vocoder GSM. Estos experimentos han permitido obtener una serie de datos que avalan las ventajas de las técnicas de especificación y generación de software propuestas. Este capítulo pretende dar también una visión global de las posibilidades de integración de estas técnicas en flujos reales con datos concretos acerca de plataformas de desarrollo, plataformas objetivo y herramientas utilizadas. No se hace, en cambio, una descripción exhaustiva de los ejemplos mostrados, que no obstante, están documentados y disponibles junto con las librerías metodológicas.*

## 4.1 Plataformas Objetivo

La consideración de los distintos parámetros básicos que caracterizan una plataforma HW/SW (arquitectura, RTOS, API de RTOS, tamaño de los distintos tipos memoria, presencia y capacidad de lógica programable, etc) hace muy extenso el número de combinaciones que pueden configurar una plataforma objetivo específica. Por tanto, es difícil una validación exhaustiva de las metodologías propuestas, especialmente, de la de generación de software. En este trabajo se han realizado diversos experimentos, tratando de que los resultados obtenidos representen una gama representativa y relativamente amplia de plataformas objetivo. De este modo, se han probado combinaciones para los siguientes parámetros básicos:

**Arquitecturas Software:** *ARM* y *OpenRISC*.

**RTOS embebido:** *eCos* y *Linux Embebido*.

**Arquitecturas de Hardware Programable:** *ALTERA* y *Xilinx*.

A continuación se concretan las plataformas con las que ha se ha comprobando y experimentado el funcionamiento de las metodologías.

### 4.1.1 Plataforma HSDT100 de Sidsa

En la Figura 4-1 se muestra la primera plataforma de experimentación. Se trata de la plataforma HSDT100 de Sidsa **[Sid08]**.

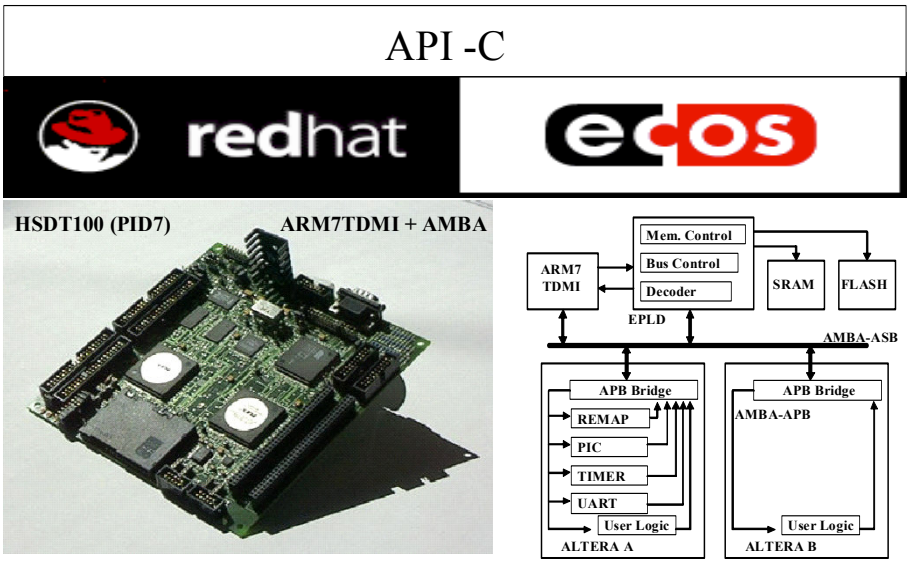



**Figura 4-1. Plataforma (*eCos*-ARM7) HSDT100 de Sidsa.**

El núcleo de esta plataforma es un chip que integra el procesador de 32 bits ARM7TDMI (sin memoria cache), con memoria *flash* de 1Mbyte y RAM de 1Mbyte. La plataforma incluye también dos FPGAs ALTERA de 100Kpuertas. En esta plataforma, parte del hardware de periferia eran módulos Verilog, que se sintetizaban junto con el hardware específico para generar el fichero de configuración de una de las FPGAs (ALTERA A). El RTOS *eCos* completaba la parte software de la plataforma.

Para esta plataforma, la librería *SWGen* se configuraba para generación para el API-C de *eCos*.

### 4.1.2 Plataforma OpenRISC 1500 del GIM/UC

La Figura 4-2 muestra otra plataforma con la que se ha probado la metodología para arquitectura software OpenRISC **[ORSC00]**. Se trata de una placa proporcionada por DS2 **[DS208]**, usada para prototipado de modems PLC, para comunicación a través de la red eléctrica. El cómputo de esta plataforma es realizado en 4 FPGAs Virtex II (cada una en un chip, tal y como se observa en la Figura 4-2). Esto dota a esta placa de una gran capacidad de lógica configurable, de manera que todo el hardware de plataforma, procesador incluido, era hardware configurable. Esto permitió ejemplificar un diseño de SoC en el que hay un control total, tanto de la arquitectura software, como del hardware específico. En este caso, el procesador empleado fue el OpenRISC 1500 **[BPCH04]**, siendo tanto este procesador como la arquitectura de la plataforma un diseño del grupo GIM. Este control de la plataforma permitió, además de verificar la metodología de generación de SW, obtener y cotejar datos de estimación de rendimiento sobre la especificación de nivel de sistema con precisión **[PHFSV04]**.

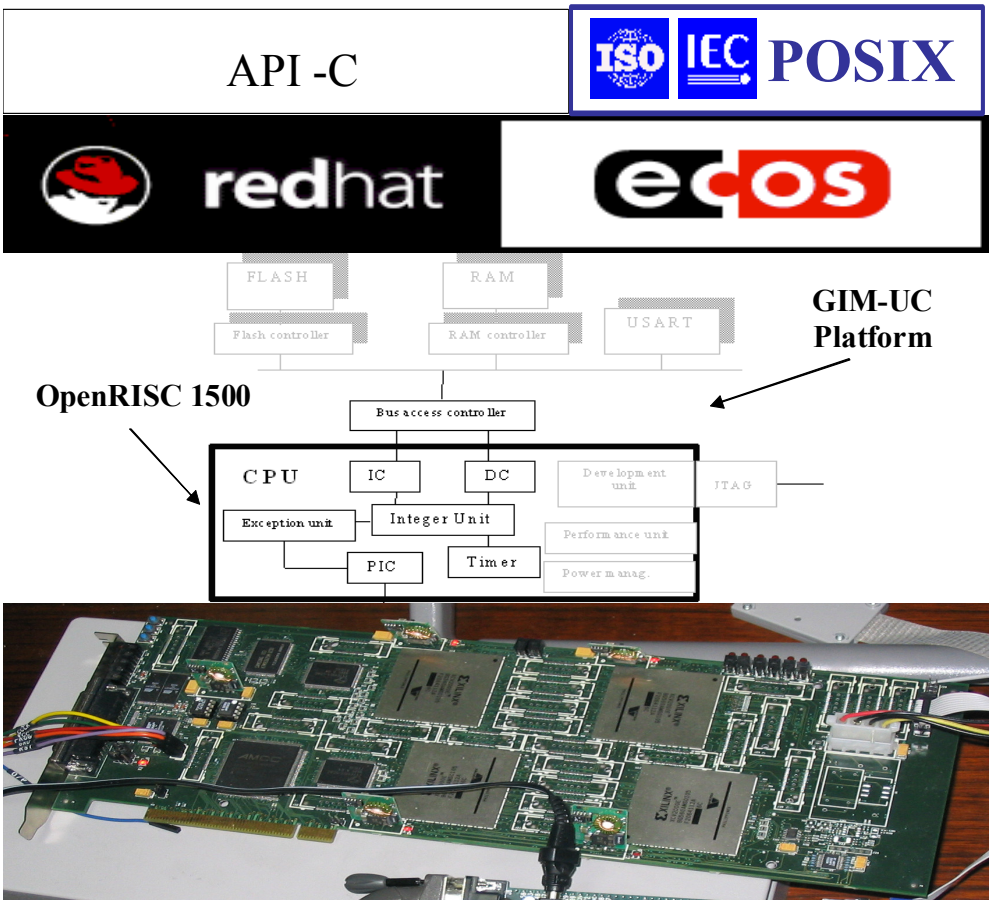



**Figura 4-2. Plataforma *eCos*-OpenRISC en la placa de DS2, con 4 FPGAs Virtex II.**

Esta plataforma se probó para dos versiones de *eCos* (*1.3.1 y 2.0*). La particularización (o "*port*") de *eCos* para esta plataforma era también un desarrollo del

grupo GIM. La versión 2.0 de *eCos* ya soportaba el API-POSIX, lo cual, hizo posible el desarrollo y prueba de la capacidad de mapeo de *SWGen* al API POSIX, tal y como se ha resaltado en la Figura 4-2.

### 4.1.3 Plataforma Excalibur EPXA1 de Altera

En la Figura 4-3 se muestra la plataforma Excalibur EPXA1 de Altera. Esta plataforma presenta un dispositivo EPXA1F484C1, con un procesador ARM922T y lógica configurable de hasta 263Kpuertas, con una memoria flash de 8Mbytes y una SDRAM de 32Mbytes. En esta plataforma, tanto el núcleo procesador como la lógica configurable se encuentran en el mismo chip. Además, mediante las herramientas de desarrollo de Altera (Quartus II) es posible configurar diferentes aspectos de la plataforma, como la habilitación de las líneas de interrupciones, el mapa de memoria, etc.


**Figura 4-3. Plataforma con *eCos* en Excalibur de ALTERA.**

### 4.1.4 Plataforma CSB536FS de Freescale

En la Figura 4-4, se representa la última plataforma con la que se han ejercitado las metodologías propuestas. Se trata de la CSB536FS de *Freescale* **[Free08]**. Tiene un procesador i.MXL (con un núcleo ARM920T), 8Mbytes de flash y 64Mbytes de RAM. Esta plataforma tiene soporte para GX-Linux, una distribución de Linux Embebido. La aplicación accede a sus servicios mediante un API POSIX, con la cual se ha probado la metodología *SWGen*. Por tanto, en este caso, *SWGen* se configuraba para generación para esa API. Así, se ha demostrado cómo el soporte de un API POSIX en *SWGen*, desarrollado inicialmente para mapeo sobre una plataforma basada en *eCos*, permite la aplicación de la metodología a otros RTOS que también soportan la misma API.

Esta plataforma representa las crecientes capacidades que han ido adquiriendo los sistemas embebidos. En términos de memoria, los 8Mbytes de flash, hacen posible que un RTOS con una huella de alrededor de 1Mbyte, como Linux, no ocupe más del 15% de la memoria ROM del sistema, quedando un margen amplio para el valor añadido.

En términos de entrada salida, el avance ha sido también notable y se ha reflejado en las plataformas empleadas. En tanto que la HSDT100 presentaba apenas una interfaz serie RS232 y otra interfaz para tarjetas inteligentes, la placa de DS2 contaba con un conector PCI y la Excalibur EPXA1 con una interfaz Ethernet. En el caso de la CSB536FS, esta placa tiene una variedad importante de interfaces de I/O, de los que destacan una interfaz Ethernet (que se puede emplear para el depurado, carga y descarga de aplicaciones), dos conectores RS232, una interfaz USB y una pantalla de entrada/salida de cristal líquido de 4"x4".


**Figura 4-4. Plataforma CSB536FS de Freescale con GX-Linux.**

## 4.2 Entorno y Herramientas de Desarrollo

Un objetivo importante en el desarrollo de las metodologías presentadas es que estas sean flexibles y se puedan integrar en entornos de desarrollo de bajo costo, de forma que sean asequibles a la pequeña y mediana empresa. Se trata, asimismo, de preservar la productividad que proveen las herramientas de desarrollo de sistemas embebidos básicas, tales como los compiladores cruzados, etc. Se han probado diferentes combinaciones de entornos que prueban la capacidad de integración de las librerías *HetSC* y *SWGen* con herramientas usuales en el diseño de sistemas heterogéneos y generación de software embebido. En la Figura 4-5, se da un esquema genérico del entorno de desarrollo en el que se han integrado las metodologías y librerías *HetSC* y *SWGen*. Como se puede ver, no es un entorno que imponga requerimientos muy específicos en cuanto a tipo de computador y sistema operativo anfitrión, herramientas de edición de código y herramientas de desarrollo cruzado. De hecho, las librerías *HetSC* y *SWGen* se han instalado e integrado en diversas combinaciones de entorno. Para los experimentos se ha empleado en la mayoría de las ocasiones un computador personal (PC), aunque también es posible usar una estación de trabajo (*Workstation*). Como sistema operativo de desarrollo se ha utilizado normalmente *Linux*, aunque las librerías metodológicas se pueden instalar sobre *Unix* y *Windows+Cygwin* también.

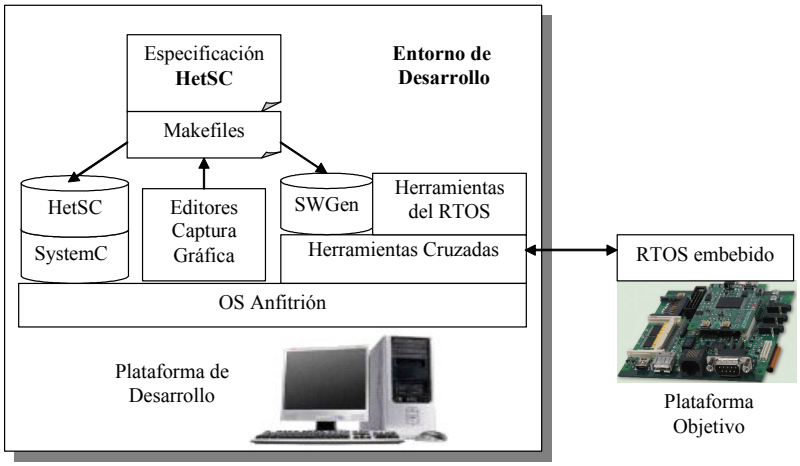



**Figura 4-5. Integración de *HetSC* y *SWGen* en el entorno de desarrollo.**

Para la compilación y simulación de la especificación heterogénea (parte izquierda de la Figura 4-5), es preciso un compilador de C/C++. Principalmente se ha empleado GCC (de GNU). Este compilador suele estar disponible como parte de la distribución del SO en el caso de *Linux*. La incorporación al flujo de otros compiladores (Borland, VisualC, etc) es relativamente sencilla y más una cuestión de adaptación del código de las librerías metodológicas que de innovación en las mismas. Hasta la fecha, la librería *HetSC* se ha ido actualizando para permitir la compilación con las últimas versiones de gcc (4.x) y de SystemC (2.2). Asimismo, las librerías también se han probado en un entorno *Linux* de 64 bits. Para la edición del código de fuentes de la especificación y los *Makefiles* se han usado editores suministrados con el sistema operativo de la plataforma de desarrollo. La mayoría viene con características de resaltado sintáctico para C/C++ y algunos permiten definir plantillas para el resaltado de SystemC.

En la Figura 4-5 se ha representado también la necesidad de instalación de herramientas cruzadas (requeridas por *SWGen*). Normalmente, estas herramientas son provistas por el desarrollador de la plataforma o por un tercero. Normalmente, bastará una instalación de compilador cruzado para una arquitectura determinada. Por ejemplo, para compilar para un ARM7 y para un ARM9 se puede utilizar la misma instalación del compilador de GNU *gcc* cruzado (*arm-elf-gcc*).

En algunas ocasiones, las herramientas se particularizan y optimizan para alguna plataforma en particular. Por ejemplo, en el caso de la HSDT100, se ha empleado un conjunto de herramientas cruzadas de un tercero: *OCDemon* de *Macraigor Systems* **[OCD08]**. Consistía en una versión precompilada de las utilidades binarias (*binutils*) de GNU, del compilador *gcc* y del depurador *gdb*. En este caso, se optó por estas herramientas porque la particularización (*port*) del depurador (*gdb*) permitía la conexión, descarga de la aplicación y depurado de la aplicación embebida a través del *Wiggler*, una interfaz de adaptación de bajo costo entre el puerto paralelo y la interfaz JTAG. Entre estas herramientas se empleó también el *OCDRemote*, que permitía un depurado remoto de la plataforma objetivo. Esta conexión entre el entorno de desarrollo y plataforma objetivo (que se ha representado como una línea de doble dirección en la Figura 4-5) es muy importante a la hora de asegurar la posibilidad de depuración sobre

el plataforma objetivo. De hecho, en el caso del HSDT100 determinó el OS de desarrollo, ya que el *gdb* de *OCDemon* estaba disponible en *Cygwin+Windows*, entorno en el que se pudo instalar y aplicar las librerías metodológicas propuestas. En la Figura 4-6 se muestra el conexionado de la plataforma HSDT100 al entorno de desarrollo y se señala, entre otras, la conexión JTAG a través del *Wiggler*.

En casos en que la arquitectura es propia, el entorno de desarrollo puede desarrollarse adaptando herramientas de libre distribución **[BPCH04]**. Este ha sido el caso de las herramientas empleadas para la plataforma OpenRISC. En ese caso, se utilizó una plataforma de desarrollo *PC + Linux*, en la que la cadena de herramientas cruzadas probada fue *gcc-2.95.3* y *gcc-3.x* para las versiones 1.3 y 2.0 de *eCos* respectivamente. En ese caso, tanto *binutils*, como *gcc* y *gdb* eran particularizaciones (*ports*) desarrollados por el GIM. En esta ocasión, la carga del programa y el depurado se realizaba a través del puerto serie.

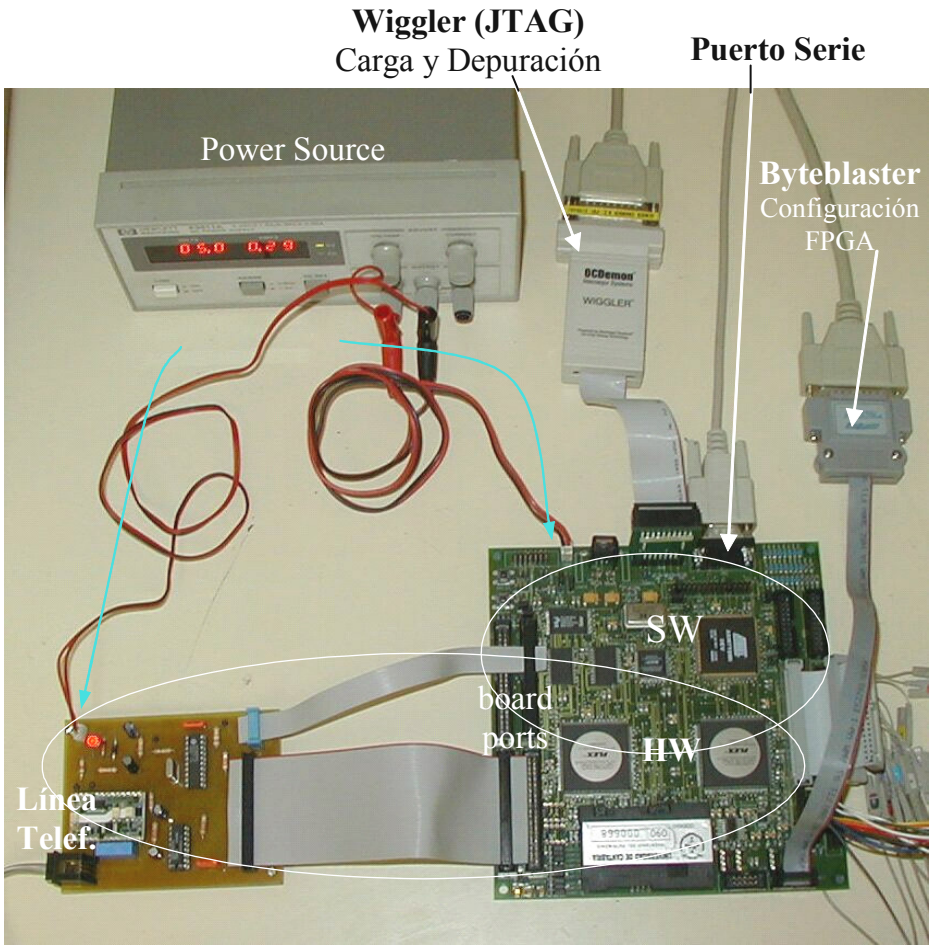



**Figura 4-6. Conexionado de la HSDT100 al entorno de desarrollo.**

En el caso de la EPXA1, el entorno de desarrollo (tanto cruzado como de HW) era Quartus II, disponible tanto para Windows como para Cygwin. Este era un caso, en el que las herramientas cruzadas y la plataforma son suministradas por el mismo proveedor (Altera). Como RTOS para esta plataforma se utilizó *eCos* 2.0. Una característica interesante de esta plataforma era una de sus posibilidades de

configuración. En ella, el binario generado que se carga en *flash* incluye, además del programa de aplicación, los datos de configuración de la lógica programable. De esta forma, la primera acción tras el reset del sistema consiste en la configuración del hardware. Este entorno se probó sólo en Linux, debido a que sólo existían fuentes de *eCos* 2.0 para la plataforma EPXA10, una plataforma superior y similar a la EPXA1. Para utilizar estas fuentes en la EPXA1 se utilizó un parche sólo disponible para Linux. Una vez fijada esta instalación básica, las librerías *HetSC* y *SWGen* se incrustaron en este entorno sin problemas.

En el caso de la CSB536FS de *Freescale*, se empleó un entorno de desarrollo cruzado provisto por un tercero, *Microcross* **[Micr08]**. Este es un caso híbrido, en el que una empresa desarrolladora de software se alía con otra de plataformas y suministran tanto la plataforma como el entorno de desarrollo como un paquete conjunto. La plataforma CSB536FS y su interconexionado con el PC sobre el que se han realizado las pruebas de desarrollo se muestran en la Figura 4-7. Las herramientas de desarrollo provistas por *Microcross* incluyen en este caso, además de las utilidades binarias, el compilador y el depurador; una particularización de *Linux* (*GX-Linux*) para la plataforma. Esto es muy ventajoso en un entorno de desarrollo como este, ya que el RTOS provee muchos de los controladores de la plataforma, que cuenta con diversas y complejas interfaces de entrada/salida. Estas herramientas están provistas tanto en un entorno basado en *Linux* como en un entorno *Cigwin+Windows*.

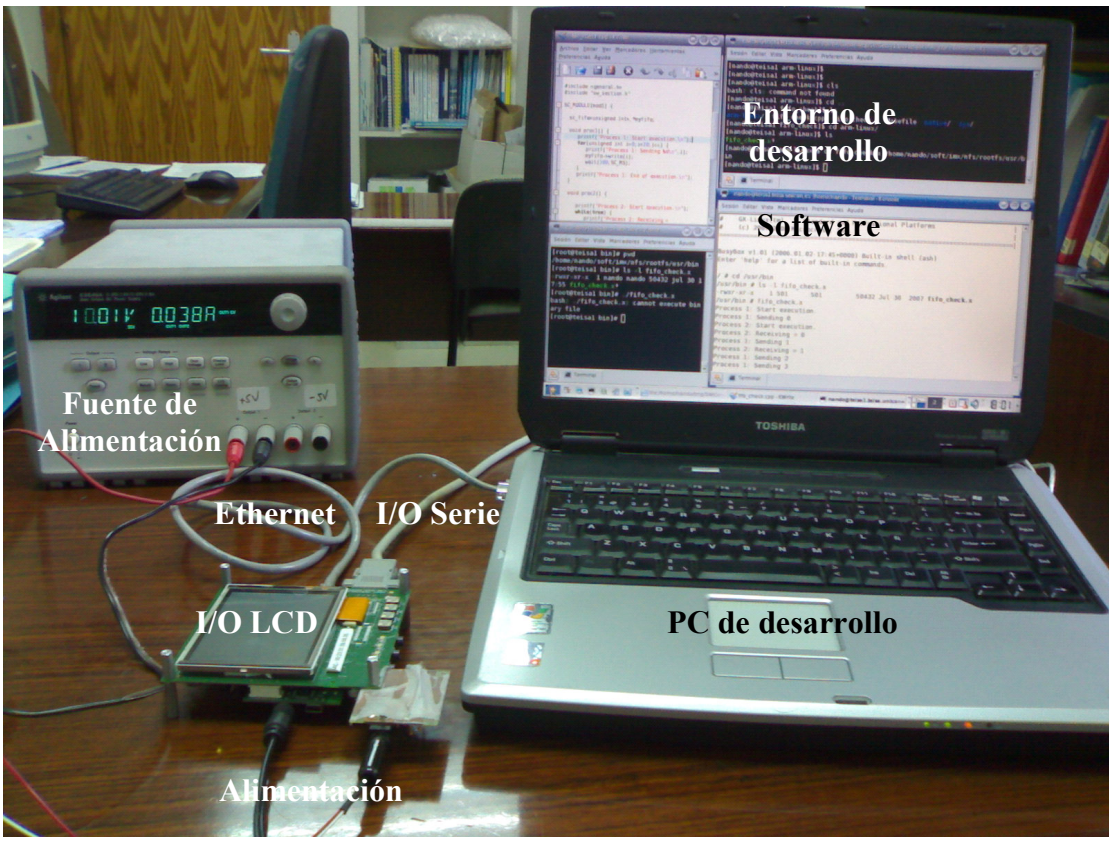



**Figura 4-7. Conexionado de la plataforma de Freescale con la plataforma de desarrollo.**

El RTOS (*GX-Linux*) de esta plataforma y el entorno de desarrollo provisto permitió comprobar el desempeño de la metodología propuesta junto con técnicas de desarrollo más avanzadas. El entorno de desarrollo se esquematiza en la Figura 4-8. Después de arrancar el kernel de *GX-Linux*, la plataforma objetivo montaba en red a través de la interfaz *Ethernet*, un directorio (*/home/nando/soft/imx/nfs/rootfs/usr/bin*) situado en el disco duro de la plataforma de desarrollo (un PC). Para ello, la plataforma objetivo arrancaba un cliente NFS en *GX-Linux*, que utilizaba los servicios de un servidor NFS configurado en el PC. En *Linux*, esta configuración es relativamente sencilla, ya que se trata de editar un fichero y arrancar el servicio. Por otro lado, la entrada/salida básica con la plataforma se realizaba mediante *minicom*, un programa de modo consola, que ejecutando en el PC, permitía lanzar a través del puerto serie comandos *Linux* desde el PC a la plataforma objetivo, para que ésta los interpretara y ejecutara (Figura 4-9d), y ver las respuestas y mensajes de la plataforma (incluidos los del arranque del núcleo de Linux).

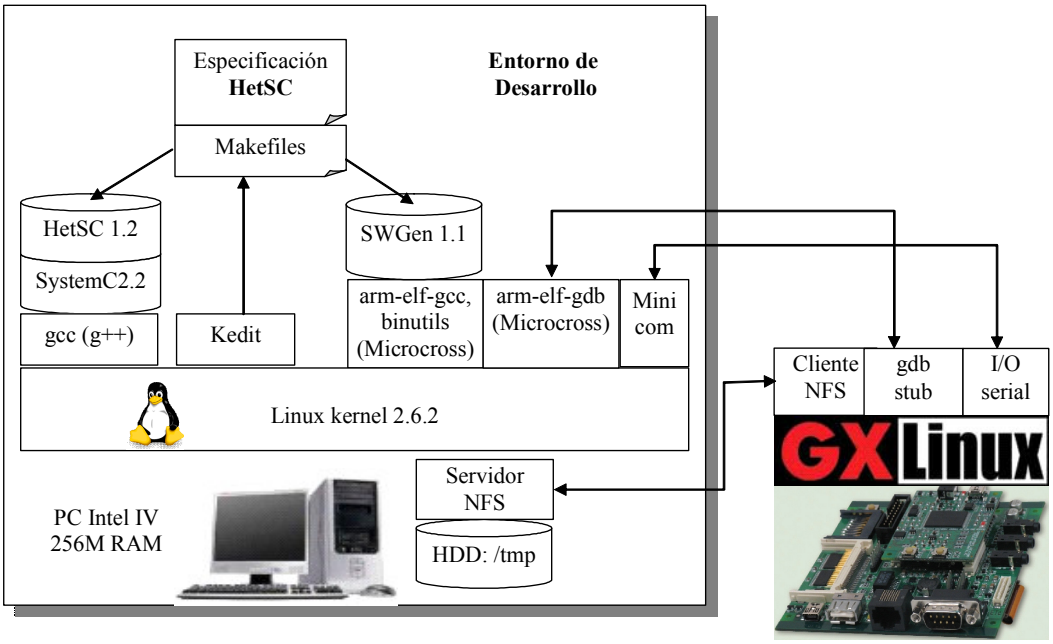



**Figura 4-8. Esquema del entorno de Desarrollo sobre Linux embebido.**

En este entorno, el desarrollo era rápido y cómodo, en especial para la generación y prueba de software. En el PC se editaba la especificación *HetSC* (Figura 4-9a). Esta se compilaba contra la librería *HetSC* y se simulaba. Tras comprobar su funcionamiento en el nivel de sistema, se realizaba la generación de software haciendo la compilación cruzada contra la librería *SWGen*. El binario resultante se copiaba automáticamente a través del *Makefile* al directorio */home/nando/soft/imx/nfs/rootfs/usr/bin* (Figura 4-9b).

Si este binario se trata de ejecutar en el PC, se obtiene un error (Figura 4-9c), ya que realmente contiene código máquina de la plataforma objetivo (de arquitectura ARM), en tanto que el PC de desarrollo tiene una arquitectura Intel. En la plataforma nativa Linux habría que emplear un simulador de plataforma (del que no se disponía) o uno de nivel de instrucciones (ISS) si el código generado no tuviera entrada/salida. Sin embargo, se contaba con la plataforma y la carga y comprobación del binario generado

era muy cómoda. En efecto, el binario se podía lanzar desde la consola *minicom* conectada a la plataforma objetivo (Figura 4-9d), en la que el directorio */home/nando/soft/imx/nfs/rootfs/usr/bin* está montado como */usr/bin*. En la Figura 4-10d se puede observar el resultado de lanzar en binario generado a partir de la especificación de la Figura 4-10a.

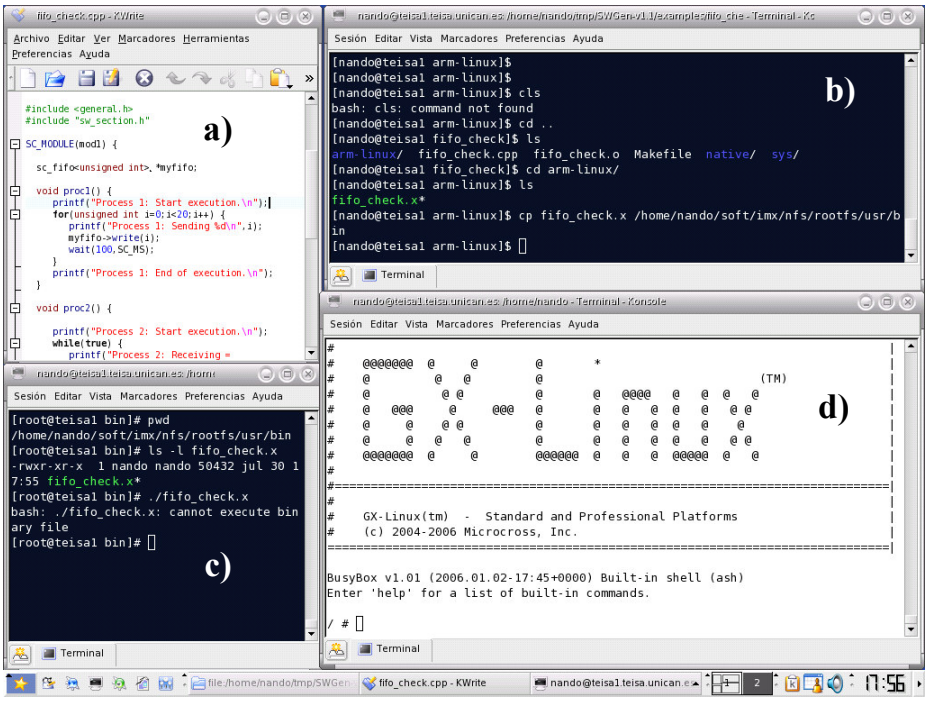



**Figura 4-9. Entorno de desarrollo software para la plataforma con Linux Embebido.**

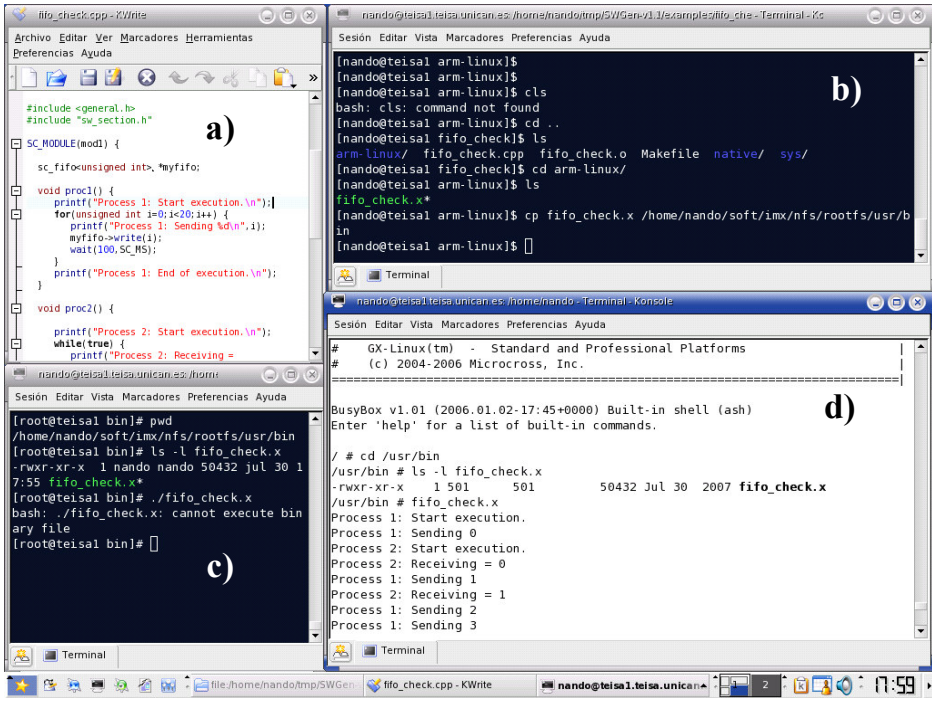



**Figura 4-10. Lanzamiento de la aplicación embebida en el entorno de Linux embebido.**

De este modo, se podía comprobar fácilmente la generación de software en la placa hasta que se decidía que era satisfactoria. En ese momento, se configura el sistema para el trabajo en campo. Es decir, el ejecutable se carga junto con la distribución de GX-Linux en la memoria flash y se configuraba GX-Linux para el arranque automático de esa aplicación tras el arranque del núcleo.

Como se explicó, otra herramienta importante en estos entornos de desarrollo son las herramientas de configuración del RTOS. En la Figura 4-11 se muestra una captura de la herramienta de configuración gráfica de *eCos*. Esta herramienta ha sido ampliamente utilizada, ya que en las plataformas en las que se ha empleado, la huella era un factor más limitante. En este sentido, fue preciso buscar configuraciones manuales ajustadas, respetando los servicios mínimos requeridos por la librería *SWGen*.

El kernel de Linux embebido, y específicamente el de *GX-Linux* también es configurable. Existen diversas herramientas gráficas para configurar el núcleo de Linux embebido. No obstante, en el caso de la plataforma de *Freescale*, con una configuración estándar de *GX-Linux* se satisfacían todas las necesidades de *SWGen*, quedando aún más del 80% de la memoria flash libre.

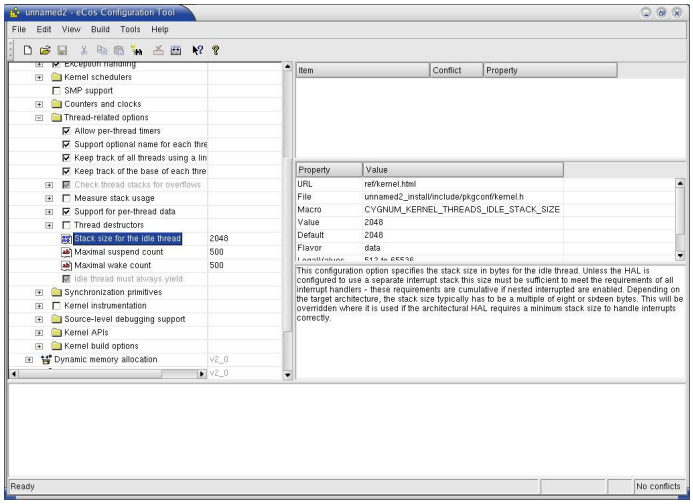



**Figura 4-11. Herramienta de configuración gráfica de *eCos*.**

El diseño y configuración del hardware es lo que normalmente ha exigido la herramienta más costosa en el entorno. En este caso, para la HSDT100, se ha usado *MaxPlus 10* para configurar las FPGAs FLEX10K100 de Altera. En el caso de la *Excalibur EPXA1*, se ha usado *Quartus II,* en tanto que en el caso de la placa de DS2 se empleó *ISE* para la síntesis y configuración de las *Virtex II*. En este último caso, la configuración de las mismas se realizó con un programador de FPGAs para *Linux* realizado por el GIM.

En definitiva, se ha comprobado que las metodologías y librerías desarrolladas se pueden integrar y usar, al menos, para una muestra representativa de las diferentes plataformas de implementación y entornos de diseño de sistemas embebidos. En las siguientes secciones se repasarán datos demostrativos obtenidos con ejemplos concretos.

## 4.3 Ejemplos

Las distribuciones de *HetSC* y *SWGen* vienen provistas de pequeños ejemplos que permiten al usuario ejercitar aspectos específicos de las metodologías. En las siguientes secciones se proveen y razonan datos de algunos ejemplos desarrollados para la medida y comparativa de las capacidades de estas metodologías.

### 4.3.1 Resultados de especificación: Ejemplo FIR

Mediante doce versiones diferentes del ejemplo del filtro FIR que se provee con la distribución de SystemC, se han obtenido datos que permiten comparar el beneficio de utilización de unos MoCs frente a otros en un entorno metodológico basado en SystemC.

La Figura 4-12 provee representaciones HetSC de esos ejemplo FIR. En todas existe un bloque que genera los estímulos, escribiendo secuencialmente las 500.000 muestras, que son leídas por el bloque FIR, que las procesa y transfiere al bloque de salida, que las lee sin realizar ningún procesado. Todas las especificaciones eran funcionalmente equivalentes. Es decir, las simulaciones producían las mismas muestras de salida para las 500.000 muestras de entrada. Asimismo, todos emplearon el mismo tipo de dato (entero) para los datos transferidos y procesados.

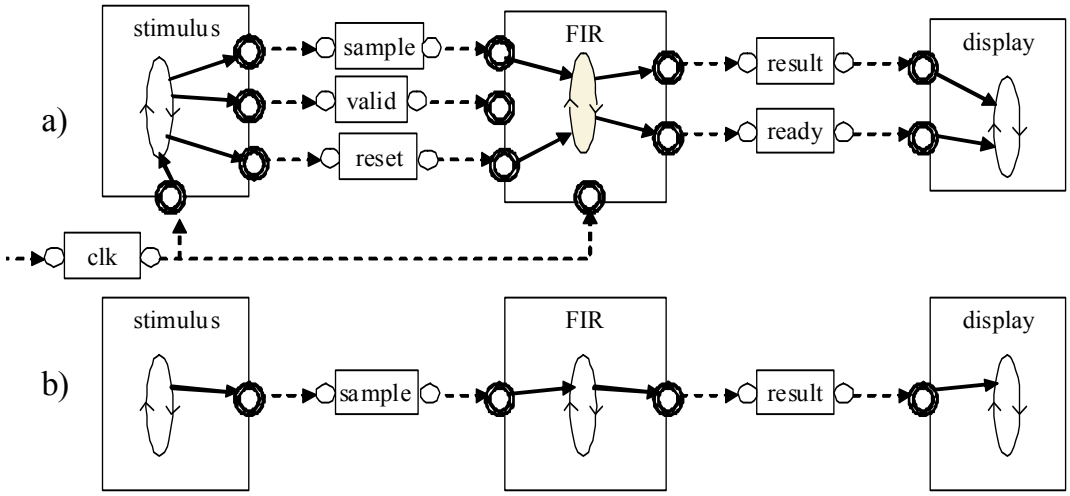



**Figura 4-12. Representaciones HetSC del ejemplo FIR.**

La Figura 4-12a representa las dos primeras versiones:

1. Versión RTL.
2. Versión de comportamiento o algorítmica.

Estas dos versiones representan ejemplos del MoC síncrono de reloj. Son básicamente los ejemplos de la distribución de SystemC con ligeras modificaciones introducidas para hacer más justa la comparación con las siguientes versiones. Por ejemplo, se han sustituido los SC_METHODs por SC_THREADs (el tipo de proceso empleado en las siguientes 7 versiones). Se comprobó que el uso de SC_THREAD suponía un aumento en torno al 15% del tiempo de simulación respecto a las versiones

con SC_METHOD. En estos modelos es preciso usar dos canales de tipo señal booleanas (*sc_signal<bool>*) utilizados como señales de protocolo (*valid* y *ready*), además de las dos instancias de canal señal para transferencia de datos (*sc_signal<int>*) (*sample* y *result*). Se puede observar también la necesidad de señales de reloj de reset. Por tanto, se precisaban 6 instancias de canal de tipo señal de SystemC.

La representación de la Figura 4-12b se corresponde con las siete versiones siguientes que se han probado. Estas versiones, además de funcionalmente equivalentes, son más abstractas, bajo MoCs atemporales, lo que significaba una disminución del código necesario para realizar la especificación. Como muestra la Figura 4-12, la estructura de la especificación se simplificaba, requiriendo sólo dos instancias de canal para realizar la comunicación de los módulos del sistema. Así, en cada uno de los 7 ejemplos, las 6 instancias de canal *sc_signal*, se sustituyeron por instancias de diferentes tipos de canal propios de los diferentes MoC atemporales. Las versiones realizadas son:

3. Versión KPN, usando canales *uc_inf_fifo*.
4. Versión BKPN, usando canales *uc_fifo* de tamaño 1.
5. Versión BKPN, usando canales *uc_fifo* de tamaño 100.
6. Versión BKPN, usando canales *uc_fifo* de tamaño 10.000.
7. Versión SDF, usando canales *uc_arc* de tasa de lectura y escritura 1.
8. Versión SDF, usando canales *uc_arc* de tasa de lectura y escritura 100.
9. Versión SDF, usando canales *uc_arc* de tasa de lectura y escritura 10.000.

De esta forma estas versiones *HetSC* se escriben más rápido y son más simples. Además de no tener que instanciar canales de protocolo, tampoco es preciso, por tanto, el cómputo de protocolo de comunicación asociado en los procesos. Como resultado, incluso en un ejemplo tan pequeño, la cantidad de código fuente necesaria se redujo alrededor de un 30% (por ejemplo, 500 líneas la versión BKPN por 700 de la RTL).

El décimo ejemplo consistió en reproducir el ejemplo FIR mediante la librería SystemC-H **[PaSh04] [PaSh05]**. En el undécimo y el duodécimo ejemplo se especificó el filtro FIR en SystemC-AMS para tasas de escritura y lectura iguales, de 1 y 100 respectivamente. Las doce versiones se han simulado y se ha obtenido el tiempo de simulación total. Los resultados se muestran en la Figura 4-13. La plataforma de simulación tenía las siguientes características: Intel IV, RAM de 768Mbytes y Linux 2.6.3 kernel. Para los ejemplos SystemC estándar y *HetSC* se ha usado SystemC v2.2 y *HetSC v*1.2 con gcc-3.3.0. El ejemplo SystemC-H se ha realizado con SystemC 2.0.1. El ejemplo SystemC-AMS con SystemC 2.2 y gcc 4.1.0.

Como se puede observar en la Figura 4-13, el tiempo de simulación se reduce entre un 70% y un 90%, dependiendo del modelo computación *HetSC*. Una de las razones es que, al evitar las señales de protocolo (menor número de instancias de canal), se necesita un menor trasiego de datos.

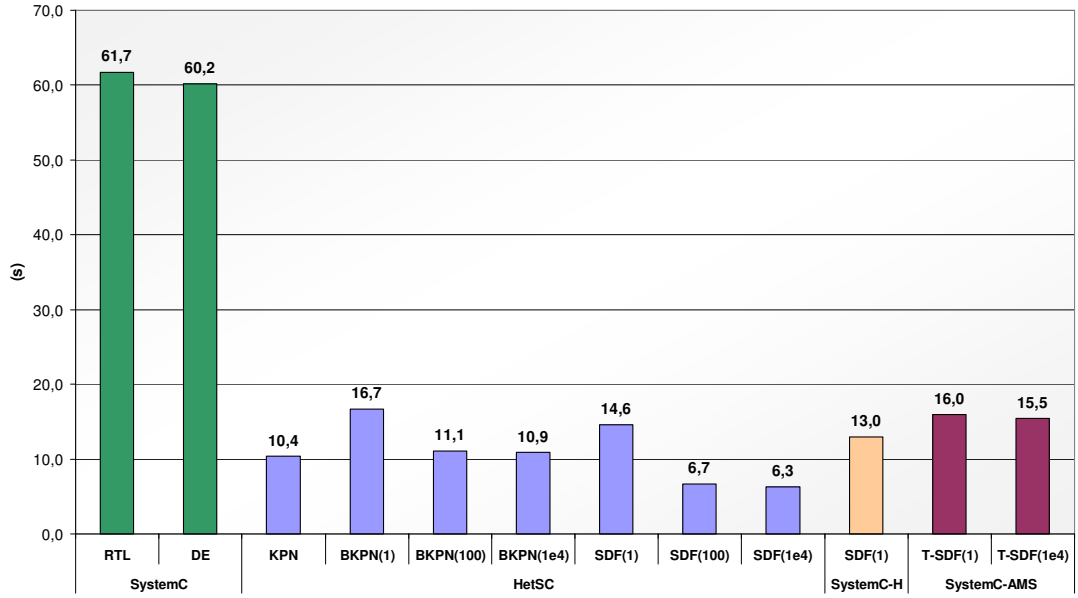



**Figura 4-13. Tiempos de simulación del ejemplo FIR.**

No obstante, un estudio más detallado reveló que uno de los principales factores que afectan al tiempo de simulación en estos ejemplos es el número de ciclos delta que realiza internamente el simulador de eventos discretos de SystemC. Esto se muestra en la Figura 4-14. La especificación PN(1) presenta un máximo de deltas para el tamaño mínimo de fifo, lo que provoca un mayor número de cambios de contexto entre los 3 procesos del sistema, llegando a la ejecución de 1E7 deltas de simulación. A medida que se aumenta el tamaño de la memoria intermedia del canal (a través del tamaño de los canales fifo o de las tasas de los canales arco) el número de deltas se reduce de forma exponencial. Esa exponencial es cuadrática debido a la topología del ejemplo, en la que los canales actúan como separadores de las etapas de un cómputo segmentado, siendo la memoria intermedia de cada canal lo que determina el número de "disparos" o cómputos realizados en cada proceso antes de su bloqueo.

Se observó también que cuando se reduce hasta cierto punto el número de deltas, el factor limitante en velocidad de simulación pasa a ser el peso del cómputo de los procesos. Por ejemplo, para un tamaño de fifo 10.000, el tiempo de simulación es muy cercano al obtenido en el caso KPN. En cualquier caso, las versiones síncronas de reloj provocan aún más deltas de simulación (1,8E8) que cualquiera de los modelos abstractos, demostrando como mínimo que estos deltas son un factor influyente en el tiempo de simulación, haciendo que las versiones más abstractas sean al menos un 70% más rápidas.

Los resultados de las versiones *HetSC,* incluida la aproximación dinámica de SDF, son equiparables o mejores que el SDF estático de SystemC-H y el T-SDF estático de SystemC-AMS, que cuentan con su propio solucionador. Esto no es sorprendente si se tiene en cuenta la optimización conseguida gracias a la planificación estática no es tan importante como el efecto de reducción de ciclos delta de simulación. Esto es extensible al menos a las especificaciones en las que se cumple la suposición de especificación de grano grueso **[LeMe87]**, como es el caso de este ejemplo FIR.

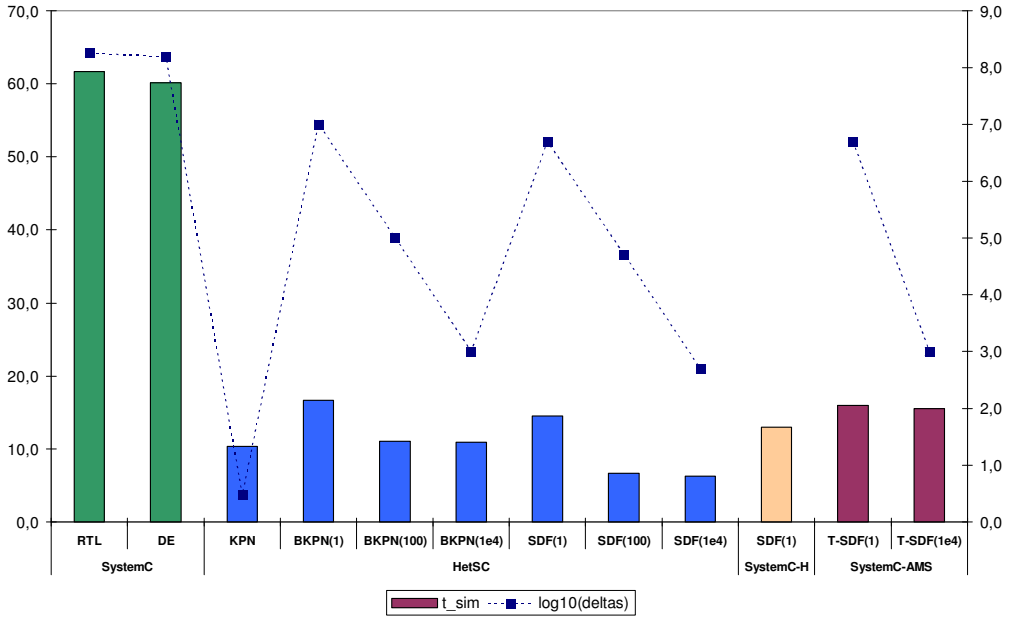



**Figura 4-14. Influencia de los deltas de simulación.**

Estos datos muestran que *HetSC* puede conseguir prestaciones en velocidad de simulación para MoCs abstractos tan buenas como otras metodologías que proveen *solucionadores* específicos. La consideración de la ley de Amdahl en especificaciones heterogéneas en las que los bloques menos abstractos tienen un peso de cómputo importante respecto al cómputo de los menos abstractos, quita incluso más relevancia a las optimizaciones de esta índole. Aún más, se ha medido la influencia de factores que pueden ser obviados cuando se trata de acelerar la simulación y que, por otro lado, son usuales en especificaciones de nivel de sistema basadas en C como:

- El uso de una u otra versión de compilador y optimizaciones. Por ejemplo, en el ejemplo FIR, la diferencia entre usar la versión 4.1.0 de gcc (en lugar de la 3-3.2) suponía una ganancia entre el 20%-40% para simulaciones de alto nivel y entre el 5-15% para distintas versiones RTL.
- Cuando se introducían redirecciones, volcados a fichero o pantalla la simulación se alargaba hasta dos órdenes de magnitud.

### 4.3.2 Resultados de Generación de Software: ABS y EFRVocoder

La librería *SWGen* tiene una serie de ejemplos que han sido comprobados sobre las plataformas objetivo de la sección 4.1 y en los entornos mostrados en la sección 4.2. Esa comprobación ha consistido en realizar primero una compilación contra la librería SystemC y la realización de una simulación de nivel de sistema en la plataforma de desarrollo. Posteriormente se realizaba una generación de software con una compilación cruzada usando la librería *SWGen*, se cargaba el binario generado y se ejecutaba en la plataforma objetivo, observando que la funcionalidad se mantenía.

Mediante otros ejemplos, además de una verificación funcional, se ha realizado un análisis de otros parámetros, como la huella, el soporte requerido al RTOS o la facilidad de adecuación de *SWGen* a un RTOS embebido. Esta estimación se hizo en un ejemplo

consistente en un sistema de frenado antibloqueo (ABS) **[FHSV03]**. Para ello, se realizó primeramente una particularización (o *port*) de *SWGen* a *eCos*. En la Figura 4-15 se esquematiza el número de primitivas del RTOS que se necesitaron para tener un soporte básico de *SWGen* (concurrencia, temporización y algunas primitivas de comunicación). Como se puede comprobar es muy reducido y, por tanto, el coste de soporte de esta API también (aproximadamente 2 personas·día).

| | Gestión de hilos | Sincronización | Gestión de Interrupciones |
|---|---|---|---|
| **Funciones eCos** | cyg_thread_create<br>cyg_thread_resume<br>cyg_user_start<br>cyg_thread_delay | cyg_flag_mask_bits<br>cyg_flag_set_bits<br>cyg_flag_wait | cyg_interrupt_create<br>cyg_interrupt_attach<br>cyg_interrupt_acknowledge<br>cyg_interrupt_unmask |

**Figura 4-15. Funciones del API-C de *eCos* utilizadas por *SWGen*.**

Seguidamente se analizó la huella del sistema, resultando los datos de la Figura 4-16. La figura muestra el bajo costo adicional de la librería *SWGen* en tamaño de código respecto a la huella del sistema.

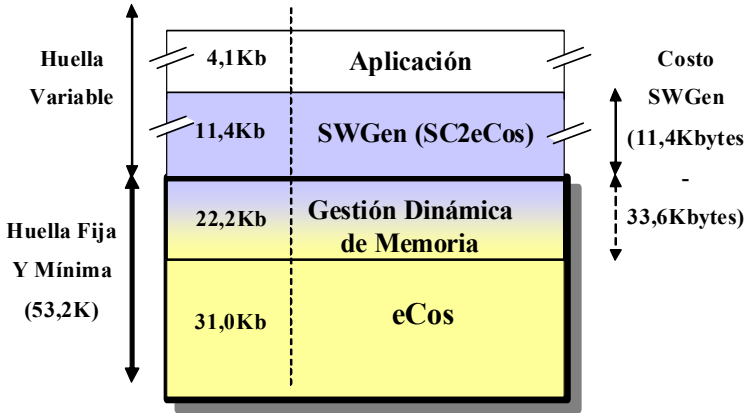



**Figura 4-16. Huella del ejemplo del ABS implementado en la plataforma HSDT100.**

El código de *SWGen* para esta especificación consumía 11.4Kbytes, menos de un 17% del tamaño total de la huella (68,7Kbytes). Esta penalización es pequeña si se considera que se trataba de una aplicación pequeña y con poco valor añadido. El tamaño requerido por el código con valor añadido, es decir, el que no es debido ni a la librería *SWGen*, ni al RTOS, ni a librerías estándar, era de sólo 4,1Kbytes. Los 11,4Kbytes representaban un aumento de la huella mínima de apenas el 21% (la huella mínima sin *SWGen* era de 53,2Kbytes) y representaba sólo el 1,1% del tamaño de la *flash* de la HSDT100 (1Mbyte).

Se realizó un análisis más profundo para tratar de esclarecer si el mapeo a C++, en lugar de a C, producía un impacto significativo en la huella. Este análisis reveló que parte de esa huella (31.0Kbytes) era la mínima requerida por *eCos* para una implementación desde C que no hiciera uso alguno de los recursos de gestión dinámica de memoria (*new, calloc, malloc, etc*). Si la aplicación C hacía gestión dinámica de memoria, la huella crecía 22,2Kbytes, hasta los 53,2Kbytes. Un aplicación C++ e, indefectiblemente, la librería *SWGen*, hace uso de esos servicios, con lo cual fuerza el costo adicional de 22.2Kbytes. Por eso, y porque la gestión dinámica de memoria es muy frecuente, incluso en aplicaciones C, se ha considerado esa componente parte de la

huella mínima en la Figura 4-16. Por tanto, el mapeo de *SWGen* a C++ no supone costo adicional de tamaño respecto a una generación de C desde una aplicación que realice gestión dinámica de memoria (llamando al menos una vez a *malloc*, etc).

Se comprobó también que la componente de tamaño debido al código de de *SWGen* es reducida en un ejemplo mayor, específicamente, en un vocoder GSM, cuya especificación se explica con más detalle en la sección 4.3.3. Este ejemplo se implementó en una plataforma objetivo y entorno de desarrollo distinto al del ejemplo ABS: en la plataforma DS2 con OpenRISC 1500 y con herramientas cruzadas de GNU portadas por el GIM-UC y RTOS *eCos*. Estos datos se muestran en la Figura 4-17.

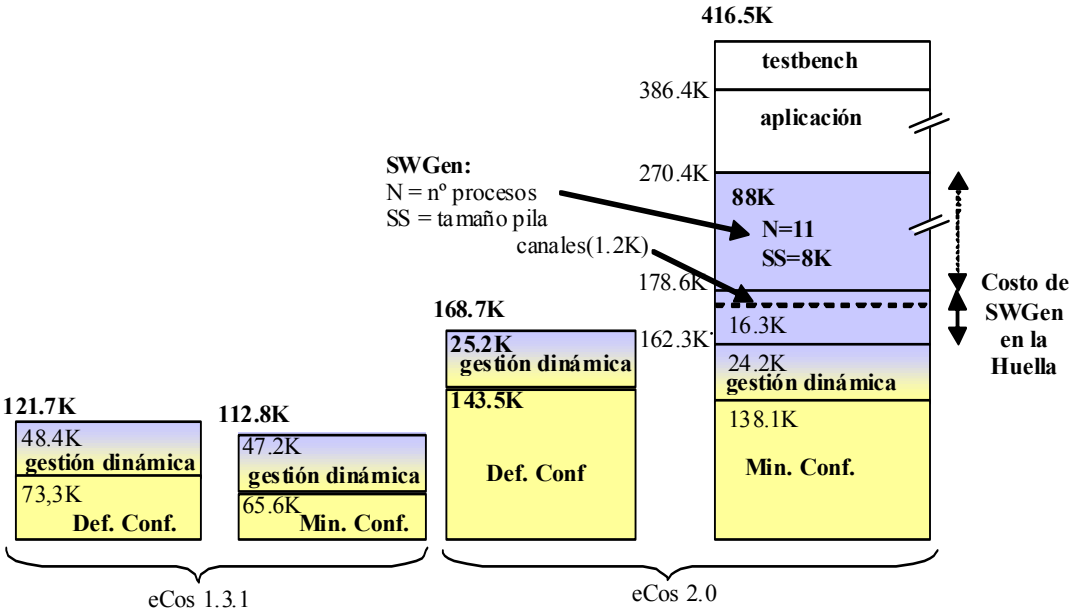



**Figura 4-17. Distintas huellas obtenidas sobre la plataforma OpenRISC.**

En la Figura 4-17 se representan las huellas mínimas obtenidas para *eCos*-1.3.1 y *eCos*-2.0. Para estas combinaciones de RTOS se precisaba usar distintas cadenas de desarrollo cruzadas (básicamente, gcc-2.95.3 y gcc-3.3.0 respectivamente). Por cada versión de RTOS, se obtuvieron datos para una configuración por defecto y para una configuración mínima. Para el caso de *eCos* 2.0 con una configuración mínima, se hizo un análisis similar al de la Figura 4-16 (parte derecha de la Figura 4-17), distinguiendo además en el tamaño de código de aplicación la parte debida al código de la especificación del sistema de la de la mesa de pruebas (que se implementó en software y se incluyó en la plataforma también).

La componente de *SWGen* varía en función de las invocaciones de servicios realizadas por el código de especificación. Este crecimiento, en general, no es lineal, ya que según el código de especificación va creciendo, existe más probabilidad de que el código de librería *SWGen* invocado ya esté presente. Tiene un costo mínimo fijo (16,3Kbytes), del cual, una pequeña parte (1,2Kbytes) corresponde a la implementación de los canales empleados (esta generación se probó para una versión del Vocoder con canales fifo). Este costo representa aproximadamente un 10% de la huella sin *SWGen* (162,3Kbytes).

Otra componente de la huella es variable y proporcional a dos parámetros: el tamaño de la pila asignada a cada proceso (SS) y el número de procesos de la especificación (N). En la versión actual de *SWGen,* N se ajusta al número de procesos de la especificación más uno, en tanto que la librería permite configurar un valor de SS por defecto para todos los procesos. También se puede fijar desde la propia especificación por cada proceso, tal y como se explicó en la sección 3.5.9. A este respecto, la metodología de generación ha de fijar un tamaño de pila ni muy pesimista ni insuficiente por cada proceso. Esta componente no es atribuible a *SWGen,* ya que sin usar *SWGen*, una aplicación C/C++ usando los servicios de concurrencia del RTOS requeriría igualmente reservar tamaño de pila por cada proceso. No obstante, en la Figura 4-17 se ha querido reseñar (mediante la flecha de puntos) que un uso incorrecto de la librería de generación en este aspecto puede resultar en un costo adicional.

Se pudieron hacer otras observaciones acerca de la huella. En primer lugar, la huella mínima en el caso de OpenRISC es mayor que la de la arquitectura ARM (más del doble). En efecto, ARM es una arquitectura muy eficiente en tamaño de código. Además, existía una diferencia de tamaños también notable entre los datos obtenidos con *eCos*-1.3.1 y *eCos*-2.0. Esta diferencia, no obstante, más que debida a la particularización del RTOS para la plataforma (*port*), era debida al compilador. En el caso del *eCos*-2.0, se requería gcc-3.3.0, cuya particularización para OpenRISC no estaba tan refinada como la de gcc2.95.2 (la usada para *eCos-1.3.1*). Esto revela que algunos aspectos, tales como la arquitectura software seleccionada y la calidad de la particularización del compilador pueden ser tan o más críticos a la hora de reducir la huella del sistema, que la propia librería *SWGen* u otros factores, inicialmente más importantes, como la configuración del RTOS. De hecho, entre una configuración por defecto y una mínima apenas había una diferencia de apenar unos Kilobytes.

En conclusión, la librería *SWGen* es fácilmente portable a distintas APIs, y supone un costo adicional en la huella reducido (entre el 10% y 20%, dependiendo de la arquitectura software) en aplicaciones pequeñas que usen memoria dinámica. En el tamaño final de la aplicación, otros factores, como la arquitectura software objetivo, la calidad del compilador cruzado, el número de procesos de la especificación y los tamaños de pila asignados por proceso, requieren más consideración.

### 4.3.3 Especificación y generación de un EFR-Vocoder

Para poner en práctica las metodologías *HetSC* y *SWGen* con un ejemplo real y significativo, se ha realizado la especificación de un vocoder EFR, cumpliendo la norma GSM 06.60 o ETSI/EN 301 245 **[EN245]**. Las fuentes y la documentación de este ejemplo están disponibles en **[HSC08][VoUG06]**. Aquí se hará un breve resumen del sistema y de los resultados. Este sistema es un codificador/decodificador (o codec) de voz. El codificador toma como entrada registros de audio consistentes en 160 muestras de 13 bits a 8Kmuestras/seg (16,64Mbps). Cada uno de esos registros es codificado en un chorro de 244 bits, lo que produce una tasa de 1,952Mbps y un factor de compresión de aproximadamente 8,5. El decodificador realiza la función inversa, sintetizando a la salida registros de 160 muestras a partir de chorros de 244bits.

El estándar **[EN244]** provee un código de referencia ANSI-C del EFR vocoder de unas 15.000 líneas de código fuente. Este código se ha utilizado como modelo de oro

funcional con precisión de bit. Además, en **[EN250]** se provee también un conjunto de ficheros de banco de prueba para la verificación del sistema. Entre estos ficheros se encuentran los registros de voz de entrada y de salida del codec, así como los de etapas intermedias (de salida del codificador y entrada del decodificador). Estos ficheros ponen a prueba el sistema para tonos de voz de hombre y de mujer, en distintas condiciones de ruidos de fondo, etc. Algunos tests prueban también características específicas del codec, como su comportamiento en modo discontinuo. En ese modo, el codificador detecta si no hay voz en la línea, en cuyo caso, produce una salida que emula el ruido de la entrada, pero empleando una tasa binaria reducida. Esto evita un efecto indeseado de ruido pulsado (on/off) en recepción.

En un primer paso se codificó el *sc_EFRVocoder*, es decir, la especificación HetSC. Esta especificación se realizó como una adaptación funcionalmente equivalente del código de referencia ANSI-C. En la Figura 4-18 se muestra un esquema de la distribución de las fuentes así como de los ficheros de test y de los ejecutables ANSI-C precompilados.

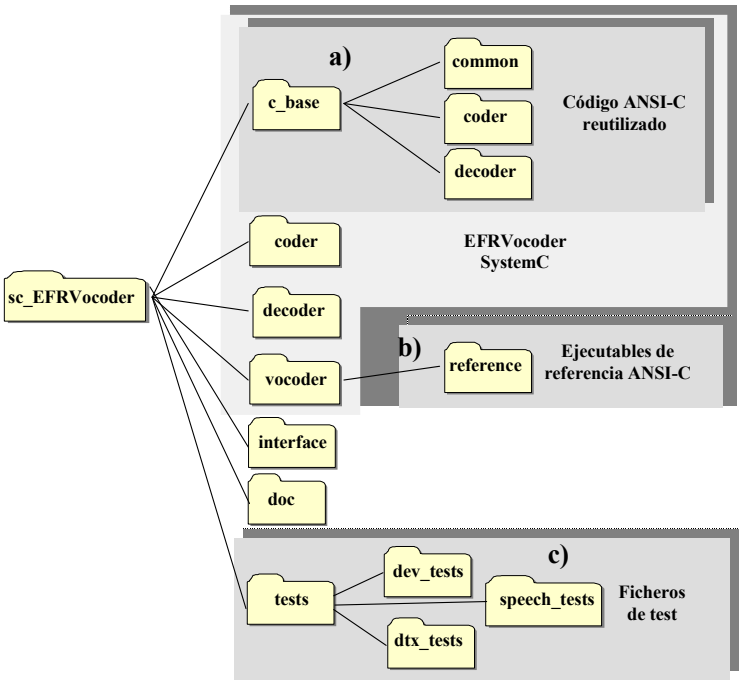



**Figura 4-18. Estructura de las fuentes del EFR Vocoder en SystemC.**

Las zonas sombreadas oscuras muestran código fuente (Figura 4-18a) y ficheros ejecutables (Figura 4-18b) y de test (Figura 4-18c) reutilizados. De este modo, un resultado interesante es que la generación de esta especificación se hizo reutilizando el 83,3% del código de la especificación ANSI-C de referencia (12.500 líneas).

La especificación *sc_EFRVocoder*, añade, respecto al modelo ANSI-C, una estructura de concurrencia y una jerarquía modular. Los dos niveles superiores de la jerarquía modular se muestran en la Figura 4-19.

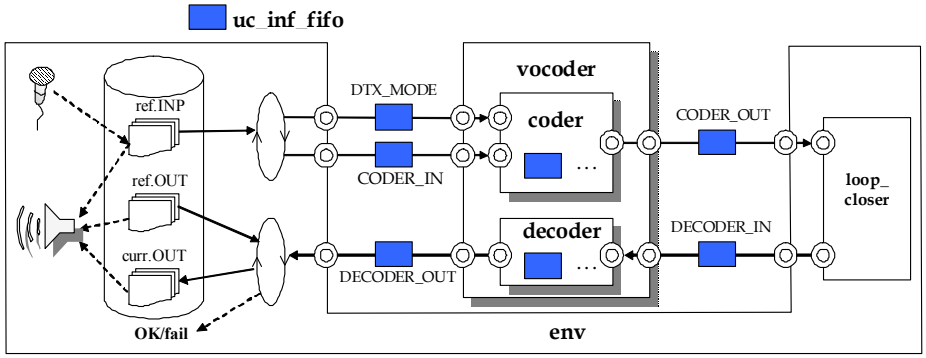



**Figura 4-19. Diagrama de los dos niveles de jerarquía superior del EFR Vocoder y su entorno.**

El nivel superior incluye los módulos de entorno y de sistema. El módulo de sistema está compuesto de dos módulos, el *EFRcoder* y el *EFRdecoder*. Ambos, codificador y decodificador encierran un nuevo nivel de jerarquía. De este modo, al final, el sistema comprende un total de 14 módulos hoja y 15 procesos. Estos procesos son SC_THREADs comunicados a través de canales fifo de tamaño no acotado (*uc_inf_fifo*) y, por tanto, constituyendo una red de Kahn. De esta forma, la versión inicial del sistema se centra en una primera partición que considera un conjunto mínimo de condiciones de interbloqueo. Esto permitió realizar la partición concurrente más fácilmente y centrarse en la funcionalidad, es decir, en el cómputo de los procesos, antes que en los problemas de bloqueos y dimensionamiento de recursos de comunicación. Esto se postergó, como se verá, para versiones posteriores, produciendo un refinamiento al MoCs BKPN y, posteriormente una especificación heterogénea.

El entorno fue escrito siguiendo el mismo MoC, aunque tomando ventaja de la posibilidad de relajar el estilo de codificación en algunos aspectos. Por ejemplo, el módulo de entorno lee los datos de entrada de fichero (*ref.INP*) y vuelca los datos de salida del sistema en fichero también (*curr.OUT*), para compararlo con el fichero de salida de referencia (*ref.OUT*). Tanto los ficheros de entrada como los de salida de referencia son los provistos en **[EN250]**. De esta forma, el ejecutable puede reportar si el test ha pasado correctamente. Para el *sc_EFRVocoder*, se han ejecutado los 28 tests propuestos por **[EN250]**, para realizar una verificación funcional de precisión de bit, resultando todos ellos correctos. Por verificación funcional de nivel de bit, se entiende que existía una coincidencia bit a bit entre las salidas obtenidas (*curr.OUT*) y las esperadas (*ref.OUT*) para los mismos registros de entrada (*ref.INP*). Adicionalmente, la cobertura de esta verificación se mejoró aplicando la técnica propuesta en **[HeVB06]**, basada en la extensión del kernel de SystemC que implementa la aleatorización reproducible de la planificación. Para ello se generó también un programa de barrido y unos ficheros de configuración para automatizar la realización de simulaciones y los ficheros de test para los que se realizaba. De esta manera se realizaron hasta 1000 ejecuciones de cada test. Más detalles se pueden encontrar en **[HeVB06] [PRS06]**.

Se especificó también un entorno interactivo. Este entorno aprovecha la flexibilidad que da *HetSC* en el uso de SystemC para la generación de entornos. Este entorno realiza llamadas al sistema operativo anfitrión para grabar mediante la tarjeta de sonido un registro de voz y volcarlo a un fichero *(ref.INP*). Este fichero se pasa posteriormente al *sc_EFRVocoder*, que lo codifica y decodifica, produciendo un

fichero *curr.OUT*. Asimismo, el entorno también pasa el fichero *ref.INP* al modelo ejecutable ANSI-C, que produce la salida de referencia (*ref.OUT*). Finalmente, el módulo de entorno analiza la coincidencia de *curr.OUT* y *ref.OUT* con precisión de bit y además reproduce estos registros mediante la tarjeta de sonido del PC para una valoración subjetiva. Resumiendo, el *sc_EFRVocoder* ha sido verificado automáticamente y con precisión de bit mediante dos tipos de simulaciones, una basada en patrones estándar enfocados en características específicas de la funcionalidad y los rangos típicos de funcionamiento, y otra que realiza un test interactivo y subjetivo.

Para los 28 tests realizados, ningún tiempo de ejecución de la especificación *HetSC* rebasó en más de un 9% el tiempo que tomaba su contrapartida ANSI-C. El valor medio de este incremento era del 4,3%. También se midió el impacto de los chequeadores de reglas. Para los 28 tests, la activación del chequeo de reglas para cada MoC supuso un incremento de menos del 10% en el tiempo de simulación.

La especificación completa del *sc_EFRVocoder* ocupa alrededor de 18.400 líneas. Alrededor de 2.000 líneas corresponden al código del entorno, que también incluía mesas de pruebas para los submódulos del codificador y decodificador. Por lo tanto, el código de adaptación SystemC ascendió a alrededor de 3.900 líneas, aproximadamente un 21% de las fuentes del *sc_EFRVocoder*.

Después de realizar la versión inicial KPN, se han realizado experimentos de refinado del ejemplo hacia otros MoCs y de mezcla de distintos modelos en la misma especificación. El primer experimento ha sido un refinado a modelo BKPN. Para estimar los tamaños de fifo adecuados, se ha utilizado una de los reportes del canal *uc_inf_fifo*. Este canal reporta, por cada una de sus instancias, cual ha sido el tamaño máximo de unidades de datos presente en el canal. De este modo, por cada instancia, se ha cogido el máximo teniendo en cuenta los 28 tests. Para que estas dimensiones sean más realistas se ha usado la única información temporal de entorno disponible, incluyendo un retardo de 125µs entre la inyección de cada muestra. Una vez obtenidas esas cifras, se han sustituido las instancias de fifo infinita por instancias *uc_fifo*, produciendo un modelo BKPN puro. Los 28 tests ejecutaron otra vez, con la especificación bajo el MoC BKPN sin ningún problema de interbloqueo. Una ventaja de esta especificación frente a la KPN es que, siendo realizable, mantiene la funcionalidad incluso si la temporización de salida no cumple los requisitos. Si el sistema vocoder es muy lento, no se perdería el registro de voz íntegro, aunque éste estaría retardado.

Seguidamente se sustituyeron los canales *uc_fifo*, por *sc_fifo*, una vez que mediante aquellas instancias quedó verificado que no había varios escritores, lectores u otras posibles violaciones del modelo. En este estadio, la especificación era SystemC estándar.

Durante ese refinado, se probó con éxito una versión PN-KPN. Es decir, primero se refinaron los canales del codificador, llegando a una versión intermedia en la que los canales del codificador eran *uc_fifo*, en tanto que los del decodificador *uc_inf_fifo*.

Para continuar con estos experimentos de refinado en los que la heterogeneidad vertical y horizontal se ponen de manifiesto, se decidió refinar dos módulos de salida del coder: el módulo serializador y del módulo *sid_encoder* (codificador de trama de silencios). Inicialmente, estos módulos eran KPN y tras el refinado quedan bajo el MoC

síncrono de reloj, de forma, que quedan preparados para un flujo de implementación hardware. Como se refina solo una parte, esto da lugar a una heterogeneidad horizontal, ya que en la especificación resultante aparecen dos partes bajo dos MoCs distintos (KPN y CS) que hay conectar. Para esa conexión se utilizó los canales frontera *uc_inf_fifo_signal* y *uc_signal_inf_fifo*. Como se muestra en la Figura 4-20, estos canales se encuentran en el tercer nivel de jerarquía, en el que se compone el módulo codificador. En la Figura 4-20, se puede observar también cómo la comunicación entre los dos módulos síncronos de reloj se realiza a través del canal *uc_sigclocked_fifo*. Este canal provee una versión refinada de canal FIFO, con semántica bloqueante en escritura y lectura, pero a través de interfaces de señal y usando protocolos de nivel de ciclo, pero no es un canal frontera, ya que conecta partes bajo el mismo MoC.

El último experimento en especificación fue relativo a la sincronización en banda del coder. Tanto coder como decoder pueden inicializar su estado a través de tramas de entrada especiales. Internamente, la inicialización es decidida por un submódulo que monitoriza cada trama de entrada. Una vez decidida la inicialización, esto conlleva la inicialización de varios submódulos (lo que supone la inicialización de muchos de sus datos internos). Por tanto, la salida del módulo que decide la inicialización en banda, debe estar sincronizada para cada trama con el inicio del cómputo de cada uno de los submódulos que deben inicializarse. En las versiones KPN y BKPN, estos datos de inicialización se transferían mediante canales fifo con bloqueo (*uc_inf_fifo* o *uc_fifo*). Cada unidad de datos transferida en una instancia determinaba si el cómputo, asociado a una trama y realizado por el submódulo que lo recibe, conlleva una inicialización previa o no. En una versión posterior se sustituyeron estos canales por canales rendez-vous de *HetSC*, lo cual parece más natural de cara a una concepción inicial de la especificación. La especificación ejecutaba sin problemas para los 28 tests.

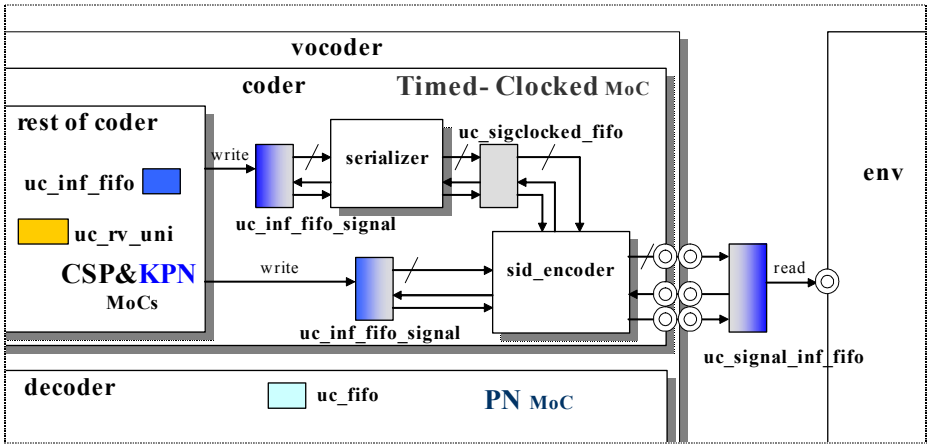



**Figura 4-20. Detalle de la especificación heterogénea del Vocoder.**

Por último, se realizó una implementación todo software de la especificación mediante la metodología de generación *SWGen*. Se generó el binario ejecutable para la plataforma CSB536FS con RTOS GX-Linux mediante el entorno mostrado en la Figura 4-7 y esquematizado en la Figura 4-8. El binario se cargó y ejecutó en la plataforma, verificando que la generación fue correcta.

# Conclusiones

En este trabajo de tesis se han logrado una serie de objetivos que habilitan al lenguaje SystemC para el diseño de sistemas embebidos basado en plataforma Hardware-Software.

Se han sentado las bases de una metodología de diseño de fuente única, en la que las distintas actividades de diseño se centran y parten de una especificación de nivel de sistema escrita en SystemC.

Para ello, se ha desarrollado la metodología de especificación *HetSC*. *HetSC* provee una novedosa sistematización en el uso del lenguaje SystemC para la realización de especificaciones concurrentes, modulares y heterogéneas. Para el soporte de heterogeneidad, *HetSC* provee una necesaria extensión de SystemC, sin modificar para ello el núcleo estándar del lenguaje. De esta forma, es posible escribir especificaciones de sistemas electrónicos embebidos completos, analizando los aspectos más importantes en cada parte del sistema, asegurando una serie de propiedades esenciales y haciendo posible un flujo de implementación adecuado para cada parte del sistema.

Enlazando con el último punto del párrafo anterior, mediante el desarrollo de la metodología *SWGen* se ha contribuido en el campo relativamente poco explorado de la generación de software embebido desde un lenguaje de nivel de sistema, y específicamente desde SystemC. *SWGen* permite generar automáticamente, a partir del código de especificación *HetSC*, el software embebido del sistema, incluyendo llamadas al RTOS. Además, el código generado mantiene una estructura similar a la de la especificación, lo que lo hace reconocible y más fácilmente adaptable.

Las metodologías *HetSC* y *SWGen* se han materializado como contribuciones reales a la comunidad SystemC en forma de documentación y librerías metodológicas. Ese material, además de publicado, está libremente disponible en la web.

Las dos líneas de investigación, especificación y generación software, han sido exitosamente probadas durante su desarrollo con diversos ejemplos y, finalmente, con un demostrador real, un EFR vocoder, sobre distintas plataformas HW/SW, tanto comerciales como una desarrollada íntegramente por el GIM.

También se han realizado contribuciones a líneas de investigación relacionadas, tales como el perfilado temporal de nivel de sistema y la mejora de la verificación de nivel de sistema mediante simulación.

Finalmente, en el plano teórico se ha demostrado que es posible soportar eficientemente diversos MoCs abstractos sobre un núcleo de simulación de eventos discretos como el de SystemC. De este modo, sobre un lenguaje con un núcleo compacto, pero descrito sin ambigüedades, el GIM y el autor han sido capaces de aportar contribuciones que se integran eficientemente con las de otros grupos de investigación europeos. Esta colaboración y contribución conjunta es necesaria para el desarrollo de una metodología de diseño competitiva en el plano internacional.

# Conclusions

In this thesis work, SystemC has been enabled as a valid language for the design of embedded systems based on HW/SW platforms.

The basis for a single-source design methodology, where the different design activities are centred and start from a system-level specification written in SystemC, has been settled.

For it, a specification methodology called *HetSC* has been developed. *HetSC* provides a novel sistemation in the usage of SystemC in order to build concurrent, modular and heterogeneous specifications. In order to support hetereogeneity, *HetSC* provides a necessary extension of SystemC without modifying its standard core. In this way, it is possible to write specifications of complete electronic embedded systems, to analyze the most important aspects of each system part, to ensure a set of essential properties and to enable a suitable implementation flow for each system part.

Linking with the last point of the previous paragraph, the development of the *SWGen* methodology has contributed to the relatively not much explored field of embedded software generation from system-level languages, and specifically from SystemC. *SWGen* enables the automatic generation of the system embedded software, including RTOS calls from the *HetSC* specification. Moreover, the generated code keeps a similar structure to that one of the specification, which makes it recognizable and more easily adaptable.

*HetSC* and *SWGen* methodologies have been materialized as real contributions to the SystemC community in the shape of documentation and methodological libraries. All this methodological material, as well as published, is also available in the web.

The two research lines, specification and software generation, have been successfully checked by means of several examples during their development, and finally by means of an actual demonstrator, an EFR Vocoder, over different commercial HW/SW platforms and over one developed by the GIM group.

Moreover, contributions to other related research lines have been done, such as system-level time profiling and the improvement of the coverage of verification by simulation of system-level concurrent specifications.

Finally, in a theoretical plane, it has been shown that it is possible to efficiently support several abstract MoCs over a discrete event simulation kernel such as that of SystemC. In this way, over a language with a compact, but unambiguously described core, the GIM and the author have been able to provide contributions which are efficiently integrated with the contributions of other European research groups. This collaboration and joint contribution is necessary for the development of a design methodology competitive in an international plane.

# Líneas de Investigación Actuales y Futuras

Las metodologías *HetSC* y *SWGen* se han diseñado para facilitar su extensión. Las librerías desarrolladas son demostradores que admiten aún trabajos de desarrollo (adaptaciones, optimizaciones, completitud, interfaz gráfica, etc) de cara a una aplicación industrial. Un ejemplo sería el portar *SWGen* para todos los canales de *HetSC* y para otras APIs de sistema operativo.

Hay también múltiples posibilidades de investigación e innovación para la extensión y mejora de las metodologías *HetSC* y *SWGen*. Esta sección se dedica a este punto, tratando de orientar acerca de cuales son y pueden ser las líneas interesantes de trabajo para el GIM y otros grupos que partan de los resultados aquí presentados.

En cuanto a la metodología de especificación *HetSC*, en la fecha de presentación de este trabajo, el GIM está involucrado en el proyecto IST ANDRES (*Análisis and Design of run-time REconfigurable, heterogeneous Systems*) **[AND08]**. En este proyecto, se explotan y mejoran las capacidades de las metodologías *HetSC* y *SWGen* para especificación e implementación de software embebido en un marco de especificación e implementación de sistemas adaptativos y heterogéneos, con software, hardware analógico y digital dinámicamente reconfigurable. Algunos resultados actuales del GIM en ANDRES son la cooperación de las metodologías *HetSC* y *SystemC-AMS* **[HVG07] [HDG07] [CHVG08] [RHDG08]** y, recientemente, de *HetSC* con *OSSS+R*, una metodología para la especificación de hardware reconfigurable en SystemC **[HVH08]**. El trabajo desarrollado en este proyecto producirá asimismo otros resultados, como el avance en la definición de un marco formal para *HetSC* basado en ForSyDe y el soporte de especificación de adaptatividad en *HetSC*, que se convertirá en *AHetSC* (**A**dapative **Het**erogeneous **S**ystem**C**), lo que tendrá asimismo implicaciones en el desarrollo de *SWGen*. En este sentido, serán interesantes los resultados acerca de las posibles interpretaciones de adaptatividad en software, incluyendo aquellas que se ajustan más a los objetivos de ANDRES. Otros trabajos interesantes para la futura extensión de *HetSC* estarían en la ampliación del conjunto de MoCs soportados (por ejemplo, incluyendo MoCs como Grafos de Computación, Redes de Petri, FSM, Chart-flows, etc). Más interesante aún es la posibilidad de generalización de algunos MoCs ya soportados, que soporten reglas más versátiles y cercanas a las necesidades de un mayor rango de aplicaciones concurrentes. Un caso concreto sería la definición de MoCs con concurrencia dinámica en los que se aseguren ciertas propiedades, como el determinismo, y su aplicación a SystemC. Finalmente, otra línea interesante sería la explotación de los conceptos de *HetSC* en TLM2.0. Es decir, la experiencia y resultados obtenidos en este trabajo se podrían emplear para analizar y tratar de mejorar los estilos de codificación de TLM 2.0.

Relativo a la metodología de generación de software embebido *SWGen*, existen aún cuestiones por desarrollar que deberían servir de base para la extensión de la librería *SWGen*. Sería interesante un mayor estudio en la sistematización de controladores de I/O. La exploración inicial de esta cuestión hecha en **[FHSV02]** admite un estudio más exhaustivo, que el GIM está realizando en el contexto del proyecto MEDEA Lomosa+ **[LoMo+]**. Otro punto interesante es el de generación de C y la comparativa con la generación de código C++ en diversas plataformas objetivo. Del mismo modo, y en concordancia con la evolución de las plataformas en un futuro inmediato, será de interés la extensión de la metodología *SWGen* para generación de software multiproceso y multiprocesador. Otro aspecto sería la provisión de un proceso de síntesis, de forma que, antes de la etapa de mapeado actualmente resuelta, que permite optar entre diversas implementaciones de la misma semántica de canal, exista una decisión automática de la implementación óptima. Evidentemente, esta fase de síntesis está ligada a las herramientas de DSE a través de un algoritmo decisor que use criterios de peso para el consumo, velocidad, etc. Otra posibilidad para convertir la metodología de generación en una metodología de síntesis, consistiría en una aplicación específica de las distintas técnicas de síntesis optimizadas para cada MoC. También es deseable el desarrollo de un marco formal asociado que asegure que en el proceso de generación se preserven las propiedades deseadas del sistema. Para ello, se podría aprovechar las conclusiones del trabajo de relación de *HetSC* con ForSyDe y aplicar las reglas de refinado de ForSyDe. De este modo, se proveería un soporte formal completo para la heterogeneidad horizontal y vertical. Por último, se podría añadir a la metodología un extractor desde el código SystemC de los servicios de RTOS necesarios (para generar automáticamente la configuración de RTOS). Sin embargo, este último punto se puede considerar más de desarrollo que de investigación en la actualidad.

Otras líneas de investigación involucrarían ambas metodologías *HetSC* y *SWGen*. Se ha mencionado la posibilidad de explorar MoCs más generales, que sean más flexibles o cercanos a las ciertas aplicaciones. Por otro lado, podría ser interesante explorar la posibilidad de describir MoCs específicos, enfocados a la obtención de propiedades que habiliten una generación de software automática más robusta. Por ejemplo, se podrían buscar las condiciones de especificación que permiten una predicción segura de los tamaños máximos de pila usados por cada proceso de la especificación.

Otras líneas de investigación interesantes están en la combinación de las potencialidades de SystemC como lenguaje de acción con las de otros lenguajes de especificación. En el proyecto ITEA SPICES, el GIM ha especificado una metodología de modelado en AADL, a partir de la cual es capaz de obtener una especificación ejecutable SystemC a través de la herramienta AADS **[VaVi08]**. En una línea parecida, el GIM actualmente trabaja en proyectos del Séptimo Programa Marco (FP7) como SATURN **[SAT08]**, en el que se definirá un marco en el que UML se podrá emplear para el modelado heterogéneo, utilizando SystemC como lenguaje de acción. De este modo, se podrán aplicar los conceptos desarrollados en *HetSC*.

# Anexos

# A.1 Traducciones

El presente anexo da la correspondencia entre algunas palabras y expresiones utilizadas y el tecnicismo anglosajón correspondiente.

**A medida**. *Full Custom***.**

**Analizador Sintáctico**. *Parser***.**

**Aproximadamente Temporales.** *Approximately Timed.* En un contexto TLM 2.0.

**Bandera.** *Flag.* En un contexto software, primitiva de sincronización que permite la notificación de evento y la sensibilización, bloqueo y disparo de un proceso. En un contexto hardware, un bit de un registro hardware.

**Brecha de Diseño.** *Design Gap*.

**Carga Útil Genérica**. *Generic Payload.* En un contexto TLM 2.0.

**Huella.** *Footprint.* En un contexto de desarrollo de SW embebido.

**Interfaces y Conectores Combinados.** *Combined Interfaces and Sockets.* En un contexto TLM 2.0.

**Compilador Portable.** *Retargetable Compiler.* En un contexto de desarrollo SW.

**Controlador**. *Driver*. En un contexto software, pieza de código que interactúa como intermediario entre las aplicaciones de usuario o de sistema operativo y un elemento hardware accesible bien mediante primitivas de entrada/salida ensamblador o bien mediante acceso directo a memoria física si el dispositivo hardware está mapeado en memoria.

**Débilmente Temporales**. *Loosely Timed*. En un contexto TLM 2.0.

**Derechos de Uso**. *Royalties.*

**Dirigida por Datos**. *Data-Driven***.**

**Dirigida por Eventos**. *Event-Driven***.**

**Conectores Iniciador y Objetivo.** *Initiator and Target sockets*. En un contexto TLM 2.0.

**Controlador Software**. *Driver*. En un contexto software o hardware/software.

**Estampa Temporal.** *Time Stamp*. En SystemC la estampa temporal se refiere a la etiqueta temporal con noción de tiempo físico.

**Grupo de nodos SDF**. *SDF Cluster*.

**Hilo.** *Thread*. En un contexto software.

**Instancia.** *Instance.* Se refiere a la materialización de elementos de un tipo o clase determinada.

**Interbloqueo.** *Deadlock.* Situación en la que dos o más procesos concurrentes no pueden continuar su ejecución debido a la espera del cumplimiento de condiciones que dependen de la continuación de su ejecución y que, por tanto, nunca se satisfarán.

**Latido del Sistema.** *Heartbeat or Tick Timer*. En un contexto software.

**Lista de Nodos**. *Netlist*.

**Memoria intermedia**. Memoria *buffer*. Memoria interpuesta entre cómputos independientes, uno generador y otro consumidor.

**MoC atemporal.** *Untimed MoC*.

**MoC síncrono.** *Synchronous MoC*.

**MoC temporal.** *Timed MoC*.

**Módulos Iniciador, Objetivo, Puente y Componente de Interconexión.** *Initiator, Target, Bridge and Interconnection Component modules.* En un contexto TLM 2.0.

**Número de disparos de cada nodo de un grafo SDF.** *Firing Schedule of a SDFG*.

**Objeto transacción.** *Transaction Object.* En un contexto TLM 2.0.

**Particularización.** *Port.* En el contexto de un software con un gran componente independiente de plataforma (por ejemplo, un RTOS), se refiere a la parte de de código dependiente de plataforma.

**Pila.** *Stack*. En un contexto de software.

**Planificación Estática de un Grafo SDF**. *Schedule of a SDFG*.

**Ranura Temporal.** *Slot*. En un contexto de MoC síncrono reactivo o síncrono perfecto.

**Retrollamada**. *Callback.* En el contexto de SystemC, método sobrecargable asociable a una clase SystemC y que es llamado por el kernel de simulación en algunas etapas de ejecución (básicamente elaboración y simulación).

**Script de Enlace.** *Link Script.*

**Solucionadores, Máquinas de Simulación.** *Solver.*

**Unidad de datos.** *Token.* Se refiere a la mínima unidad de datos transferible en un canal de comunicación determinado. En el contexto de este trabajo, dicho canal es un canal SystemC.

# Bibliografía


**[AaRo03]** E. Aarts and R. Roovers. "*IC Design Challenges for Ambient Intelligence*". Keynote. In Proc. of DATE'03. Munich. Germany. March, 2003.

**[ADA97]** "*Ada 95 Reference Manual Language and Standard Libraries: International* Standard *ISO/IEC 8652:1995*". T.S.Tuf & R.A.Duff (Editores). Springer. 1997. (Lecture notes in computer Science, Vol.; 1246). ISBN 3-540-63144-5.

**[ADLP02]** Assigoni, G. Duchini, L., Lavagno, L. Passerone, C., Watanabe, Y. **"***False path elimination in Quasi-Static Scheduling"*. In Proc. of DATE'02. 2002.

**[ALB99]** L. Arditi et al., Texas Instrument, V. Loubet, F. Bouali et al., CMA/INRIA Sophia-Antipolis, France. "*Using Esterel and Formal Methods to Increase the Confidence in the Functional Validation of a Commercial DSP*". In proceedings of ERCIM workshop on Formal Methods for Industrial Critical Systems, Trento, Italy, 1999.

**[AlKa04]** H. Al-Junaid and T. Kazmierski. "*An Extension to SystemC to allow modelling of Analoge and Mixed Signal Systems at Different Abstractions Levels"*. In SoC Design, Test and Technology Seminar, 15 September, Loughborough University, United Kingdom. 2004. Available at http://eprints.ecs.soton.ac.uk/9944/ .

**[AlKa05]** Al-Junaid, H. and Kazmierski, T. (2005). "*An Analogue and Mixed-Signal Extension to SystemC"*. The Institution of Electrical Engineers Proceedings Circuits, Devices & Systems. Available at http://eprints.ecs.soton.ac.uk/10644/ .

**[Amd67]** G.M. Amdahl "*Validity of the single-processor approach to achieving large scale computing capabilities*". In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.

**[AMSWG]** http://www.systemc.org/projects/ams-wg/

**[AND08]** http://andres.offis.de

**[ARC08]** http://aadl.enst.fr/arc/

**[BCEH03]** A.Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. Simone. "*The Synchronous Languages Twelve Years Later*". In Proceedings of the IEEE, 91(1):64-83, January 2003.

**[BCGH97]** F.Balarin, M. Chiodo, P.Giusto, H.Hsieh, A. Jurecska, L.Lavagno, C.Passerone, A.Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, B.Tabbara "*Hardware-Software Co-design of Embedded Systems: The POLIS Approach*". Kluwer. 1997.

**[BCHP03]** M.Bolado, J. Castillo, P. Huerta, H. Posadas, P. P. Sánchez. "*Implementation of a microprocessor core*". DS2-WP5-Q4/03 Deliverable of the Medea+ A511 TOOLIP Project. 2003-12.

**[BDT08]** M. Brun, J. Delatour, Y. Trinquet. *"Code Generation from AADL to a Real Time Operating System: An Experimentation Feedback on the Use of Model Transformation"*. In 13th IEEE International Conference on Engineering on Complex Computer Systems 2008, ICECCS'08. pp257-262.

**[BeBe91]** EA. Benveniste & G. Berry. "*The Synchronous Approach to Reactive and Real-Time Systems*". Proceedings of the IEEE, V.79, N.9, September, 1991.

**[BeGo92]** G. Berry and G. Gonthier. "*The Esterel synchronous programming language: Design, semantics, implementation"*. Science of Computer Programming, 19(2):87–152, Nov. 1992.

**[Berry00]** G. Berry. "*The Foundations of Esterel*". In Proof, Language, and Interaction: Essays in Honour of Robin Milner, 2000.

**[BHY01]** T.Bonnerud, B.Hermes, and T.Ytterdal. "*A Mixed-Signal Functional Level Simulation Framework based on SystemC"*. In *IEEE Custom Integrated Circuits Conference*. San Diego, California USA. May, 2001.

**[Binu07]** http://sources.redhat.com/binutils

**[BlDo04]** D.C.Black, J.Donovan. "*SystemC: From the Ground Up*". May, 2004. Kluwer Academic Publishers.

**[BLLN07]** C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng (eds.), *"Heterogeneous Concurrent Modeling and Design in Java (Volumes 1 and 2: Introduction to Ptolemy II)"*. EECS Department, University of California, Berkeley, UCB/EECS-2007-7/8, January 11, 2007. Available at http://ptolemy.eecs.berkeley.edu/.

**[BML96]** S.S.Bhattacharyya, P.K.Murphy, E.A.Lee. "*Software Synthesis from Dataflow Graphs*". Kluwer Academic Press, Norwell, MA, 1996.

**[BoSi91]** F.Boussinot and R. de Simone. "*The Esterel Language*" In Proceedings of the IEEE, September 1991.

**[BPCH04]** M. Bolado, H. Posadas, Javier Castillo, P. Huerta, P. Sánchez, C. Sánchez, H. Fouren, F. Blasco *"Platform based on open-source cores for industrial applications"*. Proc. of DATE'04, IEEE CS Press. 2004-02.

**[BRCB04]** Benny, O., Rondonneau, Chevalier, J., M., Bois, G., Aboulhamid, M. *"SoC software refinement approach for a SystemC Platform"*. Design & Verification Conference (DVCon'04). San Jose (USA). February, 2004.

**[BVH02]** F.Blasco, E.Villar and F.Herrera. "*System-Level Dynamic Estimation of Time Performance for Codesign based on SystemC and HW/SW platform*". DCIS'02, Santander, November 19-22. 2002.

**[BWHL03]** F.Balarin, Y. Watanabe, H.Hsieh, L.Lavagno, C.Passerone, A.Sangiovanni-Vincentelli. "*Metropolis: An Integrated Electronic System Design Environment*". In IEEE Computer Magazine. 2003. Available at http://www.embedded.eecs.berkeley.edu/*Metropolis*/index.html.

**[CAL08]** J. *Eker and J. Janneck. "An introduction to the Caltrop actor Language"*. White Paper. In http://embedded.eecs.berkeley.edu/caltrop/language.html.

**[CBRB04]** Chevalier, J., Benny, O., Rondonneau, M., Bois, G., Aboulhamis, M., Boyer, F.-R. *"SPACE: A Hardware/Software SystemC modeling platform including an RTOS"*. Languages for System Specification. Sous la direction de GRIMM, C., Norwell (USA): Kluwer Academic Publishers. P. 91-104. (ACM Portal) June 2004.

**[CAD08]** www.cadence.com

**[CCHM99]** Chang, H., Cooke, L., Hunt, M., Martin, G., McNelly, A. and Todd, L. "*Surviving the SoC Revolution: A guide to platform-based design*". Kluwer Academic Publisher. 1999.

**[CCKL00]** R. Clarisó, J.Cortadella, A. Kondratyev, L. Lavagno, C. Passerone and Y. Watanabe. "*Synthesis of Embedded Software for Reactive Systems"*. Proc. 2nd International Workshop on Integration of Specification Techniques for Applications in Engineering (INT'02), pages 2-20. April 2002.

**[CCOB04]** M.Conti, M.Caldari, S.Orcioni, G.Biagetti. *"Analog Circuit Modelling in SystemC"* in the book *"Languages for System Specification and Verification"*. C. Grimm (Editor), Kluwer Academic. 2004. pp. 229-242.

**[Cel08]** http://www.agilityds.com/

**[CFH01]** V. Fernández and F. Herrera. Chapter "*Introduction to SystemC" in "Design of Hardware/Software Embedded Systems*". Ed. E.Villar. Publishing Services of the University of Cantabria.2001.

**[CFHS03]** V.Fernández, F.Herrera, P.Sánchez and E.Villar. "*Embedded Software Generation From SystemC For Platform Based Design". Chapter 9, pgs 247-272, in "SystemC: Methodologies and Applications*". Ed. W.Mueller, W. Rosenstiel, J.Ruf. Kluwer Academic Publishers. 2003.

**[CGJL95]** M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh, K. Suzuki, A. Sangiovanni-Vincentelli, and E. Sentovich. *"Synthesis of Software Programs for Embedded Control Applications"*. In proc. of DAC'95, pp 587–592, San Francisco, CA, June 1995.

**[Choe05]** Choe, S. *"ESL enables software-driven SoCs"*. In D&R Industry articles. July 2005. Available in http://www.design-reuse.com/articles/10952/esl-enables-software-driven-socs.html and also in http://www.coware.com/PDF/ESLenablesSWdrivenSoCs.pdf .

**[CHPS03]** F.Herrera, H.Posadas, P.Sánchez and E.Villar. "*Systematic Embedded Software Generation from SystemC*", in A.Jerraya, S.Yoo, D.Verkest&N.When (Eds):"Embedded Software for SoC". Kluwer. 2003.

**[CHSV04]** F.Herrera, P.Sánchez, E.Villar. "*Modeling and Design of CSP, KPN, and SR Systems with SystemC*", pg 133-148 in Christoph Grimm (Editor.) "*Languages for System Specification*" CHDL Series. Kluwer, 2004. ISBN: 1-4020-7990-7.

**[CHSV05]** F.Herrera, P.Sánchez and E.Villar. "*Heterogeneous System-Level Specification in SystemC*" in Ed. P. Boulet (Eds) "*Advances in Design and Specification Languages for SoCs*". CHDL Series. 2005. Springer. ISBN: 0-387-26149-4.

**[CHV06]** F.Herrera and E.Villar. "*Mixing Synchronous Reactive and Untimed Models of Computation in SystemC*" in "*Applications of Specification and Design Languages for SoCs*". A. Vachoux (Editor) CHDL Series, Springer. Sept. 2006. ISBN: 1-4020-4997-8. pp 61-80.

**[CHVG08]** F.Herrera, E.Villar, C.Grimm, M.Damm and J. Haase. "*Heterogeneous Specification with HetSC and SystemC-AMS. Widening the support of MoCs in SystemC*" in E.Villar "Embedded Systems Specification and Design Languages". The CHDL Series, Springer. June. 2008.

**[Cie08]** *Cierto Virtual Component Co-Design* Datasheet. Disponible en http://www.cqpub.co.jp/dwm/editors/sn/edat2000/cadence/vcc.pdf

**[CJRD99]** Chiou, D., Jain, P., Rudolph, L., and Devadas, S. "*Application-specific memory management for embedded systems using software-controlled caches*". In Proc. Of Design, Automation Conference, DAC'99. 1999.

**[CKLM00]** J.Cortadella, A. Kondratyev, L.Lavagno, M.Massot, S.Moral, C.Passerone, Y.Watanabe and A.L. San Giovanni-Vincentelli. "*Task Generation and Compile Time Scheduling for mixed data-control embedded software"*. In Proceedings of Design Automation Conference. June 2000.

**[CKLP05]** J. Cortadella, A. Kondratyev, L. Lavagno, C. Passerone, and Y. Watanabe. *"Quasi-static Scheduling of Independent tasks for Reactive Systems"*. IEEE Transactions on Computer-Aided Design, 24(10):1492-1514, October 2005.

**[CKLT04]** J. Cortadella, A. Kondratyev, L. Lavagno, A. Taubin, and Y. Watanabe. *"Quasi-static scheduling for concurrent architectures*". Fundamenta Informaticae, 62(2):171-196, July 2004.

**[CLI00]** Chapter 3: "*VSS C Language Interface*" in *"VHDL Simulation Interfaces Manual"*. Version 2000. December, 2000.

**[CNFB06]** J. Chevalier, M.Nanclas, L.Filion, O.Benny, M.Rondonneau, G.Boys and E.M. Aboulhamid. *"A SystemC Refinement Methodology for Embedded Software"*. IEEE Design & Test of Computers. March/April 2006 (Vol. 23, No.2).pp. 148-158.

**[Cow08]** www.coware.com

**[CowV08]** http://www.coware.com/products/virtualplatform.php

**[CPPP06]** ISO/IEC Working Group WG21 of Subcommittee SC 22. *Technical Report on C++ Performance*. ISO/IEC TR 18015:2006(E). August, 2003. Availabe in http://www.open-std.org/jtc1/sc22/wg21/docs/projects#18015.

**[CPVM07]** J. Castillo, H. Posadas, E. Villar, M. Martínez (DS2). *Energy Consumption Estimation Technique in Embedded Processors with Stable Power Consumption based on Source-Code Operator Energy Figures*. XXII Conf. on Design of Circuits and Integrated Systems, DCIS'07 . 2007-11.

**[CrJT03]** T.J.Critzer, A.Jantsch, H.Tenhunen. "*Networks on Chip*". Kluwer Academic Publishers. February, 2003. ISBN 1402073925.

**[DCBG02]** M. Diaz-Nava, W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A.A. Jerraya. "*Component-Based Design Approach for Multicore SoCs*", Proc. Design Automation Conf. June 2002.

**[DDMP07]** A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G.Yang, H.Zeng, Q.Zhu. *"A Next-Generation Design Framework for Platform-Based Design"*. At proceedings of DCCon'07. February. 2007.

**[DeEs95]** J. Desel, J. Esparza, "*Free Choice Petri Nets*", Cambridge University Press, 1995

**[Dijk75]** E.W.Dijkstra. "*Guarded Commands, Non-Determinacy and Formal Derivation of Programs*". Comm. ACM, N8, V18. August, 1975. pp. 453-457. At http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD472.PDF.

**[DLHV06]** E. de las Heras, E. Villar *"Specification for SystemC-AADL interoperability"* IEEE Proceedings of the 5th International Workshop on Intelligent Solutions in Embedded Systems (WISES'07). 2007-06.

**[Dol01]** L.Doldi. "*SDL Illustrated- Visually design Executable Models*". May 2001. ISBN 2-9516600-0-6.

**[DS208]** www.ds2.es

**[DVM00]** Desmet, D., Verkest, D., and De Man, H.. "*Operating System based Software Generation for Systems-on-Chip*". In proceedings of the Design Automation Conference. DAC'00. 2000.

**[ECL08]** www.eclipse.org

**[Edw00]** S. Edwards, Synopsys, Mountain View, CA, USA. "*Compiling Esterel into Sequential Code"*. In Proceedings of the 37th Design Automation Conference (DAC'2000). Los Angeles, California, June 5-9, 2000. pp. 322-327.

**[Edwa97]** S.A.Edwards. "*The Specification of Heterogeneous Synchronous Reactive Systems*". PhD Thesis, University of California, Berkeley,1997. Available asUCB/ERLM97/31. http://ptolemy.eecs.berkeley.edu/papers/97/sedwardsThesis/.

**[Edw98]** S.A Edwards *"Embedded Systems"*. CSEE W4840. which in times references *"ESP: A 10-Year Retrospective"*. Embedded Systems Programming. November 1998.

**[EGV05]** K.Einwich, C.Grimm, A.Vachoux. "*SystemC-AMS Reference Manual*". Versión 1.0. April 2005.

**[EGVM02]** K.Einwich, C. Grimm, A.Vachoux, N. Martínez, F. Ruiz, C.Meise. "*Analog Mixed Signal Extensions for SystemC*". Available in http://www.systemc-ams.org . White Paper SystemC-AMS Study Group. June, 2002.

**[EHMC00]** Ellerve, Hemani,A, P., Miranda, M. and Catthoor, F. "*System level data format exploration for dynamically allocated data structures"*. In Proc. Of Design, Automation Conference, DAC'00.

**[ELIX]** EL/IX Base API Specification DRAFT. http://sources.redhat.com/elix/api/current/api.html.

**[EML07]** The Mathworks**. "***Embedded MATLAB User's Guide"*. March, 2007.

**[EN244]** ETSI EN 301 244 (GSM 06.53 version 4.0.1). "*Digital cellular telecommunications system (Phase 2); ANSI-C code for the GSM Enhaced Full Rate (EFR) speech codec*". January, 1998.

**[EN245]** ETSI EN 301 245 (GSM 06.60 version 4.0.1). "*Digital cellular telecommunications system (Phase 2); Enhaced Full Rate (EFR) speech transcoding*". 1998.

**[EN250]** ETSI EN 301 250 (GSM 06.54 version 4.0.1). "*Digital cellular telecommunications system (Phase 2); Test sequences for the GSM Enhanced Full Rate (EFR) speech codec*". 1998.

**[ERHT02]** Ernst, R., Ritcher, K., Haulbelt, C. and Teich, J. "*System Design for flexibility*". In Proceedings of the Design, Automation and Test in Europe, DATE'02.

**[FA08]** Catálogo de Farnell 2007-2008. www.farnellinone.com.

**[Fern98]** V.M. Fernández. "*Técnicas de Síntesis de Alto Nivel de Sistemas Digitales Altamente Testables*". Tesis doctoral presentada en la Universidad de Cantabria. Septiembre de 1998.

**[FGH06]** P. H. Feiler, D.P. Gluch, J.J.Hudak. "*The Architecture Analysis & Design Language (AADL): An Introduction"*. Technical Note CMU/SEI-2006-TN-011. February, 2006. Available in http://www.aadl.info/.

**[FHSV02]** "*Conclusiones: Metodología Industrial de Diseño de Sistemas Embebidos HW/SW*", Documento Entregable Final del proyecto FEDER *"Desarrollo de Metodologías Industriales de Diseño de Sistemas Embebidos Hw/Sw"*, V.Fernández, F.Herrera, P.Sánchez y E.Villar. Febrero 2002. Universidad de Cantabria.

**[FHT06]** J. Falk, C. Haubelt, and J. Teich. *"Efficient representation and simulation of model* based *designs in SystemC"*. In Proc. Of FDL'06, Darmstad, September 2006.

**[Free08]** www.freescale.com

**[FSCG00]** Fornaciari, W., Sciuto, D., Catthoor, F., and Gupta, R. "*Análisis of high-level address code transformations for programmable processors*". In In Proc. of the Design, Automation and Test in Europe DATE'00. 2000.

**[GaVN94]** Gajski, D. D., Vahid, F., Narayan, S. "*Specification and Design of Embedded System*". Prentice Hall, 1994.

**[GCC07]** http://gcc.gnu.org/

**[Gebo01]** C. Gebotys. "*Utilizing memory banthwith in DSP embedded processors*". In Proceedings of the Design Automation Conference. DAC'01.

**[Gepp00]** L. Geppert: "*Electronic Design Automation*", IEEE Spectrum, V.37, N.1, January 2000.

**[Ghe05]** F.Ghenassia (Ed.). "*Transaction Level Modelling with SystemC: TLM concepts and Applications for Embedded Systems*". Springer. 2005.

**[GIM08]** www.teisa.unican.es/gim

**[GLMS02]** T. Grötker, S. Liao, G. Martín & S. Swan. "*System Design with SystemC*". Kluwer Academic Publishers. Boston/Dordrecht/London. 2002.

**[GMCG01]** Gupta, S., Miranda, M., Catthoor, F., and Gupta, R. "*Analysis of high-level address code transformations for programmable processors*". In Proceedings of the Design Automation Conference. DAC'01. 2001.

**[Groe02]** T.Groetker. "*Modeling software with SystemC*". In 6th European SystemC Users Group Meeting. 2002. Available at www-ti.informatik.uni-tuebingen.de/~systemc

**[Gup02]** R. Gupta. "*HDL/C Interface Exploration*". Tech. Report submitted to the ICS Dpt. University of California. 2002. Available in http://mesl.ucsd.edu/gupta/cse237b-f04/PastProjects/Cinterface.pdf#search='HDL%20CLI'.

**[Gup95]** R.K.Gupta. "*Co-synthesis of Hardware and Software for Digital Embedded Systems*". Ed. Kluwer. August 1995. ISBN 0-7923-9613-8.

**[GYG03]** Gerstlauer, A., Yu, H. and Gajski, D.D. "*RTOS Modeling for System Level Design*". In Proc. of DATE'03. 2003.

**[GYJ01]** L. Gauthier, S. Yoo and A.A. Jerraya "*Automatic Generation of Application-Specific Operating Systems and Embedded Systems Software*", In Proc. of DATE'01. Mar. 2001.

**[GZD00]** D.D.Gajski, J.Zhu, R.Domer. "*SpecC: Specification Language and Methodology*". Kluwer Boston. 2000. ISBN: 0-7923-7822-9.

**[HaLi01]** C.C.Hanselman and B.Littlefield. *Mastering {MATLAB 6}: a comprehensive tutorial and reference*. ISBN 0-13-019468-9. 2001.

**[Hoa78]** C.A.R. Hoare. "*Communicating Sequential Processes*". Communications of the ACM, V.21, N.8, August 1978.

**[HCW08]** www.teisa.unican.es/~fherrera

**[HDG07]** J. Haase, M. Damm, C. Grimm, F. Herrera, E. Villar. *"Using Converter Channels within a Top-Down Design Flow in SystemC"*. The 15th Austrian Workhop on Microelectronics. Graz, Austria. 2007-10.

**[Her00]** F.Herrera. "*Especificación reutilizable para codiseño de un sistema de transmisión AAL-ATM*". Proyecto Fin de Carrera presentado en la Universidad de Cantabria. Dir. E. Villar. ETSIIT. 2000.

**[HeVA06]** F.Herrera and E.Villar. "*A framework for Embedded System Specification under Different Models of Computation in SystemC*". Proceedings of DAC'06, ACM. San Francisco. Julio. 2006.

**[HeVB06]** F.Herrera and E.Villar. "*Extension of the SystemC kernel for Simulation Coverage Improvement of System-Level Concurrent Specifications*". In Proc. Of FDL'06. Darmnstadt. September, 2006.

**[HeVi05]** F.Herrera and E.Villar. "*Mixing Synchronous Reactive and Untimed Models of Computation in SystemC*". In proceedings of FDL'05. Laussane. ECSI. Sept.2005.

**[HKMB05]** A. Helmerich, N. Koch, L. Mandel, P.Braun, P. Dombush, A. Gruller, P. Keil, R. Leisibach, J. Romberg, B. Schätz, T. Wild and G. Wimmel. "*Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising of a Technology Platform in the Area*". Final Report of FAST GmbH and the Technical University of Munich for the European commision. November, 2005.

**[HOHS07]** A.Herrholz, F. Oppenheimer, P.A.Hartmann, A.Schallenberg, W. Nebel, C.Grimm, M.Damm, J.Haase, F.Brame, Fernando Herrera, Eugenio Villar, I.Sander, A.Jantsch, A.-M.Foulliart, M.Martínez. *"The ANDRES project: Analysis and Design of Run-time REconfigurable, heterogeneous Systems"*. 17th International Conference on Field Programmable Logic and Applications. Amsterdam. 2007-08.

**[HOS07]** A. Herrholz, F. Oppenheimer, A. Schallenberg, W. Nebel, C. Grimm, M. Damm, Fernando Herrera, Eugenio Villar, A-M. Fouilliart, M. Martínez. *"ANDRES- ANalysis and Design of run-time REconfigurable, heterogeneous Systems"*. Workshop on "Adaptive Heterogeneous Systems-On-Chip and European Dimensions". In Proc. of Design Automation and Test in Europe 2007, DATE'07. 2007-04.

**[HPSV03]** F.Herrera, H.Posadas, P.Sánchez, E.Villar. "*Systematic Embedded Software Generation from SystemC*". In Proc. of Design Automation and Test in Europe 2003, DATE'03, Munich, 3-7, March. 2003.

**[HRFS00]** F.Herrera, R.Rodríguez, V.Fernández, P.Sánchez y E.Villar. "*Desarrollo de Metodologías Industriales de Diseño de Sistemas Embebidos Hw/Sw*", TEDEA. Septiembre 2000. Ciudad Real.

**[HSC08]** www.teisa.unican.es/*HetSC*

**[HSUV99]** F.Herrera, C.Sanz, I.Ugarte y E. Villar. "*Specification components: reusability at the HW/SW system specification level*", In *"Soft Cores, Reuse and Systems Integration"*. In Proc. VIUF'99 Orlando, Florida, October 1999.

**[HSV02]** F.Herrera, P.Sánchez and E.Villar. "*HW/SW Interface Implementation from SystemC for Platform Based Design*", In Proc. of FDL'02, Marseille 26-29 Sept. 2002.

**[HSV03]** F.Herrera, P.Sánchez, E.Villar. "*Modeling of CSP, KPN and SR Systems with SystemC*". FDL'03, Frankfurt ,23-26 Sept. 2003.

**[HSV04]** F.Herrera, P.Sanchez, E.Villar. "*Heterogeneous System-Level Specification in SystemC*". FDL'04, Lille ,13-17 Sept. 2004.

**[HUM08]** F.Herrera, E. Villar. "*HetSC Users Manual*". Universidad de Cantabria. Santander. 2008. Disponible en www.teisa.unican.es/*HetSC*/documentation.html.

**[HVG07]** F. Herrera, E. Villar, C. Grimm, M. Damm, J. Haase *"A General Approach to the Interoperability of HetSC and SystemC-AMS"*. Proceedings of the Forum on Design Languages 2007, FDL'07. Barcelona. 2007-09.

**[HVH08]** F. Herrera, E. Villar, P. A. Hartmann. "*Specification of HW/SW Adaptive Embedded Systems in SystemC"*. Proc.of FDL'08. Stuttgart. Germany. September, 2008.

**[IEEE05]** IEEE Computer Society. "*IEEE Standard SystemC Language Reference Manual*". IEEE Standard 1666-2005. March, 2006.

**[ITRS01]** A. Allan, D. Edenfeld, W. Joyner, A. Kahng, M. Rodgers, Y. Zorian: "*2001 Technology Roadmap for Semiconductors*". IEEE Computer. January 2002.

**[ITRS03]** W. Edenfeld, A. Kahng, M.Rodgers and Y.Zorian. "*2003 Technology Roadmap for Semiconductors*". IEEE Computer, January, 2004.

**[ITRS07]** *International Technology Roadmap for Semiconductors*. 2005.Design. Editions at http://public.itrs.net.

**[Jan04]** Axel Jantsch. "*Modeling Embedded Systems and SoC's*". Morgan Kaufman Publishers. Elsevier Science. 2004.

**[JDER01]** Jain, P., Devadas, S., Engelks, D., and Rudolph, L. "*Software-Assisted Cache Replacement Mechanisms for Embedded System*". In International Conference on Computer Aided Design. ICCAD'01. 2001.

**[JeWo05]** A.A.Jerraya, W.Wolf. "*Multiprocessor System-On-Chips*". 2005. Morgan Kaufman. ISBN 0-12385-251-X.

**[KRIS05]** R. Krishnan. *Future of Embedded Systems Technology*. Research Report # G229R. June, 2005. Available in www.electronics.ca/reports/embedded/systems_technology.html.

**[JiBr02]** Jiang,Y. and Brayton,R. "*Software Synthesis from synchronous specifications using logic simulation techniques*". In Proc. Of Design, Automation Conference, DAC'02.

**[Jon02]** G.Jong. "*A UML-Based Design Methodology for Real-Time and Embedded Systems*". In Proc. of DATE'02. 2002.

**[Kahn74]** G. Kahn. "*The Semantics of a simple Language for Parallel Programming*". Proc. of the IFIP Congress 74, North-Holland, 1974.

**[KaMi66]** R. M. Karp and R. E. Miller, "*Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing*." SIAM Journal, vol. 14, No. 6, pp. 1390-1411, Nov. 1966.

**[KCA01]** Kjeldsberd, P..G., Cathoor, F., and Aas, E. "*Detection of partially simulataneously alive signal in storage requirement estimation for data intensive applications"*. In Proceedings of the Design Automation conference. DAC'01.

**[KMNR00]** Keutzer, K., Malik, S., Newton, R., Rabacy, J., and Sangiovanni-Vincentelli, A."*System level design: Ortogonalization of Concerns and Platform Based Design*".IEEE Trans. On Computer-Aided Design of Circuits and Systems, 19(12). 2000.

**[KoMo99]** A.Koenig, B.E.Moo. "*Performance: myths, measurements, and morals; Part 1: myths"*. Journal of Object-Oriented Programming Vol. 12, no. 6.October, 1999.

**[Kri05]** L. Kriaa "*Modélisation et Validation des systemes hétérogènes: Definition d'un modéle d'execution*". PhD thesis, Université Joseph Fourier. ISC, TIMA Laboratory, Nov.2005.

**[LCVA07]** T.Leonardi, M.Conti, E.Vidal, E.Alarcón. "*SystemC-WMS Modeling of Control Techniques for Switching Amplifiers Targeting Polar RF Transmitters"*. In Proceedings of the Forum of Design Languages, FDL'07. Barcelona.

**[Lee00]** Lee.E.A."*What's ahead for embedded software?*" IEEE Computer Magazine. Sept. 2000.

**[Lee06]** E.A.Lee. "*The Problem with Threads*".IEEE Computer, vol. 36, no. 5, May 2006, pp. 33-42, available online as U.C. Berkeley EECS Department Technical Report UCB/EECS-2006-1 at http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html.

**[LeMe87]** E.A.Lee, D.G. Messerschmitt "*Synchronous Data Flow*". Proc. of the IEEE, Vol75, No.9,September, 1987.

**[LePa95]** E.A.Lee, T.M.Parks. "*Dataflow Process Networks*". Proceedings of the IEEE. 1995.

**[LeSV98]** E. Lee & A. Sangiovanni-Vincentelli. "*A Framework for comparing Models of Computation*". IEEE Trans. on CAD of ICs and Systems, V.17, N.12, December, 1998.

**[LHG98]** López, J.C., Hermida, R., Geisselhardt, W., "*Advanced Techniques for Embedded System Design and Test*". Kluwer Academic Publisher. 1998.

**[LHW00]** Lekatsas, H., Henkel, J. and Wolf, W. "*Code Compression as a variable in hardware/software co-design"*. In Proceeding of the International Symposium on Hardware Co-Design. CODES'00.

**[LinA98]** B.Lin. "*Efficient compilation of process-based concurrent programs without run-time scheduling*". In Proceedings of Design Automation and Test in Europe, DATE'98. February, 1998.

**[LinB98]** B.Lin. "*Software Synthesis of Process-Based Concurrent Programs"*. In Proceedings of Design Automation Conference. June, 1998.

**[LMS03]** L.Lavagno, G.Martin, B.Selic (Eds.). "*UML for Real Design of Embedded Real-Time Systems*". Kluwer. 2003. ISBN 1-4020-7501-4.

**[LoMo+]** http://www.lomosa.org/

**[Lope98]** A.López. "*Especificación y codiseño de sistemas embebidos en un entorno ADA-VHDL*". Proyecto Fin de Carrera presentado en la Universidad de Cantabria. Dir. E. Villar. ETSIIT. 1998.

**[MaGo95]** Marwedel, P. and Goossens, G. "*Code Generation for Embedded Processors*". Kluwer Academic Publishers. 1995.

**[Mas03]** A.Massa. *"Embedded Software Development with eCos"*. Prentice Hall PTR. Nov 25, 2003. ISBN: 0-1303-5473-2.

**[Micr08]** http://www.microcross.com/iMX-LiteKit-Datasheet.pdf.

**[MoBl02]** Mooney, V. and Bloug, D. "*A hardware-software real-time operating system* framework *for SoCs*". IEEE Design and Test of Computers. 2002.

**[MSCL02]** "*Master Slave Communication Library*". Version 2.0.1. Available at the MS2.0.1 Library distribution in www.systemc.org. 2002.

**[MuRR03]** Ed. W. Mueller, W. Rosenstiel (Ed), J.Ruf (Ed). "*SystemC: Methodologies and Applications*". Kluwer Academic Publishers. March 2003.

**[OBC06]** S.Orcioni, G.Biagetti, M.Conti. *SystemC-WMS: Mixed-Signal Simulation based on Wave exchanges"*. In the book *Aplications of Specification and Design Languages for SoCs*. A. Vachoux (Editor), The Chdl series. Springer. 2006. pp 171-184.

**[OCD08]** www.macraigor.com.

**[OMGS07]** MG. *OMG SysML specification*. http://www.omgsysml.org

**[ORSC00]** OPENCORES – Project: OpenRISC 1000. Available in http://www.opencores.org/projects.cgi/web/or1k/overview. See also http://www.eetimes.com/story/OEG20000228S0007.

**[OSCI05]** OSCI. "*SystemC Synthesizeable Subset*". 2005. En www.systemc.org.

**[OSCI08]** OSCI. *"OSCI TLM-2.0 User Manual"***.** June, 2008. En www.systemc.org.

**[Pag96]** I.Page. "*Hardware-Software Co-synthesis Research at Oxford*". Hardware-Software Co-synthesis Research at Oxford". May 1996. Informe técnico en http://citeseer.ist.psu.edu/cache/papers/cs/3848/ftp:zSzzSzftp.comlab.ox.ac.ukzSzpubzSzDocumentszSztechpaperszSzIan.PagezSzumist.pdf/page96hardwaresoftware.pdf

**[Pag04]** I. Page, "*Compiling Software to Gates*" in Embedded Systems Programming Magazine, December 2004.

**[Park95]** T.M.Parks. "*Bounded Scheduling of Process Networks*". PhD.Dissertation, Technical Report UCB/ERL-95-105. EECS Dpt. University of California, Berkeley. December, 1995. Available in http://ptolemy.eecs.berkeley.edu/publications/papers/95/parksThesis/.

**[PaSh04]** H.D.Patel&S.K.Shukla. "*SystemC kernel extensions for Heterogeneous System Modelling: A framework for Multi-MoC Modelling and Simulation*". Kluwer. Aug. 2004.

**[PaSh05]** H.D. Patel, S.K.Shukla. "*Towards a Heterogeneous Simulation Kernel for System-Level Models: A SystemC Kernel for Synchronous Data Flow Models*".IEEE Transactions on computer Aided Design of Integrated Circuits and Systems, V.24, N8. August, 2005. Available in http://fermat.ece.vt.edu/techpapers.htm.

**[PASV06]** H. Posadas, J. A. Adámez, P. Sánchez, E. Villar, F. Blasco (DS2). "*POSIX modeling in SystemC*". 11th Asia and South Pacific Design Automation Conference, ASP-DAC'06. 2006-01.

**[PAVE06]** H. Posadas, J. Adámez, E. Villar, F. Escuder (DS2), F. Blasco (DS2). *RTOS modeling in SystemC for Real-Time embedded SW simulation: A POSIX model*. Design Automation for Embedded Systems, V.10, N.4, Springer, pp.209-227. 2006-12.

**[Petr62]** C.A. Petri. "*Kommunikation mit Automaten*". PhD Thesis. Institut fur Instrumentelle Mathematik, Bonn. 1962.

**[PHLB95]** Pino, J.L., Ha, S., Lee, E.A., Buck, J.T.. "*SW Synthesis for DSP using Ptolemy*". Journal on VLSI Processing, vol 9., no.1, pp 7-21. January. 1995.

**[PHSV02]** H.Posadas, F.Herrera, P.Sánchez and E.Villar. "*Library for Microprocessor Core Analysis*". Final Deliverable DS2-T1.3-Q4/02. UC/ToolIP/IR/02 Internal Report of the MEDEA+ project. (Authorized by F.Blasco, DS2). December, 2002.

**[PHSV04]** H.Posadas, F.Herrera, P.Sánchez and E.Villar.(Univ. Cantabria) & F.Blasco. "*System-Level Performance Analysis in SystemC*". Proceedings of DATE'04, Paris. Feb, 2004.

**[Pla97]** P.J.Plauger. *"Embedded C++: An Overview"*. In Embedded Systems Programming. Available at www.embedded.com/97/feat9712.htm

**[PMS04]** H.Patel, D. Mathaikutty and S. Shukla, "*Implementing Multi-MoC Extensions for SystemC: Adding CSP & FSM Kernels for Heterogeneous Modeling*". FERMAT Technical Report. June, 2004.

**[POSIX04]** IEEE: "*Information technology-Portable Operating System Interface*", IEEE Standar. 1003.1, 2004.

**[PRS06]** F.Herrera, E.Villar. "*Improvements of the SystemC Referente Kernel: Controlled Pseudo-Random Schedulling*". Universidad de Cantabria. Santander. January, 2006. Available in English and in Spanish in www.teisa.unican.es/HetSC/Fix_Ext_downloads.html.

**[PWL01]** Passerone, C., Watanabe, Y., and Lavagno,L. "*Generation of minimal size code for schedule graphs*". In Proc. of DATE'01.

**[Resa05]** J.J. Resano. "*Técnicas de Optimización del Coste de Reconfiguración en Sistemas Dinámicamente Reconfigurables*". Tesis Doctoral remitida a la Universidad Complutense de Madrid. Mayo, 2005.

**[RHDG08]** J. Haase, M. Damm, C. Grimm, F. Herrera, E. Villar. *"Bridging MoCs in SystemC Specifications of Heterogeneous Systems"*. EURASIP Journal on Embedded Systems. Special Issue *"C-Based Design of Heterogeneous Embedded Systems"*. Volume 2008 (2008), Article ID 738136, 16 pages. In http://www.hindawi.com/getarticle.aspx?doi=10.1155/2008/738136 .

**[RHV07]** F.Herrera and E.Villar. "*A Framework for Heterogeneous Specification of Electronic Embedded Systems in SystemC*". ACM Transactions on Design Automation of Electronic Systems, Special Issue on Demonstrable Software Systems and Hardware Platforms, V.12, Issue 3, N.22. 2007-08.

**[Rodr00]** R.Rodrígez. "*Metodología para el diseño y verificación de sistemas embebidos*". Proyecto Fin de Carrera presentado en la Universidad de Cantabria. Dir. E. Villar. ETSIIT. 2000.

**[RPAG00]** I.Ripoll, P.Pisa, L.Abeni, P.Gai, A. Lanuse, S.Saez, B.Privat. "*RTOS Analysis"*. Deliverable D1.1 of the Open Components for Embedded Real-Time Application (OCERA) project. Noviembre, 2002. Disponible en http://www.ocera.org/archive/deliverables/ms1-month6/WP1/D1.1.html.

**[RPHF04]** H.Posadas, F.Herrera, V.Fernández, P.Sánchez, E.Villar. "*Single Source Design Environment for Embedded Systems based on SystemC*". On Journal on Design Automation for Embedded Systems. V.9. N.4, Springer.pp.293-312. Dec. 2004.

**[RS08]** Catálogo de RS 2007-2008. www.rsonline.es.

**[RSPF05]** A.Rose, S.Swan, J.Pierce, J.M.Fernández. "*Transaction Level Modeling in SystemC*". White Paper. Disponible en www.systemc.org.

**[RSRB05]** E. Riccobenne, P. Scandurra, A.Rosti and S.Boccio. "*A UML 2.0 profile for SystemC*". In Proceedings of DATE'05, IEEE.

**[RSRB06]** E.Ricobenne, P.Scandurra, A.Rosti and S.Bocchio. "*A Model-driven Environment for Embedded Systems*". In Proceedings of Design Automation Conference, DAC'06.

**[SAB02]** B.Sirpatil, J.Armstrong, J.Baker. "*Using SystemC to Implement Embedded Software*". International HDL Conference and Exhibition (HDLCon 2002), March, 2002.

**[San02]** Sangiovanni-Vincentelli-A. "*The context for platform-based design*". IEEE Design and Test of Computer. 2002.

**[San07]** A.Sangiovanni-Vincentelli. *"Quo Vadis, SLD? Reasoning about the Trends and Challenges of System Level Design"*. Proceedings of the IEEE. 95(3):467-506, March 2007.

**[SanA03]** I. Sander. "*System Modeling and Design Refinement in ForSyDe*". Thesis. Stockholm. 2003.

**[SanB03]** I. Sander. "*The ForSyDe Standard Library*"**.** KTH. Stockholm. April, 2003.

**[Sanc01]** P. Sánchez: "*Embedded SW and RTOS*", in E. Villar (Ed.): "Design of HW/SW embedded systems". University of Cantabria. 2001.

**[Sanc91]** P.P.Sánchez, "*PSAL: Estudio, Análisis e Implementación de Algoritmos de síntesis de Alto Nivel*". Tesis doctoral presentada en la Universidad de Cantabria. Marzo de 1991.

**[SAT08]** www.saturnsysml.eu

**[SCFS01]** "*SystemC Functional Specification*". October 2001. Disponible en www.systemc.org.

**[Schr07]** R.Schroll. *Design komplexer heterogener Systeme mit Polymorphen Signalen*. PhD thesis, Institut für Informatik, Universität Frankfurt, 2007.

**[ScNe06]** T.Schubert and W. Nebel. "*The Quiny SystemC Front End: Self-Synthesizing Designs*". In Proc. Of FDL'06. Darmnstadt, September 2006.

**[SCUG02]** "*SystemC User's Guide. Update for 2.0.1*" 2002. Disponible en www.systemc.org.

**[SCV03]** Members of the SystemC Verification WG. "*SystemC Verification Standard Specification. Version 1.0e*". May, 2003. www.systemc.org.

**[SCW08]** http://www.deit.univpm.it/systemc-wms

**[Shiu01]** Shiue, W-T. "*Retargetable compilation for low power*". In Proc. of the Int. Symposium on Hardware/Software Co-Design. CODES'01. 2001.

**[ShKe92]** K.Shumate, M.Keller. "*Software Specification and Design*". John Wiley & Sons, Boston/Dordrecht/London, 1992.

**[Shu02]** S.Sutherland. "*The Verilog PLI Handbook. A Users Guide and Conprehensive Reference on the Verilog Programing Language Interface*". 2nd Ed. Springer. ISBN 0-7923-7658-7. February, 2002.

**[Sid08]** www.sidsa.es

**[Sir02]** B.Sirpatil. "*Software Synthesis of SystemC Models*". Master Thesis. Virginia Polytechnic Institute and State University. Blacksburg, Virginia. July, 2002.

**[SLWS99]** Sgroi, M., Lavagno, L., Watanabe, Y., and Sangiovani-Vincentelli, A. "*Synthesis of Embedded Software using free-choise Petri nets*". In Proc. Of Desing Automation Conference, DAC'99. 1999.

**[SML06]** *SysML Partners web site*. http://www.sysml.org

**[SmSe91]** T.Smith and D.Setliff. "*Towards an Automatic Synthesis System for Real-Time Software*". In Proc. Of Real-Time System Symposium. 1991.

**[SPI06]** SPIRIT website. 2006. The SPIRIT Consortium. www.spirirtconsortium.com.

**[SPI08]** http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/

**[SPIC08]** http://www.spices-itea.org

**[STL05]** **"***Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library"*. Addison-Wesley, 2001.

**[StWo97]** J.Straunstrup, W.Wolf and all. "*Hardware/Software Codesign: Principles and Practice*". Kluwer Academic Publishers. 1997.

**[STZE99]** K.Strehl, L. Thiele, D. Ziegenbein, R.Ernst and J. Teich. "*Scheduling Hardware/Software Systems Using Symbolic Techniques*". In International Workshop on Hardware/Software Codesign, 1999.

**[SuSa96]** Suzuki, K. and Sangiovani-Vincentelli, A. "*Efficient software performance estimation methods for hardware/software codesign*". In Proc. of Design Automation Conference, DAC'96. 1996.

**[Swan01]** S.Swan. "*An Introduction to System-Level Modeling in SystemC 2.0*". White Paper. Cadence Design Systems, Inc. May 2001. Disponible en www.systemc.org.

**[SPC08]** http://www.cecs.uci.edu/~specc/reference/

**[SWG08]** www.teisa.unican.es/*SWGen*

**[SYN02]** Synopsys. "*Describing Synthesizable RTL in SystemC. Version 1.2*". November 2002.

**[Tabu95]** P.Tabuenca. "*Desarrollo de un Sistema Inteligente para Síntesis de Comportamiento: Aplicación al Co-diseño HW/SW*". Tesis doctoral presentada en la Universidad de Cantabria. Diciembre de 1995.

**[Too08]** http://toolip.fzi.de

**[UC08]** www.unican.es

**[VaVi08]** R.Varona, E.Villar. *Deliverable D3.4.1: "SystemC Generator - first release"*, del proyecto ITEA 05015 Spices. April, 2008.

**[VCH01]** E.Villar, C.I. Camargo and F.Herrera. "*Embedded Systems Design Methodology based on SystemC*". FDL'01, Lyon 3-7 Sept. 2001.

**[VEA00]** Open Verilog International. "*Verilog-AMS Language Reference Manual 2.0"*. January 2000.

**[VGE04]** A. Vachoux, C. Grimm, and K. Einwich. *"Towards analog and mixed-signal SoC design with SystemC-AMS"*. In IEEE DELTA'04, Perth, Australia, 2004.

**[VHA97]** IEEE Inc. "*IEEE Standard VHDL Language Reference Manual" (Integrated with VHDL-AMS Changes), IEEE std 1076.1, 1997.*

**[ViHe01]** E.Villar and F.Herrera. "*SystemC-Level specification in SystemC of a Residential Gateway*". DCIS'01 Oct. 2001.

**[ViSa93]** Villar, E. and P.Sánchez. "*Fundamentals and Standards in Hardware Description Languages*", in "Synthesis applications of VHDL". Kluwer Academic Publishers. 1993.

**[VLM96]** Vercauteren, S., Lin,B. and deMan, H. "*A Strategy for Real-Time Kernel Support in Application-Specific HW/SW Embedded Architectures*". In Proceedings of the Design Automation Conference. DAC'96. 1996.

**[VoUG06]** F.Herrera, E. Villar. "*Especificación de EFR Vocoder (GSM 6.60) en SystemC*". Universidad de Cantabria. Santander. Agosto, 2006.

**[VXW08]** Web, http://www.windriver.com/products/vxworks/

**[Wer02]** Wernli, L. *"Design and Implementation of a Code Generator for the CAL Actor Language"*. Dipl. Thesis. ETH Zurich. April 2002.

**[WGM99]** D.Webb**,** E. Gamma and K.Maciounas. "*Process Networks as Higher-level Notation for Metacomputing*". Proceeding of the Int. Parallel Symposium (IPPS'99), Workshop on Java for Distributed Computing. Puerto Rico, 1999.

**[YDG03]** H. Yu, R. Doemer, D. Gajski. "*Automatic Software Generation for System Level Design*", Center for Embedded Computer Systems (CECS), Technical Report 03-18, May 2003.

**[YDG04]** Yu, H., Doemer,R., Gajski, D. "*Embedded Software Generation from System Level Design Languages*". In Proceedings of ASPDAC'04. 2004.

**[YDG05]** Yu, H., Doemer, R., Gajski, D. "*Software and Driver Synthesis from Transaction Level Models*", Proceedings of the International Embedded Systems Symposium, "*From Specification to Embedded Systems Application*" (ed. A. Rettberg, Z. Mauro, F. Rammig), Springer, Manaus, Brazil, August 2005.

**[YNGJ02]** S. Yoo; G. Nicolescu; L. Gauthier & A.A. Jerraya: "*Automatic generation of fast simulation models for operating systems in SoC design*", Proc. of DATE'02, IEEE Computer Society Press, 2002.

# Índices Bibliográficos

En esta sección se dan otros índices referidos a la bibliografía de esta tesis. Se pretende que el lector pueda buscar rápidamente información en aspectos concretos y relacionados con la tesis tales como, documentación de SystemC y sus extensiones, especificación heterogénea, desarrollo de software embebido, etc. También se da la relación completa de las publicaciones del autor hasta la fecha y algunas de las del Grupo de Ingeniería Microelectrónica relacionadas directamente con este trabajo.

**AUTOR:**

Artículos (18): **[BVH02] [HDG07] [HeVA06] [HeVB06] [HeVi05] [HOHS07] [HOS07] [HPSV03] [HRFS00] [HSUV99] [HSV02] [HSV03] [HSV04] [HVG07] [HVH08] [PHSV04] [VCH01] [ViHe01]**

Revistas (3): **[RHDG08][RPHF04][RHV07]**

Capítulos de Libro (7): **[CFH01] [CFHS03] [CHPS03] [CHSV04] [CHSV05] [CHV06] [CHVG08]**

Sitios Web (2): **[HSC08] [SWG08]**

Proyecto Fin de Carrera : **[Her00]**

Tesis: **[Her08]**

Documentos Públicos en Web: **[HUM08] [PRS06] [VoUG06]***

Documentos Entregables de Proyecto: **[FHSV02]**

Más información acerca del autor y de sus publicaciones en:

www.teisa.unican.es/~fherrera

* Otros documentos disponibles en **[HSC08]** y **[SWG08]**.

**GIM/TEISA/UC:**

**[BCHP03] [BCPS04] [CPVM07] [DLHV06] [Fern98] [GIM08] [Lope98] [PASV06] [PAVE06] [Rodr00] [Sanc91] [Tabu95] [UC08] [ViSa93]**

**Generales de Sistemas Embebidos, Codiseño y Tendencias del Diseño Electrónico:**

**[AaRo03] [CCHM99] [CrJT03] [ERHT02] [Edw98] [GaVN94] [Gepp00] [HKMB05] [ITRS01] [ITRS03] [ITRS07] [JeWo05] [KMNR00] [Lee06] [Pag04] [Pla97] [San02] [San07] [SPI06]**

**Generales de SystemC:**

**[BlDo04] [GLMS02] [IEEE05] [MuRR03] [SCFS01] [SCUG02] [Swan01]**

**Extensiones y Librerías de SystemC:**

**[AlKa04] [AlKa05] [AMSWG] [BHY01] [EGV05] [EGVM02] [FHT06] [Ghe05] [Groe02] [LCVA07] [MSCL02] [OBC06] [OSCI05] [OSCI08] [PaSh04] [PaSh05] [PMS04] [Schr07] [SCW08] [VGE04] [RSPF05] [RSRB05] [ScNe06] [SCV03] [Sir02] [SYN02]**

**Lenguajes de Programación y de Diseño Electrónico:**

**[ADA97] [ALB99] [CAL08] [CLI00] [DLHV06] [Dol01] [EML07] [FGH06] [Gup02] [GZD00] [HaLi01] [Pla97] [SanA03] [Shu02] [SPI08] [VEA00] [VHA97] [Jon02] [LMS03] [OMGS07] [SML06] [SPC08]**

**Modelos de Computación y Especificación Heterogénea:**

**[BCEH03] [BeBe91] [BeGo92] [Berry00] [BLLN07] [BoSi91] [BWHL03] [CCOB04] [DDMP07] [DeEs95] [Dijk75] [Edwa97] [Edw98] [Hoa78] [Jan04] [Kahn74] [KaMi66] [Kri05] [Lee06] [LeMe87] [LePa95] [LeSV98] [Park95] [Petr62] [WGM99]**

**Software Embebido (Estimación y Generación en Metodologías de Codiseño):**

**[ADLP02] [ALB99] [BCGH97] [BeGo92] [Binu07] [BML96] [BRCB04] [CBRB04] [CCKL00] [CGJL95] [CJRD99] [CKLM00] [CKLP05] [CKLT04] [CNFB06] [CPPP06] [DCBG02] [DVM00] [Edw00] [EHMC00] [EML07] [Edw98] [GCC07] [Groe02] [GYG03] [GYJ01] [JDER01] [JiBr02] [Jon02] [KCA01] [KoMo99] [Lee00] [LHW00] [LinA98] [LinB98] [MaGo95] [Mas03] [PHLB95] [POSIX04] [PWL01] [RPAG00] [SAB02] [Sanc01] [ShKe92] [Sir02] [SLWS99] [SmSe91] [STL05] [STZE99] [SuSa96] [VLM96] [YDG03] [YDG04] [YDG05] [YNGJ02]**

**Poyectos:**

**[AND08] [FHSV02] [LoMo+] [SAT08] [SML06] [SPIC08] [Too08]**

# Índice de Figuras

**Capítulo 1**



**Capítulo 2**

**Capítulo 3**

**Capítulo 4**