CHALLENGES IN THE DESIGN OF CYBER-PHYSICAL SYSTEMS

DESAFÍOS EN EL DISEÑO DE SISTEMAS CYBER-FÍSICOS

John C. Chandy
Connecticut University
JC.Chandy@usa.com

(Tipo de Artículo: **REFLEXIÓN**. Recibido el 20/09/2010. Aprobado el 10/11/2010)

Abstract. *Cyber-physical systems ─CPS─ is a process that integrates computation with physical processes. Embedded computers, network monitoring and control of physical processes, usually have feedback loops where physical processes affect computations and vice versa. This paper discusses the challenges in designing these systems and raises the question of whether computers and existing network technologies provide an adequate basis for them. The conclusion is that to improve the design processes of these systems will not be enough to raise the level of abstraction, or verify, formally or not, the designs based on today's abstractions. The social and economic potential of the CPS is much higher than hitherto thought, throughout the world are investing heavily to develop this technology, but the challenges are considerable. To realize the full potential of CPS will have to reconstruct the processes of abstraction and computer networks, and processes must be open in the middle of the principles of physical dynamics and computation.*

Keywords. *Abstraction, Computer Science, cyber-physical, software engineering, synchronization.*

Resumen. Los sistemas cyber-físicos ─Cyber-Physical Systems CPS─ es un proceso que integra la computación con los procesos físicos. Los computadores embebidos, el monitoreo de redes y el control de procesos físicos, usualmente tienen ciclos de retroalimentación en los que los procesos físicos afectan los cálculos, y viceversa. En este artículo se examinan los desafíos en el diseño de estos sistemas, y se plantea la cuestión de si la informática y las tecnologías de redes actuales proporcionan una base adecuada para ellos. La conclusión es que para mejorar los procesos de diseño de estos sistemas no será suficiente con elevar el nivel de abstracción o verificar, formalmente o no, los diseños en los que se basan las abstracciones de hoy. El potencial social y económico de los CPS es mucho mayor de lo que hasta el momento se ha pensado; en todo el mundo se están realizando grandes inversiones para desarrollar esta tecnología, pero los retos son considerables. Para aprovechar todo el potencial de los CPS se tendrán que reconstruir los procesos de las abstracciones informáticas y de las redes, y los procesos se deberán acoger en pleno a los principios de las dinámicas físicas y de la computación.

Palabras clave: Abstracción, Ciencias Computacionales, cyber-físico, ingeniería de software, sincronización.

1. INTRODUCCIÓN

Los sistemas cyber-físicos es un proceso que integra la computación con los procesos físicos. Los computadores embebidos, el monitoreo de redes y el control de procesos físicos, usualmente tienen ciclos de retroalimentación en los que los procesos físicos afectan los cálculos y viceversa. En el mundo físico, el paso del tiempo es inexorable y la concurrencia es intrínseca. Ninguna de estas propiedades las tienen en cuenta las actuales abstracciones de la informática y de las redes.

Podría decirse que las aplicaciones de los CPS poseen el potencial de empequeñecer la revolución de las IT del siglo XX. Estos sistemas incluyen: dispositivos médicos altamente confiables, sistemas de vida asistida, sistemas avanzados de control de tráfico, control de procesos, conservación de energía, control medio-ambiental, aeronáutica, instrumentación, control de infraestructuras críticas ─por ejemplo, energía eléctrica, recursos hídricos y sistemas de comunicaciones─, robótica distribuida ─telepresencia, telemedicina─, sistemas de defensa, manufactura, y estructuras inteligentes.

Con todo esto es fácil imaginar nuevas aplicaciones para ellos, como la generación de energía micro-distribuida acoplada a la red eléctrica, donde cuestiones como sincronización de tiempo y seguridad son fundamentales. Los sistemas de transporte podrían beneficiarse considerablemente de una mejor inteligencia embebida en los automóviles, lo que podría mejorar su seguridad y eficiencia. Una red de vehículos autónomos podría mejorar drásticamente la eficacia de las fuerzas armadas y podría hacer sustancialmente más eficientes las técnicas de recuperación de desastres. Las redes de sistemas de control de edificios ─como la climatización y la iluminación─ podría mejorar significativamente su eficiencia energética y la variabilidad de la demanda, reduciendo nuestra dependencia de los combustibles fósiles y por tanto las emisiones de gases de efecto invernadero. En las comunicaciones, la radio cognitiva podrían beneficiarse enormemente de un consenso distribuido alrededor del ancho de banda disponible y de las tecnologías de control distribuido. Las redes financieras podrían cambiar radicalmente por la sincronización del tiempo. Los sistemas de servicios a gran escala aprovecharían la RFID ─Radio Frequency IDentification─ y otras tecnologías para el rastreo de bienes y servicios, ya que podrían adquirir la naturaleza de sistemas

de control distribuido en tiempo real. Los juegos distribuidos en tiempo real que integran sensores y actuadores podrían cambiar la naturaleza, relativamente pasiva, de las interacciones sociales en línea.

El impacto económico de cualquiera de estas aplicaciones sería enorme, sin embargo, la informática y las tecnologías de red actuales, pueden obstaculizar innecesariamente el progreso hacia estas aplicaciones. Por ejemplo, la falta de semántica temporal y de adecuados modelos de concurrencia en la informática, y las tecnologías de red actuales dificultan el previsible y exacto rendimiento de tiempo real. Las tecnologías de los componentes de software, incluyendo el diseño orientado a objetos y las arquitecturas orientadas a servicios, son construidas sobre abstracciones que coinciden mejor con el software que con los sistemas físicos. Muchas aplicaciones no se pueden lograr sin cambios sustanciales en las abstracciones fundamentales.

2. REQUISITOS PARA LOS CPS

Los sistemas embebidos siempre han tenido mayor fiabilidad y estándares de previsibilidad que los de computación de propósito general. Los consumidores no esperan que su televisor se bloquee y reiniciar el sistema; ellos cuentan con que los autos sean de alta fiabilidad, donde, de hecho, se utiliza un controlador computarizado para mejorar la confiabilidad y la eficiencia de los mismos. En la transición a los CPS, esa expectativa de fiabilidad se incrementará. De hecho, sin una mayor fiabilidad y previsibilidad, los CPS no se podrán utilizar en aplicaciones como el control de tráfico, la seguridad automovilística y el cuidado de la salud.

El mundo físico, sin embargo, no es totalmente predecible. Los Sistemas Cyber-físicos operaran en un ambiente extremadamente controlado, que debe ser resistente a condiciones inesperadas, y adaptable a los errores de los subsistemas. Un ingeniero se enfrenta a una tensión intrínseca: diseñar componentes predecibles y fiables hace que sea más fáciles ensamblarlos en sistemas predecibles y fiables. Pero ninguno componente es perfectamente fiable, y el entorno físico se encargará de frustrar la previsibilidad mediante la manifestación de condiciones inesperadas. Dado que los componentes son predecibles y fiables, ¿qué tanto de esa previsibilidad y confiabilidad puede depender del diseñador cuando diseña el sistema? ¿Cómo evitarlas en los diseños frágiles, donde pequeñas variaciones en las condiciones de funcionamiento esperadas causan fallas catastróficas?

Éste no es un problema nuevo en ingeniería. Los diseñadores de circuitos digitales han llegado a confiar en circuitos asombrosamente predecibles y fiables, y han aprendido a aprovechar intrínsecamente los procesos estocásticos ─el movimiento de los electrones─ para ofrecer una precisión y fiabilidad sin precedentes en la historia de la innovación humana. Pueden ofrecer circuitos que realizan una función lógica esencialmente a la perfección, a tiempo, y miles de millones de veces por segundo, durante años. Todo esto se construye sobre un fundamento altamente aleatorio. ¿Deberían los diseñadores de sistemas confiar en esta previsibilidad y fiabilidad?

De hecho, todos los sistemas digitales actuales dependen hasta cierto punto de esto. Existe un inmenso debate acerca de si esta dependencia impide de cierta forma el progreso de la tecnología de los circuitos. Los circuitos extremadamente pequeños son más vulnerables a la aleatoriedad del fundamento subyacente, y si los diseñadores de sistemas dependieran menos de la previsibilidad y la fiabilidad de los circuitos digitales, podríamos avanzar más rápidamente a características de tamaño más pequeño.

El mayor fabricante de semiconductores no ha seguido el paso y diseñó un proceso de fabricación de circuitos que ofrece puertas lógicas que funcionan el 80% del tiempo especificado. Estas puertas se consideran un completo fracasado, y un proceso que las produzca habitualmente tendrá un rendimiento muy pobre.

Pero los diseñadores de sistemas diseñan sistemas resistentes a esos fracasos. El propósito es mejorar el rendimiento, no mejorar la fiabilidad del producto final. Una puerta que falle el 20% de las veces es una compuerta fracasada, y un sistema exitoso ha de evitarla. Las puertas deben funcionar prácticamente el 100% del tiempo. La cuestión, por lo tanto, no es diseñar sistemas robustos, sino más bien en qué nivel de robustez se va a construir. ¿Debemos diseñar sistemas que trabajen con puertas lógicas que funcionan el 80% del tiempo especificado? o ¿debemos diseñar sistemas que reconfiguren las puertas que fallan el 20% del tiempo, y luego asumir que el trabajo que esas puertas realizan esencialmente es del 100% del tiempo?

El valor de poder contar con puertas que han superado la prueba de rendimiento para trabajar prácticamente el 100% del tiempo es enorme. Esa solidez en cualquier nivel de abstracción en el diseño del sistema es valiosa, pero no elimina la necesidad de robustecer los niveles superiores de abstracción. Los diseñadores de sistemas de memoria, a pesar de la alta fiabilidad y previsibilidad de los componentes, todavía utilizan sumas de comprobación y código de corrección de errores. Si tenemos un billón de componentes ─por ejemplo, un gigabyte de memoria RAM─ que funcionan un billón de veces por segundo, entonces, incluso la fiabilidad casi perfecta, en ocasiones entregará errores.

El principio que necesitamos seguir es simple: si los componentes son tecnológicamente factibles, deben ser predecibles y fiables en cualquier nivel de abstracción; si no es tecnológicamente factible, entonces el siguiente nivel de abstracción, por encima de ellos, debe compensar esa falencia con solidez. El éxito de los diseños de hoy sigue este principio, y todavía son técnicamente viables para diseñar puertas predecibles y fiables. Por lo tanto, también el diseño de sistemas cuenta con él. Es más difícil diseñar enlaces inalámbricos predecibles y fiables, por lo que debemos compensar los niveles superiores utilizando protocolos sólidos de codificación y de adaptación.

La pregunta obvia es si es técnicamente viable diseñar sistemas de software predecibles y fiables. En los fundamentos de la arquitectura de computadores y los lenguajes de programación, el software es perfectamente previsible y fiable, si limitamos el término "software" para referirnos sólo a lo que se expresa en simples lenguajes de programación. Dado un lenguaje de programación imperativo, sin concurrencia, como C, los diseñadores pueden contar con un computador para realizar exactamente lo que se especifica, con una fiabilidad prácticamente del 100%.

El problema surge cuando pasamos de programas simples a sistemas de software, y particularmente a sistemas cyber-físicos. El hecho es que incluso el más simple programa en C no es predecible ni fiable en el contexto de los CPS, ya que dicho programa no expresa los aspectos del comportamiento que son esenciales para el sistema. Se puede ejecutar perfectamente, hacer que su semántica coincida exactamente, y todavía no puede ofrecer el comportamiento que el sistema necesita. Por ejemplo, podría perder la sincronización del reloj. Debido a que la sincronización no está en la semántica de C, si un programa la pierde, en realidad sería irrelevante determinar si se ha ejecutado correctamente. Pero es muy pertinente para determinar si el sistema ha funcionado correctamente. Un componente que es perfectamente predecible y fiable resulta no ser predecible y fiable en las dimensiones que importan. Esto es una falla de la abstracción.

El problema empeora a medida que los sistemas de software se vuelven más complejos. Si damos un paso fuera de C y utilizamos sistemas operativos antiguos para realizar I/O o para crear *threads* concurrentes, inmediatamente pasaremos de una previsibilidad y fiabilidad esencialmente perfectas al comportamiento “salvaje” no determinista que reina en el diseño de software [1]. Semáforos, cerraduras de exclusión mutua, transacciones, y prioridades son algunas de las herramientas que los diseñadores de software han desarrollado para tratar de compensar esa pérdida de previsibilidad y fiabilidad.

Pero la pregunta que debemos hacernos es si esta pérdida de previsibilidad y fiabilidad es realmente necesaria; creemos que no. El software previsible y fiable no elimina la necesidad de diseñar sistemas robustos, pero cambia radicalmente la naturaleza del desafío. Si es técnicamente posible, debemos seguir el principio de hacer que los sistemas sean predecibles y fiables, y renunciar a esto sólo cuando haya evidencias convincentes de que no es posible ni rentable. No existen tales evidencias para el software. Por otra parte, tenemos una ventaja enorme: el fundamento sobre el que se construyen los sistemas de software —los circuitos digitales— es perfectamente previsible y fiable con respecto a las propiedades que nos interesan: la sincronización y la funcionalidad.

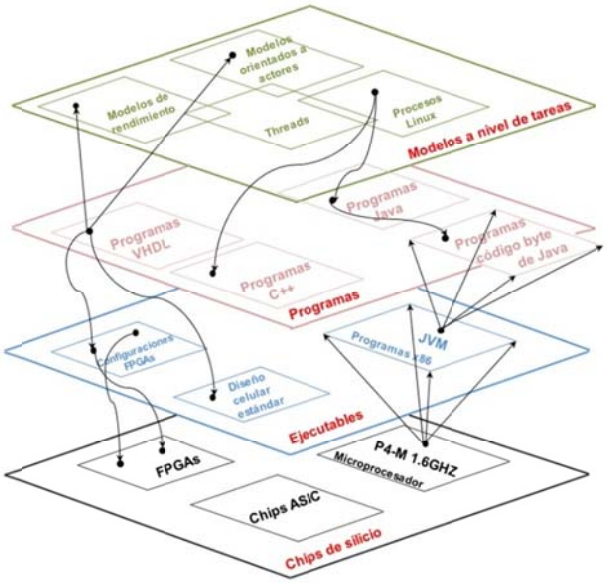

Fig. 1. Capas de abstracción en computación

Examinemos más a fondo la falla de la abstracción. La Fig. 1 ilustra algunas de las capas de abstracción de las que dependemos cuando diseñamos sistemas embebidos. En este diagrama de Venn tridimensional, cada cuadro representa un conjunto. Por ejemplo, en la parte inferior, tenemos el conjunto de todos los microprocesadores. Un elemento de este conjunto, por ejemplo el procesador Intel P4-M 1.6GHz, es un microprocesador particular. Por encima está el conjunto de todos los programas 86, cada uno de los cuales se pueden ejecutar en ese procesador. Este conjunto está precisamente definido —a diferencia del conjunto anterior, que es difícil de definir— por la arquitectura del conjunto de instrucciones x86 —ISA. Cualquier programa codificado en ese conjunto de instrucciones es un miembro del conjunto, como una implementación particular de una máquina virtual Java. Asociado a este miembro existe otro conjunto, el conjunto de todos los programas código byte de JVM. Cada uno de estos programas —normalmente— es sintetizado por un compilador desde un programa Java, que es miembro del conjunto de todos los

programas Java sintácticamente válidos. Una vez más, este conjunto está definido precisamente por la sintaxis de Java. Cada uno de estos conjuntos provee una capa de abstracción que se utiliza para aislar al diseñador ─persona o programa que selecciona los elementos del conjunto─ de los detalles que le preceden. Muchas de las mejores innovaciones en computación se han originado en una construcción y definición cuidadosa e innovadora de estos conjuntos.

Sin embargo, en el estado actual del software embebido, casi toda la abstracción falla. En ISA, la intención de ocultar al software los detalles de implementación del hardware ha fracasado, porque el usuario de ISA se preocupa por las propiedades de sincronización que ese estándar no garantiza. El lenguaje de programación, que oculta detalles de ISA a la lógica del programa, ha fracasado porque no existe un lenguaje de programación ampliamente utilizado que exprese las propiedades de sincronización. La sincronización es simplemente un accidente de la implementación. Un sistema operativo de tiempo real oculta los detalles del programa desde su instrumentación concurrente, sin embargo, esto no funciona porque la sincronización puede afectar el resultado. La red le esconde detalles de señalización a los sistemas, pero la mayoría de redes estándar no ofrecen sincronización garantizada.

Todos los diseñadores de sistemas embebidos enfrentan versiones de este problema. Los fabricantes de aviones deben almacenar los componentes electrónicos necesarios para la línea completa de producción de un modelo de aeronave para evitar tener que volver a certificar el software cuando el hardware cambia. "Actualizar" un microprocesador en un motor de una unidad de control de un auto requiere re-probar completamente el sistema. Incluso "corregir errores" en el software o el hardware puede ser extremadamente arriesgado, ya que se puede modificar el comportamiento de la sincronización.

El diseño de una capa de abstracción implica muchas propiedades, y los científicos computacionales han optado por ocultar las propiedades de sincronización de todas las abstracciones superiores. Wirth [2] dice que "*es prudente ampliar lo menos posible el marco conceptual de la programación secuencial y, particularmente, para evitar la noción del tiempo de ejecución*". En un sistema embebido, sin embargo, los cálculos interactúan directamente con el mundo físico, donde el tiempo no se puede abstraer innecesariamente. Incluso la computación de propósito general sufre de estos problemas. Ya que la sincronización no se especifica en los programas las plataformas de ejecución no la imponen, ya que las propiedades de sincronización de un programa no son repetibles. A menudo el software concurrente tiene un comportamiento dependiente de la sincronización, ya que pequeños cambios en ésta tienen grandes consecuencias.

Los diseñadores tradicionalmente han cubierto estas fallas buscando los límites del tiempo de ejecución del peor de los casos ─WCET─, y utilizando sistemas operativos de tiempo real ─RTOS─ con políticas de planificación predecibles. Pero esto requiere de importantes márgenes de fiabilidad, y en última instancia, esa fiabilidad es ─débilmente─ determinada por el plan de pruebas. Por otra parte, los WCET son una ficción problemática incremental, ya que las arquitecturas de los procesadores desarrollan técnicas cada vez más elaboradas para tratar estocásticamente con pipelines rigurosas, memoria jerárquica, y el paralelismo. Las arquitecturas de los procesadores modernos producen WCET virtualmente desconocidos; incluso los problemas simples demandan heroicos esfuerzos. En la práctica, los números confiables de los WCET llegan con muchas advertencias que son cada vez más raras en el software. El procesador ISA ha fracasado en proveer una abstracción adecuada.

El comportamiento de la sincronización en RTOS es secundario, y cada vez es menos controlable debido al incremento de la complejidad de los sistemas, por ejemplo, con la adición de la comunicación inter-procesos. Las cerraduras, la inversión prioritaria, los interruptores y cuestiones similares rompen los formalismos, forzando a los diseñadores a confiar en el plan de pruebas, que raramente identifica errores de sincronización. Peor aún, estas técnicas producen sistemas frágiles en los que los pequeños cambios pueden causar grandes fallas.

Mientras que no existan verdaderas garantías, no debemos descartar alegremente la previsibilidad que esto se pueda lograr. El hardware digital sincrónico proporciona un comportamiento de sincronización sorprendentemente preciso, y las abstracciones del software descartan varias órdenes de precisión de magnitud. Comparar la precisión de la escala de nanosegundo, con la que el hardware puede elevar una solicitud de interrupción de precisión del nivel de milisegundos, con la que los *threads* del software responden. No tenemos que hacerlo de esta manera.

3. ANTECEDENTES

La integración de los procesos físicos y la informática no es nueva. El término "sistemas embebidos" se utiliza desde hace algún tiempo para describir los sistemas de ingeniería que combinan procesos físicos con la computación. Las aplicaciones exitosas incluyen por ejemplo sistemas de comunicación, sistemas de control de tráfico aéreo, electrónica automovilística,

aplicaciones caseras, sistemas de armamento, juegos y juguetes. Sin embargo, la mayoría de estos sistemas embebidos son "cajas" cerradas que no exponen al exterior su capacidad de cómputo. La transformación radical que nos imaginamos proviene de la unión en red de estos dispositivos. Estas redes plantean desafíos técnicos considerables.

Por ejemplo, la práctica prevaleciente en el software embebido se basa en el plan de prueba para las propiedades de concurrencia y sincronización. Esto ha funcionado razonablemente bien, porque los programas son pequeños, y porque el software se almacena en una caja sin ninguna conectividad externa que pueda alterar su comportamiento. Sin embargo, las aplicaciones que nos imaginamos demandan que los sistemas embebidos sean muy sofisticados en características de conectividad, de modo que el plan de pruebas llega a ser inadecuado. En un entorno de red, se hace imposible probar el software en todas las condiciones posibles. Por otra parte, las técnicas de creación de redes de propósito general hacen que el comportamiento del programa sea mucho más imprevisible, por lo que uno de los mayores retos técnicos es lograr la sincronización predecible en esta situación.

Históricamente, los sistemas embebidos fueron en gran medida un problema industrial, como el de utilizar pequeños computadores para mejorar el rendimiento o funcionalidad de un producto. En este contexto, el software embebido difiere de otro sólo en sus recursos limitados: memoria pequeña, tamaño reducido de la palabra de datos, y relojes relativamente más lentos. Desde este punto de vista, el "problema del software embebido" es un problema de optimización. Las soluciones enfatizan en la eficiencia; los ingenieros escriben software a un nivel muy bajo ─lenguaje ensamblador o C─, evitan los sistemas operativos con un amplio conjunto de servicios y utilizan arquitecturas de computadores especializadas, como DSP programables y procesadores de red que proporcionen soporte al hardware para operaciones comunes. Estas soluciones han definido en los últimos 30 años la práctica de diseño y el desarrollo del software embebido. En un análisis que es tan válido hoy como hace más de 20 años, en 1988 Stankovic [3] lamentaba la aparición de conceptos erróneos según los cuales la computación en tiempo real era "equivalente a la informática rápida" o que era "ingeniería de rendimiento" ─la mayoría de la computación embebida es computación en tiempo real.

Pero las limitaciones en recursos de hace 30 años seguramente no son las limitaciones de la actualidad. En efecto, los desafíos técnicos se han centrado más en la previsibilidad y la robustez que en la eficiencia. Los sistemas embebidos de seguridad-crítica, como los sistemas de control de aviones para transporte de pasajeros, se ven extremadamente forzados por la mentalidad de "caja encerrada". Por ejemplo, con el fin de asegurar a 50 años un ciclo de producción inalámbrico para el vuelo de aviones, un fabricante se ve forzado a comprar, a la vez, un suministro de microprocesadores que ejecuten el software embebido para 50 años. Para asegurar que el rendimiento en tiempo real validado se mantenga, todos estos microprocesadores deben fabricarse en la misma línea de producción y desde los mismos planos. Los sistemas no podrán beneficiarse de los próximos 50 años de mejoras tecnológicas sin hacer de nuevo la validación y la certificación del software ─algo extremadamente costoso. Evidentemente, la eficacia es casi irrelevante en comparación con la previsibilidad, y la previsibilidad es difícil de lograr sin congelar el diseño a nivel físico. Es evidente que algo está mal con las abstracciones de software que se utiliza.

La falta de sincronización en las abstracciones informáticas ha sido explotada en gran medida por disciplinas de las ciencias computacionales como la arquitectura, los lenguajes de programación, los sistemas operativos y las redes. Las cachés, la comunicación dinámica, y la ejecución especulativa mejoran el rendimiento promedio de los casos a expensas de la previsibilidad. Estas técnicas hacen casi imposible saber cuánto tiempo se necesitará para ejecutar una pieza de código particular (Una respuesta simplista es que el tiempo de ejecución en un lenguaje Turing completo es de todos modos impredecible, por lo que no vale la pena ni siquiera intentar predecir su tiempo de ejecución. Esto no tiene sentido. Ningún sistema cyber-físico que dependa de la sincronización se puede implementar sin garantías de tiempo. Si la integridad de Turing interfiere con esto, entonces se debe sacrificar). Para hacer frente a estos problemas de arquitectura, los diseñadores pueden escoger arquitecturas de procesador alternativas tales como DSP programables, no para la eficiencia sino para la previsibilidad.

Las aplicaciones sensibles al tiempo son aún menos afectadas. La evidencia anecdótica desde la instrumentación basada en computador, por ejemplo, indica que el rendimiento en tiempo real entregado por los PC de hoy es aproximadamente la misma que la entregada por los PC de mediados de la década de 1980. Más de veinte años de la ley de Moore no han mejorado las cosas en esta dimensión. Por supuesto, esto no se debe exclusivamente a la arquitectura de hardware. Los sistemas operativos, los lenguajes de programación, las interfaces de usuario, y las redes se han vuelto más elaborados. Todos han sido construidos sobre una abstracción de software en la que el tiempo es irrelevante.

El punto de vista de tiempo real predominante parece haber sido bien establecido antes de que fuera popular la computación embebida [2]. El "cálculo" es ejecutado por una secuencia de terminación de transformaciones de estado. Esta abstracción central es la base del diseño de casi todos los computadores, los lenguajes de programación y los sistemas operativos que se utilizan hoy en día. Pero, desafortunadamente, esta abstracción puede no encajar muy bien en todos ellos.

Los sistemas cyber-físicos más interesantes y revolucionarios estarán conectados en red. Las técnicas de red más ampliamente utilizadas hoy introducen una gran variabilidad de sincronización y de comportamiento estocástico. Hoy en día, los sistemas embebidos suelen utilizar tecnologías de redes especializadas ─tales como buses CAN (Controller Area Network) en sistemas de fabricación, y buses FlexRay en aplicaciones automovilísticas. ¿Qué aspectos de las tecnologías de red deberían o podrían ser importantes en redes de mayor escala? ¿Que sean compatibles con las redes globales?

Para ser específicos, los recientes avances en sincronización de tiempo a través de las redes es una promesa de las plataformas interconectadas, que comparten una noción común de tiempo con una precisión conocida [4]. ¿Cómo cambiar la forma en que se desarrollan las aplicaciones cyber-físicas distribuidas? ¿Cuáles son las implicaciones para la seguridad? ¿Podemos reducir los riesgos de seguridad creados por la posibilidad de alterar la noción de tiempo compartido? ¿Pueden las técnicas de seguridad explotar eficientemente una noción compartida de tiempo para mejorar la robustez?

La tecnología de los sistemas operativos también está agobiada bajo el peso de los requisitos de los sistemas embebidos. Todavía los RTOS son esencialmente tecnologías de mejor esfuerzo. Para especificar las propiedades en tiempo real de un programa, el diseñador tiene que salir de las abstracciones de programación, y hacer llamadas al sistema operativo para establecer prioridades o para establecer las interrupciones del temporizador. ¿Son los RTOS simplemente un parche temporal para las fundaciones de computación inadecuadas? ¿Qué los puede reemplazar? ¿El límite conceptual entre el sistema operativo y el lenguaje de programación sigue siendo correcto? Un límite establecido en la década de 1960, sería realmente sorprendente si lo fuera.

Los sistemas cyber-físicos son por naturaleza concurrentes. Los procesos físicos son intrínsecamente concurrentes, y su acoplamiento con la computación requiere, como mínimo, la composición concurrente de los procesos computacionales con los físicos. Incluso hoy, los sistemas embebidos deben reaccionar concurrentemente a múltiples corrientes en tiempo real de estímulos de sensores y de actuadores de control múltiple. Lamentablemente, los mecanismos de interacción con el hardware de sensores y actuadores, construidos por ejemplo con el concepto de interruptores, no están bien representados en los lenguajes de programación. Han sido considerados dominio de los sistemas operativos, no del diseño de software. En cambio, las interacciones concurrentes con el hardware son revelados a los programadores a través de la abstracción de *threads*.

Los *threads*, sin embargo, son notoriamente problemáticos [1, 5]. Este hecho es a menudo atribuido a los seres humanos y no a la abstracción. Sutter y Larus [6] señalan que "*los seres humanos son desbordados rápidamente por la concurrencia y les resulta mucho más difícil razonar acerca de lo concurrente que del código secuencial. Aun las personas atentas pierden las posibles inserciones incluso entre simples colecciones de operaciones parcialmente ordenadas*". El problema se agrava mucho más en sistemas cyber-físicos en red.

Sin embargo, los seres humanos son muy eficientes para razonar acerca de sistemas concurrentes. El mundo físico es concurrente, y nuestra supervivencia depende de nuestra capacidad para razonar acerca de las dinámicas físicas concurrentes. El problema es que hemos elegido abstracciones concurrentes para el software que ni siquiera recuerdan vagamente a la concurrencia del mundo físico. Estamos tan acostumbrados a utilizarlas que hemos perdido de vista el hecho de que no son inmutables. ¿Podría ser que la dificultad de la programación concurrente es una consecuencia de las abstracciones, y que si estuviéramos dispuestos a dejar de lado las abstracciones, entonces el problema se podría arreglar?

4. SOLUCIONES

Estos problemas no son enteramente nuevos, y muchos investigadores creativos han hecho contribuciones. Los avances en la verificación formal, las técnicas de emulación y simulación, los métodos de certificación, los procesos de la ingeniería del software, los patrones de diseño y las tecnologías de componentes del software, todos ayudan. Estaríamos perdidos sin estas mejoras. Pero creemos que para formalizar su potencial, los sistemas CPS requieren fundamentalmente nuevas tecnologías. Es posible que estos sistemas surjan como mejoras incrementales en las tecnologías existentes, pero dada la falta de sincronización en las abstracciones básicas de la computación, esto parece improbable.

Sin embargo, las mejoras incrementales pueden tener un impacto considerable. Por ejemplo, la programación concurrente puede hacerse de una manera mucho mejor que mediante *threads*. Por ejemplo, Split-C [7] y Cilk [8] son lenguajes que como C soportan múltiples *threads,* con construcciones que son más fáciles de entender y de controlar que los *threads* primarios. Un enfoque relacionado combina extensiones del lenguaje con restricciones que limitan la expresividad de los lenguajes establecidos con el fin de obtener un comportamiento más consistente y predecible.

Por ejemplo, el lenguaje Guava [5] limita a Java para que los objetos no sincronizados no se puedan accesar desde múltiples subprocesos. Además, hace explícita la distinción entre cerraduras que aseguran la integridad de los datos leídos ─bloqueos de lectura─, y cerraduras que permiten la modificación de seguridad de los datos ─bloqueos de escritura. SHIM también proporciona interacciones más controlables de *threads* [10]. Estos cambios en los lenguajes reducen considerablemente el determinismo sin sacrificar mucho el rendimiento, pero todavía tienen riesgo de estancamiento y, nuevamente, ninguno de ellos confronta la falta de semántica temporal.

Como se indicó anteriormente, creemos que el mejor enfoque tiene que ser previsible en lo que técnicamente es factible. La computación concurrente predecible es posible, pero requiere abordar el problema de forma diferente. En lugar de comenzar con un mecanismo altamente no determinista como los *threads*, y confiando en el desarrollador para reducir lo no determinístico, debemos empezar con lo determinístico, los mecanismos componibles, e introducir lo no determinístico sólo cuando sea necesario.

Otro enfoque, que es mucho más un enfoque abajo-arriba, consiste en modificar la arquitecturas computacionales para ofrecer precisión en la sincronización [11]. Esto puede permitir la orquestación determinística de acciones concurrentes, pero deja abierta la cuestión de cómo se diseñará el software, ya que los lenguajes de programación y las metodologías tienen el tiempo completamente alejado del dominio del discurso.

Lograr la precisión en la sincronización es fácil si estamos dispuestos a renunciar al rendimiento; el desafío para la ingeniería es poder ofrecer tanto precisión como rendimiento. Si bien no podemos abandonar estructuras como cachés y pipelines y 40 años de progreso en lenguajes de programación, compiladores, sistemas operativos y redes, muchos tendrán que ser re-pensados. Afortunadamente, desde la abstracción, existe mucho trabajo sobre el que nos podemos basar. Se puede extender ISA con las instrucciones que proporcionen sincronización precisa a costos bajos [12]. Se puede utilizar memoria adicional ─Scratchpad─ en lugar de caché [13]. Pipelines altamente intercalados pueden ser eficientes y ofrecer sincronización predecible [14]. La memoria para gestionar los tiempos de pausa se puede limitar [15]. Los lenguajes de programación se pueden extender con semánticas cronometradas [16]. Mediante análisis estático es posible poder elegir modelos de concurrencia adecuados [17]. Los componentes del software se pueden hacer intrínsecamente concurrentes y sincronizados [18]. Las redes pueden proporcionar tiempo de sincronización de alta precisión [4]. El análisis programado puede proporcionar control de admisión y entregar la adaptabilidad en tiempo de ejecución sin imprecisiones de sincronización [19].

Complementando los enfoques abajo-arriba aparecen las soluciones de arriba-abajo, que se centran en el concepto de diseño basado en modelos [20]. En este enfoque, "los programas" se sustituyen por "modelos" que representan comportamientos del sistema en cuestión. El software se sintetiza a partir de los modelos. Este enfoque abre un rico espacio semántico que fácilmente puede adoptar dinámicas temporales ─ver por ejemplo [21] ─, incluyendo incluso las dinámicas temporales continuas del mundo físico.

Sin embargo, muchos desafíos y oportunidades de esta tecnología permanecen en desarrollos relativamente inmaduros. Abstracciones de tiempo ingenuas, como los modelos de tiempo discreto comúnmente utilizados para analizar sistemas de procesamiento de control y de señales, no reflejan el verdadero comportamiento del software y las redes [22]. El concepto de "tiempo de ejecución lógico" [16] ofrece una abstracción más prometedora, pero al final todavía confía en poder conseguir los tiempos de ejecución del peor caso para componentes del software. Esta solución arriba-abajo depende de una correspondiente solución abajo-arriba.

Algunos de los aspectos más interesantes del diseño basado en modelos se centran en la exploración de buenas posibilidades para la especificación y composición de interfaces. También han demostrado su utilidad para la computación de propósito general, reflejando las propiedades de comportamiento en las interfaces ─ver por ejemplo [23]. Sin embargo, en lo que se refiere a las propiedades que tradicionalmente no han sido expresadas en la informática, la capacidad para desarrollar y componer "teorías de interfaz" especializadas [24], son muy prometedoras. Estas teorías pueden reflejar propiedades de causalidad [25], cuyo comportamiento temporal abstracto utiliza recursos en tiempo real [26], limitaciones de sincronización [27], protocolos [28], recursos no renovables [29], y muchos otros [30].

Un enfoque particularmente atractivo que puede permitir aprovechar la considerable inversión en tecnología de software es el desarrollo de lenguajes de coordinación [31], que introducen nuevas semánticas en el nivel de interacción entre componentes en lugar del nivel de lenguaje de programación. Manifold [32] y Reo [33] son dos ejemplos, al igual que una serie de enfoques "orientados al actor" [34].

5. CONCLUSIÓN

Para aprovechar plenamente el potencial de los CPS se deben repensar las abstracciones fundamentales de la informática. Las mejoras incrementales, por supuesto, continúan ayudando, pero la orquestación eficaz del software y los procesos físicos requiere modelos semánticos que reflejen las propiedades de interés para ambos.

REFERENCIAS

1. E. A. Lee. "The problem with threads". *Computer*, Vol. 39, No. 5, pp. 33-42, 2006.
2. N. Wirth. "Toward a discipline of real-time programming". *Communications of the ACM*, Vol. 20, No. 8, pp. 577-583, 1977.
3. J. A. Stankovic. "Misconceptions about real-time computing: a serious problem for next-generation systems". *Computer*, Vol. 21, No. 10, pp. 10-19, 1988.
4. S. Johannessen. "Time synchronization in a local area network". *IEEE Control Systems Magazine*. Vol. 24, No. 2, pp. 61-69, 2004.
5. N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazieres, and F. Kaashoek. "Multiprocessor support for event-driven programs". In *USENIX Annual Technical Conference*, San Antonio, Texas, USA, June 9-14, 2003.
6. H. Sutter and J. Larus. "Software and the concurrency revolution". *ACM Queue*, Vol. 3, No.7, pp. 54-62, 2005.
7. D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. v. Eicken, and K. Yelick. "Parallel programming in Split-C". *Proceedings of the 1993 ACM/IEEE conference on Supercomputing.* Portland, Oregon, USA, pp. 262-273, November, 1993.
8. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. "Cilk: an efficient multithreaded runtime system". In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP).* Santa Barbara, California, USA, pp. 207-216, August, 1995.
9. D. F. Bacon, R. E. Strom, and A. Tarafdar. "Guava: a dialect of Java without data races". *ACM SIGPLAN Notices*. Vol. 35, pp. 382-400, 2000.
10. O. Tardieu and S. A. Edwards. "SHIM: Schedulingindependent threads and exceptions in SHIM". *Proceedings of the 6th ACM & IEEE International conference on Embedded software*. Seoul, Korea, pp. 142-151, October 22-24, 2006.
11. S. A. Edwards and E. A. Lee. "The case for the precisión timed (PRET) machine". *Proceedings of the 44th annual Design Automation Conference*. San Diego, CA, USA, pp. 264-265, June 4-8, 2007.
12. N. J. H. Ip and S. A. Edwards. "A processor extensión for cycle-accurate real-time software". In *IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, Vol. LNCS 4096. Seoul, Korea, pp. 449-458, August, 2006.
13. O. Avissar, R. Barua, and D. Stewart. "An optimal memory allocation scheme for scratch-pad-based embedded systems". *Trans. on Embedded Computing Sys.*, Vol. 1, No. 1, pp. 6-26, 2002.
14. E. A. Lee and D. G. "Messerschmitt. Pipeline interleaved programmable dsps: Architecture". *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Vol. 35, No. 9, 1987.
15. D. F. Bacon, P. Cheng, and V. Rajan. "The Metronome: A simpler approach to garbage collection in real-time systems". In *Workshop on Java Technologies for Real-Time and Embedded Systems*. Catania, Italy, pp. 466-478, November, 2003.
16. T. A. Henzinger, B. Horowitz, and C. M. Kirsch. "Giotto: A time-triggered language for embedded programming". *Proceedings of the IEEE EMSOFT 2001*, Vol. LNCS 2211. Tahoe City, CA, USA, 2001.
17. G. Berry. "The effectiveness of synchronous languages for the development of safety-critical systems". *White paper, Esterel Technologies*, 2003.
18. E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. "Actororiented design of embedded hardware and software systems". *Journal of Circuits, Systems, and Computers*, Vol. 12, No. 3, pp. 231-260, 2003.
19. E. Bini and G. C. Buttazzo. "Schedulability analysis of periodic fixed priority systems". *IEEE Transactions on Computers*, Vol. 53, No. 11, pp. 1462-1473, 2004.
20. J. Sztipanovits and G. Karsai. "Model-integrated computing". *IEEE Computer*, Vol. 30, No. 4, pp.110-112, 1997.
21. Y. Zhao, E. A. Lee, and J. Liu. "A programming model for time-synchronized distributed real-time systems". In *Real-Time and Embedded Technology and Applications Symposium (RTAS),* Bellevue, WA, USA, pp. 259-268 April 3-6 2007.
22. T. Nghiem, G. J. Pappas, A. Girard, and R. Alur. "Timetriggered implementations of dynamic controllers". *Proceedings of the 6th ACM & IEEE International conference on Embedded software*. Seoul, Korea, pp. 2-11, 2006.
23. B. H. Liskov and J. M. Wing. "A behavioral notion of subtyping". *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 6, pp. 1811-1841, 1994.
24. L. de Alfaro and T. A. Henzinger. "Interface theories for component-based design". In *First International Workshop on Embedded Software (EMSOFT)*, Vol. LNCS 2211. Lake Tahoe, CA, USA, pp. 148-165, October, 2001.
25. Y. Zhou and E. A. Lee. "A causality interface for deadlock analysis in dataflow". In *ACM & IEEE Conference on Embedded Software (EMSOFT)*, Seoul, South Korea, pp. 44-52, October 22-25, 2006.
26. L. Thiele, E. Wandeler, and N. Stoimenov. "Real-time interfaces for composing real-time systems". *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, Seoul, Korea, October 22-24, 2006.
27. T. A. Henzinger and S. "Matic. An interface algebra for real-time components". *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*. San José, CA, USA, pp. 253-266, April 4-7, 2006.
28. H. Kopetz and N. Suri. "Compositional design of RT systems: A conceptual basis for specification of

linking interfaces". In *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003),* Hakodate, Hokkaido, Japan, pp. 51-60, May 14-16, 2003.

29. A. Chakrabarti, L. de Alfaro, and T. A. Henzinger. "Resource interfaces". In R. Alur and I. Lee, (Eds), EMSOFT, Vol. LNCS 2855. Springer: Philadelphia, USA, pp. 117-133. 2003.
30. L. de Alfaro and T. A. "Henzinger. Interface-based design". In M. Broy, J. Gruenbauer, D. Harel, and C. Hoare (Eds.) *Engineering Theories of Software-intensive Systems, volume NATO Science Series: Mathematics, Physics, and Chemistry*, Vol. 195, pp. 83-104. Springer: USA, 2005.
31. G. Papadopoulos and F. Arbab. "Coordination models and languages". In M. Zelkowitz (Ed.) *Advances in Computers - The Engineering of Large Systems*, Vol. 46, pp. 329-400, 1998.
32. G. A. Papadopoulos, A. Stavrou, and O. Papapetrou. "An implementation framework for software architectures based on the coordination paradigm". *Science of Computer Programming*, Vol. 60, No. 1, pp. 27-67, 2006.
33. F. Arbab. "Reo: A channel-based coordination model for component composition". *Mathematical Structures in Computer Science*, Vol. 14, No. 3, pp. 329-366, 2004.
34. E. A. Lee. "Model-driven development - from object-oriented design to actor-oriented design". In *Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation, (a.k.a. The Monterey Workshop)*. Chicago, USA, September 24, 2003.