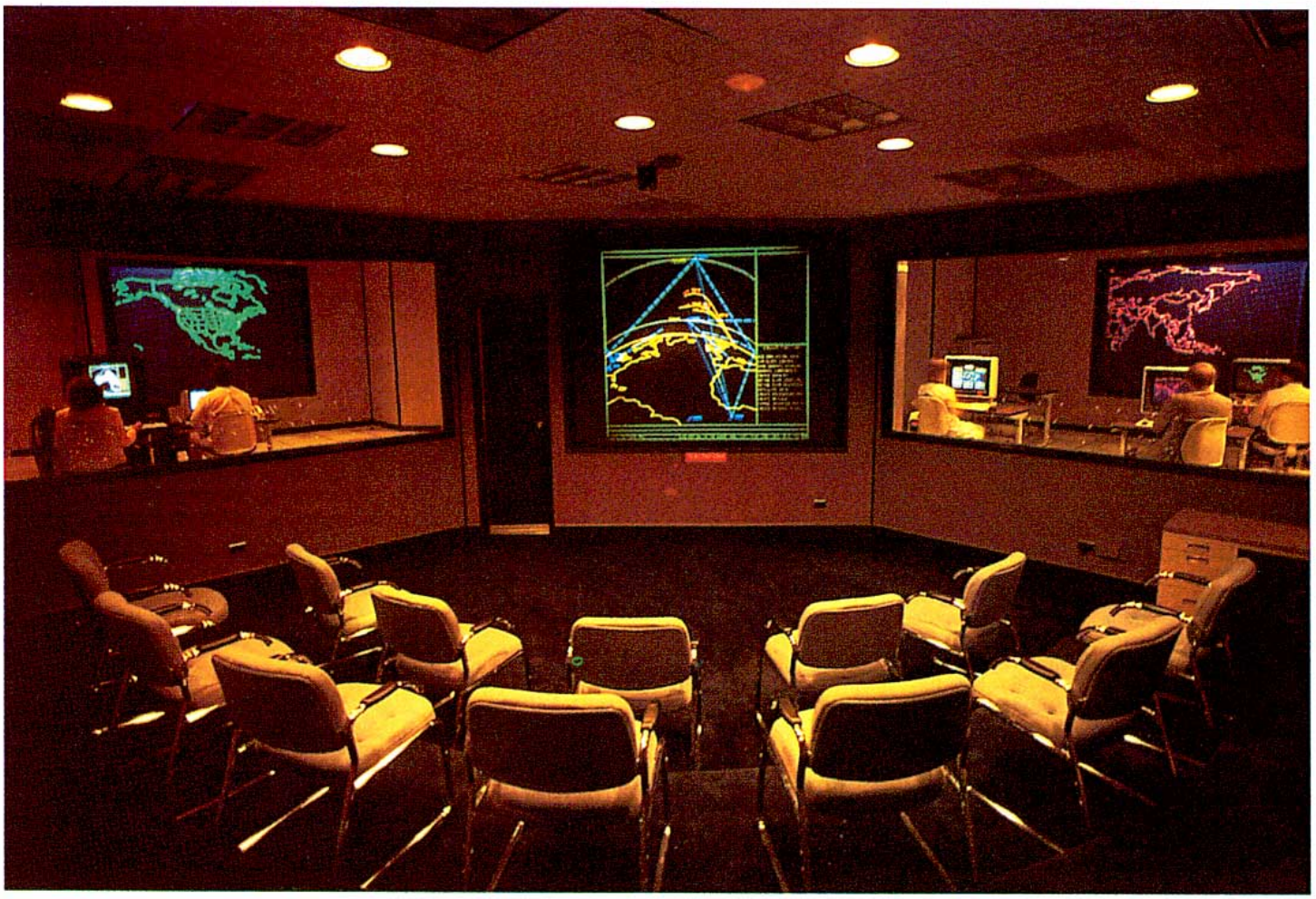


Aspectos software de la Iniciativa de Defensa Estratégica

El proyecto militar norteamericano Iniciativa de Defensa Estrátegica (IDE) o "Guerra de las Galaxias" ha abierto una serie de expectativas de desarrollo informático, tanto en su vertiente hardware como software.

El siguiente artículo, resumen de un informe publicado por D.L. PARNAS, ofrece una panorámica de los diferentes requerimientos en materia de software que, en principio, parecen inalcanzables con estado del arte de la tecnología software.

Este informe comprende un material completado mientras era miembro del "Panel on Computing in Support of Battle Management" reunido bajo el patrocinio de la Oficina de la IDE. Se nos solicitó la identificación de los problemas informáticos que debían resolverse antes de que se pudiese desplegar un sistema eficaz de misiles antibalísticos (ABM). Es claro que los computadores deben tener un papel crítico en los sistemas que la OIDE está considerando.

Los ensayos que constituyen este informe (...) se enviaron a la OIDE junto con mi dimisión del panel (...).

¿POR QUE EL SOFTWARE NO ES FIABLE?

La gente familiarizada tanto con la ingeniería software como con otras disciplinas más antiguas de la ingeniería observa que el estado del arte en software está significativamente retrasado respecto a otras áreas de la ingeniería. Cuando la mayoría de los productos de la ingeniería han sido completados, probados y vendidos, es razonable esperar que el diseño del producto es correcto y que funcionará de forma fiable. En el caso de los productos software es corriente encontrar que el software tiene fallos importantes y no funciona fiablemente para algunos usuarios. Estos problemas pueden persistir a lo largo de varias versiones y, a veces, empeoran con cada "mejora" del software. Mientras la mayoría de los productos vienen con una garantía expresa o implícita, los productos software llevan a menudo una denegación explícita de garantía.

El público, conocedor de sólo algunos casos de fallo software, puede interpretarlos como excepciones causadas por programadores ineptos. Los profesionales del software saben algo más: ni siquiera los programadores más competentes pueden evitar tales problemas.

Tipos de sistemas

Los productos de ingeniería pueden clasificarse como sistemas de estados discretos, sistemas analógicos o sistemas híbridos. (...)

Los sistemas analógicos forman el corazón de las áreas tradicionales de la ingeniería. La matemática de las funciones continuas es bien comprendida. Cuando decimos que un sistema es descrito por funciones continuas, estamos diciendo que no puede encerrar sorpresas ocultas. Un cambio pequeño en la entrada causará siempre cambios correlativamente pequeños en la salida. Un ingeniero que asegure, a través de un diseño cuidadoso, que los componentes del sistema están operando siempre dentro de su margen operativo normal, puede usar un análisis matemático para asegurar que no haya sorpresas. Cuando se combina con pruebas para asegurar que los componentes están dentro de su margen operativo, esto conduce a sistemas fiables antes de la aparición de los computadores digitales, cuando se construían sistemas de estados discretos, el número de estados en estos sistemas era relativamente pequeño. Con un número pequeño de estados eran posibles las pruebas exhaustivas. Estas compensaban la falta de herramientas matemáticas correspondientes a las utilizadas en el diseño de sistemas analógicos. Los ingenieros de tales sistemas aún tenían métodos sistemáticos que les permitían entender de forma completa el comportamiento de su sistema.

IDE: El proyecto software más ambicioso en cuanto a costes jamás intentado.

El diseño de muchos sistemas híbridos puede verificarse mediante una combinación de los dos métodos. Podemos identificar un número finito de estados operativos para los componentes con comportamiento discreto. Dentro de esos estados se puede describir el comportamiento del sistema mediante funciones continuas. Normalmente el número de estados que deben distinguirse es pequeño. Para cada uno de estos estados se pueden aplicar las herramientas de matemática continua para analizar el comportamiento del sistema.

Con la aparición de los computadores digitales, encontramos los primeros sistemas de estados discretos con muy grandes números de estados. Sin embargo, para fabricar estos sistemas era necesario construirlos utilizando muchas copias de subsistemas digitales muy pequeños. Cada uno de esos pequeños subsistemas se puede analizar y probar a un nivel exhaustivo. A causa de la estructura repetitiva, no eran necesarias las pruebas exhaustivas para obtener un hardware correcto y fiable. Aunque se encuentran errores de diseño en el hardware de computadores, estos son considerados excepcionales. Normalmente ocurren en aquellas partes del computador que no son estructuras repetitivas.

Los sistemas software son sistemas de estados discretos que no tienen la estructura repetitiva que se encuentra en la circuitería de los computadores. Raramente existe una buena razón para construir el software en forma de estructuras altamente repetitivas. El número de estados en los sistemas software es órdenes de magnitud mayor que el número de estados en las partes no-repetitivas de los computadores. Las funciones matemáticas que describen el comportamiento de estos sistemas no son funciones continuas, y la matemática de las ingenierías tradicionales no ayuda en su verificación. Esta diferencia contribuye de forma clara a la relativa falta de fiabilidad de los sistemas software y a la aparente falta de competencia de los ingenieros software.

Es una diferencia fundamental que no desaparecería con una tecnología mejorada.

Para entender el software

Existen dos técnicas para mejorar los problemas causados por esta diferencia fundamental en la tecnología: (1) la construcción del software en forma de conjuntos altamente organizados de pequeños programas y (2) el uso de la lógica matemática para reemplazar a la matemática continua.

Dividir el software en módulos y construir cada módulo mediante programas "estructurados" ayuda bastante. Cuando se hace en forma apropiada, cada componente se ocupa de un número pequeño de casos y puede ser analizado completamente. Sin embargo, los sistemas software reales tienen muchos de estos componentes y no existe una estructura repetitiva que simplifique el análisis. Incluso en los sistemas altamente estructurados, pueden ocurrir sorpresas o fallos de fiabilidad, porque la mente humana no es capaz de abarcar en su totalidad las muchas condiciones que pueden surgir a causa de la interacción de estos componentes. Además, encontrar la estructura correcta ha demostrado ser muy difícil. Los sistemas software reales bien estructurados son todavía poco frecuentes.

La lógica es una rama de la matemática que puede tratar con funciones que no son continuas. Muchos investigadores creen que puede asumir en la ingeniería software el papel que la matemática continua ostenta en las ingenierías mecánicas y eléctricas. Desafortunadamente, esto no se ha verificado aún en la práctica. El gran número de estados y la falta de regularidad en el software dan lugar a expresiones matemáticas extremadamente complejas. El uso disciplinado de estas expresiones supera las capacidades computacionales del programador humano y de los actuales computadores

Hay progreso en esta área, pero es muy lento y estamos muy lejos de poderlo aplicar siquiera a pequeños sistemas software. Con las técnicas actuales, las expresiones matemáticas que describen un programa son a menudo notablemente más difíciles de entender que el propio programa.

La educación de los programadores

Existe además un problema de personal que empeora las diferencias entre el software y otras áreas de la tecnología. La mayoría de los diseñadores en las disciplinas tradicionales de la ingeniería han sido educados para entender las herramientas de que disponen. La mayoría de los programadores ni siquiera pueden empezar a usar herramientas más simples de los ingenieros software.

¿POR QUE EL SOTWARE DE LA IDE SERA POCO DIGNO DE CONFIANZA?

(...) Para alcanzar la meta (del sistema IDE) necesitaríamos un software tan bien desarrollado que pudiésemos tener una confianza extrema en que el sistema funcionase correctamente en cuanto se requiere su uso.

Características del software de dirección de combate propuesto

1. Se precisa que el sistema identifique, realice el seguimiento y dirija las armas hacia objetivos cuyas características balísticas no se pueden conocer con certeza antes del momento del combate. Debe distinguir estos objetivos frente a otros señuelos cuyas características también son desconocidas.

2. Los cálculos se realizan sobre una red de computadores conectados a sensores, armas y entre sí, por canales cuyo comportamiento, en el momento en que se solicite la actuación del sistema, no puede predecirse a causa de las posibles contramedidas del atacante. El subconjunto de componentes del sistema que, en forma efectiva, estará disponible en el momento de la puesta en servicio, y a lo largo del mismo, no puede predecirse por la misma razón.

3. Será imposible probar el sistema bajo condiciones realistas antes de su uso real.

4. El período de servicio del sistema será tan pequeño que habrá poca posibilidad de intervención humana y ninguna posibilidad de depuración y modificación del programa durante el período de servicio.

5. Como muchos otros programas militares, hay limitaciones absolutas de tiempo real para los cálculos. Estos se compondrán básicamente de procesos periódicos, pero el número requerido de estos procesos y los requisitos computacionales de cada proceso, no pueden predecirse por anticipado, porque dependen de las características de los objetivos. Tampoco los recursos disponibles para cálculos se pueden predecir por anticipado. Ni siquiera podemos predecir el "caso peor" con cierta confianza.

6. El sistema de armas incluirá una amplia variedad de sensores y armas, la mayoría de las cuales precisarán en sí mismos un sistema software grande y complejo. La variedad de armas y sensores probablemente crecerá durante el desarrollo y tras el despliegue. Las características de las armas y sensores no se conocen aún y es probable que permanezcan cambiantes durante muchos años tras el despliegue. El resultado es que el sistema software global de dirección de combate tendrá que integrar un sistema software significativamente mayor de lo que nunca se ha intentado. Los componentes de este sistema estarán sujetos a modificaciones independientes.

"La ingeniería del software no hará alcanzables las metas de la IDE".

Implicaciones de estas características del problema

Cada una de estas características tienen claras implicaciones sobre la factibilidad de construir un software de dirección de combate que cumpla los requisitos de la IDE.

1. El software de control de disparo no puede escribirse sin hacer suposiciones sobre las características de las armas y objetivos enemigos. Esta información se utiliza para determinar los algoritmos de reconocimiento, los períodos de muestreo y las técnicas de filtrado de ruidos. Si el sistema se desarrolla sin el conocimiento de estas características, o con el conocimiento de que el enemigo pueda cambiar algunas de ellas en el día del combate, es probable que surjan en el software errores sutiles pero fatales.

2. Aunque ha habido progresos reales en el área del software tolerante a fallos ("fail-soft"), no he visto éxitos salvo en situaciones donde (a) los fallos probables pueden predecirse sobre la base de la historia pasada, (b) los fallos de componentes son improbables y estadísticamente independientes, (c) el sistema tiene excedentes de capacidad, (d) las exigencias de tiempo real, si las hay, son débiles, esto es, puede prescindirse de ellas sin efectos a largo plazo. Nada de ello se cumple en el software de dirección de combate que se ha planteado.

3. No se ha instalado nunca un sistema software de gran escala sin unas pruebas extensivas bajo condiciones realistas. Por ejemplo, en el software operacional para aeronaves militares, incluso las mínimas modificaciones requieren unas pruebas extensivas en tierra, seguidas por pruebas en vuelo en las que puedan aproximarse cerca de las condiciones del combate. Incluso con estas pruebas, pueden aparecer, y de hecho aparecen, errores software bajo las condiciones del combate. La imposibilidad de probar un sistema de defensa estratégica sobre el terreno, antes de que se necesite efectivamente su uso, se traducirá en que ninguna persona con sentido común tendría mucha fe en el sistema.

4. No es raro que se realicen modificaciones software sobre el terreno. Es habitual transportar a programadores hacia barcos de guerra mediante helicópteros; se encuentran anotaciones relacionadas con

depuraciones (de programas) en los camiones que transportaban computadores utilizados en Vietnam. Sólo mediante tales modificaciones, ese software llega a ser fiable. No se va a disponer de tales oportunidades en la guerra de 30 a 90 minutos en la que intervendría hipotéticamente un sistema de dirección de combate de defensa estratégica.

5. Los programas de este tipo deben cumplir fiablemente unas duras exigencias de tiempo real. En teoría, esto puede hacerse tanto mediante una planificación (de tareas computacionales) durante la ejecución o mediante una planificación prefijada. En la práctica, la eficiencia y la predecibilidad requieren algún tipo de preplanificación. A menudo se incorpora en los programas la planificación para una carga en caso peor. A menos que se pueda diseñar este tipo de planificación por anticipado, no se podrá tener confianza en que el sistema cumplirá las exigencias de tiempo real cuando se solicite su servicio.

6. Toda nuestra experiencia indica que las dificultades de construir software crecen con el tamaño del sistema, con el número de subsistemas modificables independientemente, y con el número de interfaces que deben definirse. Los problemas empeoran cuando las interfaces pueden cambiar. Las modificaciones subsiguientes aumentan la complejidad del software y la dificultad de hacer un cambio correctamente.

Conclusión

Todas las estimaciones de costo indican que este será el proyecto de software más masivo jamas intentado. El sistema tiene numerosas características técnicas que lo harán más difícil que los precedentes, independientemente del tamaño. Por causa de las extremas exigencias sobre el sistema y nuestra incapacidad de probarlo, nunca podremos creer, con cierta confianza, que hemos tenido éxito. Las armas nucleares seguirán siendo una potente amenaza.

¿POR QUE EL DESARROLLO CONVENCIONAL DEL SOFTWARE NO PRODUCE PROGRAMAS FIABLES?

El método convencional

La manera más fácil de describir el método de programación utilizado en la mayoría de los proyectos actuales me la dio un profesor que estaba explicando cómo enseñar programación. "Piensa como un computador", decía. Enseñaba a sus estudiantes a pensar y escribir lo que el computador tenía que hacer. (...) Primero los "pensamientos" del computador en pasos mayores y, más tarde, refinar estos pasos grandes en secuencias de pasos menores.

Este metodo, intuitivamente atractivo, funciona bien -en problemas demasiado pequeños para ser importantes. Pensamos que funciona porque funcionó en el primer programa que escribimos. Se puede seguir el método con programas que no tengan

Los aspectos informáticos de la dirección de combate son dificilmente superables.

ramas ni bucles. En cuanto nuestro pensamiento alcanza un punto en que la acción del computador debe depender de condiciones que no se conocen hasta que el programa se esté ejecutando, debemos apartarnos del método etiquetando una o más de las acciones y recordando cómo llegar a ellas. En cuanto introducimos bucles en el programa, hay muchas maneras de alcanzar algunos de los puntos, y deberíamos recordarlas todas. Según progresamos a través del algoritmo, reconocemos la necesidad de información sobre sucesos anteriores y de añadir variables a nuestra estructura de datos. Entonces tenemos que comenzar a recordar lo que significa cada dato y bajo qué circunstancias son significativos.

(...) Finalmente, cometemos un error. A veces lo notamos; a veces no se encuentra hasta la fase de pruebas. A veces, el error no es muy importante; sólo ocurre en ocasiones raras o imprevisibles. En ese caso, lo encontramos cuando el programa está en uso. A menudo, como uno precisa recordar tanto acerca del significado de cada etiqueta y de cada variable, la corrección de viejos problemas crea otros nuevos.

El efecto de la concurrencia sobre este método

En muchos de nuestros computadores hay varias fuentes de información y varias salidas a controlar. Esto podría llevar a concebir que el computador hace, a todos los efectos, muchas cosas a la vez. Si no puede predecirse por anticipado la secuencia de sucesos externos, la secuencia de acciones tomadas por el computador tampoco es predecible. El computador puede estar haciendo sólo una cosa cada vez, pero si se intenta "pensar como un computador", aparecen muchos más puntos donde la acción estará condicionada a lo que ocurrió en el pasado. Intentar diseñar estos programas pensando las cosas en el orden en que el computador las ejecutará, conduce a la confusión y da lugar a sistemas que nadie pude entender completamente.

El efecto del multiproceso

Cuando hay más de un computador en un sistema, sólo parece que el software esté haciendo más de una cosa cada vez, sino que realmente las está haciendo. No hay un programa secuencial que pueda estudiarse. Cualquier intento de "pensar como un computador" está obviamente condenado. Hay tantas posibilidades a considerar que únicamente unas pruebas extensivas pueden poner un principio de orden en las cosas. E incluso tras las pruebas tenemos incidentes. (...)

También los programadores profesionales utilizan este enfoque

He tenido ocasión de estudiar una gran cantidad de software práctico y de discutir programas con montones de programadores profesionales. En los últimos años, muchos programadores han tratado de mejorar sus métodos de trabajo, utilizando diversos enfoques de diseño de software. Sin embargo, cuando descienden al nivel de escribir programas ejecutables, vuelven al modo de pensar convencional. Aún estoy buscando un programa representativo, en uso práctico, cuya estructura no esté basada en la secuencia de ejecución esperada. Me alegraría encontrar uno.

Se discuten otros métodos en los cursos avanzados, en algunos buenos libros de texto y en las conferencias científicas, pero la mayoría de los programadores sigue usando el enfoque básico de pensar las cosas en el orden en que el computador las ejecutará. Esto es especialmente notorio en la fase de mantenimiento (corrección de deficiencias) de los programas.

Cómo librarse de este enfoque inadecuado

Debería estar claro que la escritura y comprensión de programas muy grandes en tiempo real "pensando como un computador" supera nuestras capacidades intelectuales. ¿Cómo ocurre entonces que tenemos tanto software que es bastante fiable como para utilizarlo?. La respuesta es simple: La programación es un asunto de prueba y error. Se escriben los programas sin ninguna esperanza de que serán correctos al principio. Se pasa tanto tiempo, al menos, probando y corrigiendo errores como se pasó escribiendo el programa inicial. Las grandes entidades tienen grupos separados encargados de las pruebas para asegurar la calidad. No puede confiarse en

los programadores a la hora de probar adecuadamente sus propios programas.

El software se entrega para su uso, no cuando se sabe que es correcto, sino cuando la frecuencia de descubrimiento de nuevos errores desciende a un nivel considerado aceptable por los responsables. Los usuarios se acostumbran a esperar errores y, a menudo, se les explica cómo evitar los errores hasta que el programa sea mejorado.

Conclusión

El software militar del que dependemos cada día es poco probable que sea correcto. Los métodos utilizados actualmente en la industria no son adecuados para construir los grandes sistemas software en tiempo real, que deben ser fiables desde su primera utilización. Se precisa un cambio drástico en los métodos.

LOS LIMITES DE LOS METODOS DE LA INGENIERIA SOFTWARE

Durante 25 años hemos sabido que nuestros métodos de programación eran inadecuados para grandes proyectos. La investigación en ingeniería del software, metodología de programación, diseño de software, etc., busca mejores herramientas y métodos. La orientación común de los resultados en estos campos es reducir la cantidad de cosas que un programador debe recordar cuando prueba y cambia un programa.

Dos líneas importantes de investigación son (1) la programación estructurada y el uso de semánticas formales de programas y (2) el uso de interfaces abstractas especificadas formalmente, para esconder la información sobre un módulo a los programadores que trabajan sobre otras partes. Una tercera idea, peor comprendida pero no menos importante, es el uso de procesos cooperativos secuenciales que ayuden a manejar las complejidades procedentes de la concurrencia y la multiprogramación. Ya en los últimos setenta, las ideas básicas de la ingeniería software se consideraban la "Verdad revelada" entre la comunidad universitaria. (...)

La separación entre teoría y práctica era grande y creciente. Los seguidores de los enfoques estructurados en software que decían creer en estos principios, no eran capaces de aplicarlos consistentemente.

(...) Para cubrir ese vacío, en 1977 la Marina de los EE.UU. lanzó el proyecto que ahora se conoce como Software Cost Reduction (SCR), del que se preveía una duración de dos a cuatro años. Todavía dura.

El proyecto ha aclarado dos cosas: (1) muchas de las propuestas de los universitarios pueden llevarse a cabo; (2) la buena ingeniería software es todo menos fácil. Estos métodos reducen, pero no eliminan, los errores. Reducen, pero no eliminan, la necesidad de pruebas.

El General Abrahamson, director de la SDI.

El trabajo del proyecto SCR se ha basado en los siguientes preceptos:

1. Deberían precisarse los requisitos software con un documento completo de requisitos de cada caja negra antes de comenzar el diseño del programa.

2. Debería dividirse el sistema en módulos utilizando el ocultamiento de la información (abstracción) antes de comenzar la escritura del programa.

3. Cada módulo debería tener una especificación formal precisa de caja negra antes de comenzar la escritura del programa.

4. Deberían utilizarse métodos formales para dar una documentación precisa.

5. Los sistemas en tiempo real deberían construirse como un conjunto de procesos secuenciales cooperativos, cada uno con su período y límites especificados.

6. Los programas deberían ser escritos utilizando las ideas de la programación estructurada.

Hemos demostrado que los primeros cuatro preceptos pueden ser aplicados al software militar (...). Todavía no hemos probado que estos métodos den lugar a un código fiable que cumpla las limitaciones de tiempo y espacio. Hemos encontrado que cada uno de estos preceptos es más fácil de enunciar que de aplicar. Aquellos que piensan que el diseño del software se facilitará, y los errores desaparecerán, no han atacado problemas de suficiente compeljidad.

La dificultad de la ingeniería software

Se pueden escribir documentos de requisitos software que sean completos y precisos. Entendemos los modelos matemáticos que hay detrás de los documentos y podemos seguir un procedimiento sistemático para documentar todas las decisiones necesarias sobre requisitos. Desafortunadamente, es difícil tomar las decisiones que se deben tomar para escribir esos documentos. Muy a menudo, no se sabe cómo hasta que se puede operar con el sistema. Solamente cuando se ha construido previamente un sistema similar es fácil determinar por anticipado los requisitos. Vale la pena, pero no es fácil.

Sabemos cómo descomponer en módulos sistemas complejos cuando conocemos el conjunto de decisiones de diseño que deben tomarse en la implementación. Cada una de ellas debe asignarse a un único módulo. Esto puede hacerse cuando estamos construyendo un sistema que recuerda a otro construido antes. Cuando estamos resolviendo un problema totalmente nuevo, de seguro pasaremos por alto difíciles decisiones de diseño. El resultado será una estructura que no separa completamente las opciones importantes y no minimiza la complejidad.

Sabemos cómo especificar interfaces abstractas para módulos. Tenemos un conjunto de notaciones estándar para uso en esta tarea. Desgraciadamente, es muy difícil hallar la interfaz correcta. La interfaz debería ser una abstracción del conjunto de todos los diseños alternativos. Se puede encontrar esa interfaz sólo cuando se entienden los diseños alternativos.

La conclusión común a extraer de todas estas observaciones es que, incluso con principios claros de diseño software, se precisa una amplia experiencia con sistemas similares para diseñar un software de calidad y fiable.

Los lenguajes de programación

A causa de las amplísimas mejoras en la productividad que se percibieron con la introducción de los lenguajes compilados, muchos continúan buscando otra mejora con la introducción de mejores lenguajes. Una mejor notación siempre ayuda, pero no podemos esperar que los nuevos lenguajes proporcionen la misma magnitud de mejora que la obtenida por los primeros de estos lenguajes. Nuestra experiencia en el programa SCR no ha mostrado que la falta de un lenguaje sea un problema fundamental.

Los lenguajes de programación ahora son suficientemente flexibles, de forma que pueden utilizarse casi cualquiera de ellos para casi cualquier tarea. Deberíamos buscar simplificaciones en los lenguajes de programación, pero no podemos esperar que esto de lugar a diferencias sustanciales.

Los entornos de programación

El éxito del UNIX como herramienta de desarrollo de programas ha dejado claro que el entorno en que se trabaja crea diferencias. La flexibilidad del UNIX nos ha permitido eliminar muchas de las tareas pesadas de ordenación y coordinación que aparecen en la producción de grandes programas. Consecuentemente, hay una extensa investigación en entornos de programación. También aquí espero que puedan hacerse pequeñas mejoras al basar las herramientas en mejores notaciones, pero no que se produzcan grandes rupturas. Los problemas de nuestro entorno de programación no han sido un impedimento importante para nuestro trabajo para SCR.

La ingeniería software no hará alcanzables las metas de la IDE

Aunque creo que cualquier nueva investigación sobre los métodos de ingeniería software pueden conducir a mejoras sustanciales en nuestra capacidad de construir grandes sistemas software en tiempo real, este trabajo no superará las dificultades inherentes a los aspectos informáticos de la dirección de combate de la IDE. Los métodos de ingeniería software no eliminan los errores. No eliminan las diferencias básicas entre la tecnología del software y otras áreas de la ingeniería. No eliminan la necesidad de unas pruebas de campo extensivas, o la necesidad de oportunidades de revisar el sistema una vez que está en uso. Lo más importante, hemos aprendido que la aplicación con éxito de estos métodos depende de la experiencia en la construcción y mantenimiento de sistemas similares. No hay experiencia en dirección de combate para la IDE.

Conclusión

No soy un hombre modesto. Creo que dispongo de conocimientos tan claros y tan amplios sobre los problemas de la ingeniería software como cualquiera que conozco.

> *La programación automática no supondrá ninguna ayuda, debido a la ausencia de información para escribir especificaciones fiables.*

Si se me encargase la tarea de construir el sistema, con todos los recursos que yo quisiera, no podría hacerlo. Y no espero que los próximos 20 años de investigación cambien este hecho.

LA INTELIGENCIA ARTIFICAL Y LA INICIATIVA DE DEFENSA ESTRATEGICA

Hoy en día son de uso común dos definiciones completamente diferentes de IA.

- IA.1: El uso de computadores para resolver problemas que sólo podían resolverse previamente con la aplicación de la inteligencia humana.

- IA.2: El uso de un conjunto específico de técnicas de programación conocidas como heurísticas o basadas en reglas. En este enfoque se estudia a los expertos humanos para determinar que heurísticas o reglas de experiencia utilizan para resolver los problemas. Normalmente se les pregunta por estas reglas. Posteriormente, estas son codificadas como entradas a un programa que trata de comportarse de acuerdo con ellas. En otras palabras, el programa se diseña para resolver un problema del modo que los humanos parecen resolverlo.

Debe resaltarse que la primera definición define la IA como un conjunto de problemas, la segunda define la IA como un conjunto de técnicas. La primera definición tiene un significado cambiante. En la Edad Media se pensaba que la aritmética, precisaba inteligencia. Ahora reconocemos que es un acto mecánico. Algo que hoy puede entrar dentro de la definición de IA.1, una vez que vemos como trabaja el programa y comprendemos el problema, deja de considerarse como IA en el futuro.

Aunque es posible que un determinado trabajo satisfaga ambas definiciones, el mejor trabajo de IA.1 que yo he visto no utiliza métodos heurísticos o basados en reglas. Los trabajadores de la IA. 1 utilizan a menudo enfoques ingenieriles y científicos tradicionales. Estudian el problema, sus condicionantes físicos y lógicos, y escriben un programa que no intenta imitar el modo en que las personas dicen que resuelven el problema.

La aportación hipotética de la IA al software de dirección de combate

He visto algunos trabajos sobresalientes en IA.1. Desgraciadamente, no puedo identificar un "corpus" de técnicas o tecnologías que sea exclusivo de este campo. Cuando se estudian estos programas de IA.1, se descubre que utilizan claramente enfoques científicos que también se utilizan en trabajos que no reciben la calificación de IA. La mayor parte del trabajo es específica para el problema en cuestión, y se precisa cierta abstracción y creatividad para transferirlo. Se habla de la IA como de una fuente mágica de nuevas ideas. Hay buenos trabajos en la IA.1, pero nada tan mágico que permita resolver el problema de la dirección de combate de la IDE.

En mi opinión los enfoques tomados en la IA.2 son peligrosos y, gran parte del trabajo, equívoco. Las reglas que se sacan del estudio de las personas, resultan ser inconsistentes, incompletas e imprecisas. Los programas heurísticos se desarrollan mediante un proceso de prueba y error en el que se añade una nueva regla cuando se detecta un caso no cubierto por las antiguas. Este método normalmente da lugar a programas cuyo comportamiento es mal comprendido y difícil de predecir. Los investigadores en IA.2 aceptan este método evolutivo de programación como normal y apropiado. Yo confío en este tipo de programas incluso menos de lo que lo hago en los programas no estructurados convencionales. No se sabe nunca cuando va a fallar el programa.

En ocasiones he tenido que examinar de cerca los resultados de trabajadores de la IA.2, y siempre he resultado defraudado. Vista de cerca, la heurística llegaba a cubrir un pequeño número de casos obvios, pero no funcionaba en general. (...)

Sobre los sistemas expertos

Ultimamente se ha oído hablar mucho del éxito de una clase particular de sistemas basados en reglas conocidos como sistemas expertos. Todas las discusiones citan un ejemplo de tales sistemas, utilizado para resolver problemas reales por personas ajenas a su desarrollo. Ese ejemplo es siempre el mismo -un programa diseñado para configurar computadores VAX. Para muchos de nosotros, este no parece un problema difícil: parece el tipo de problemas que se presta a una solución algorítmica, porque los sistemas VAX se construyen con componentes bien entendidos y bien diseñados. Recientemente he leído un artículo que señalaba que el mantenimiento de este programa se había convertido en una pesadilla. Se comprendía deficientemente, estaba mal estructurado y por ello, era difícil de cambiar. Tengo buenas razones para creer que podría reemplazarse por un programa mejor, escrito utilizando unas buenas técnicas de ingeniería software en lugar de técnicas heurísticas.

La IDE representa un problema que puede ser más difícil que los abordados por la IA.I y los sistemas expertos. Los trabajadores en esas áreas atacan problemas que ahora requieren expertos humanos. Algunos de los problemas de la IDE aparecen en áreas donde actualmente no tenemos expertos humanos. De hecho ¿hay actualmente personas que pueden observar, con un alto nivel de fiabilidad y confianza, misiles en vuelo balístico y distinguir las cabezas nucleares de los señuelos?.

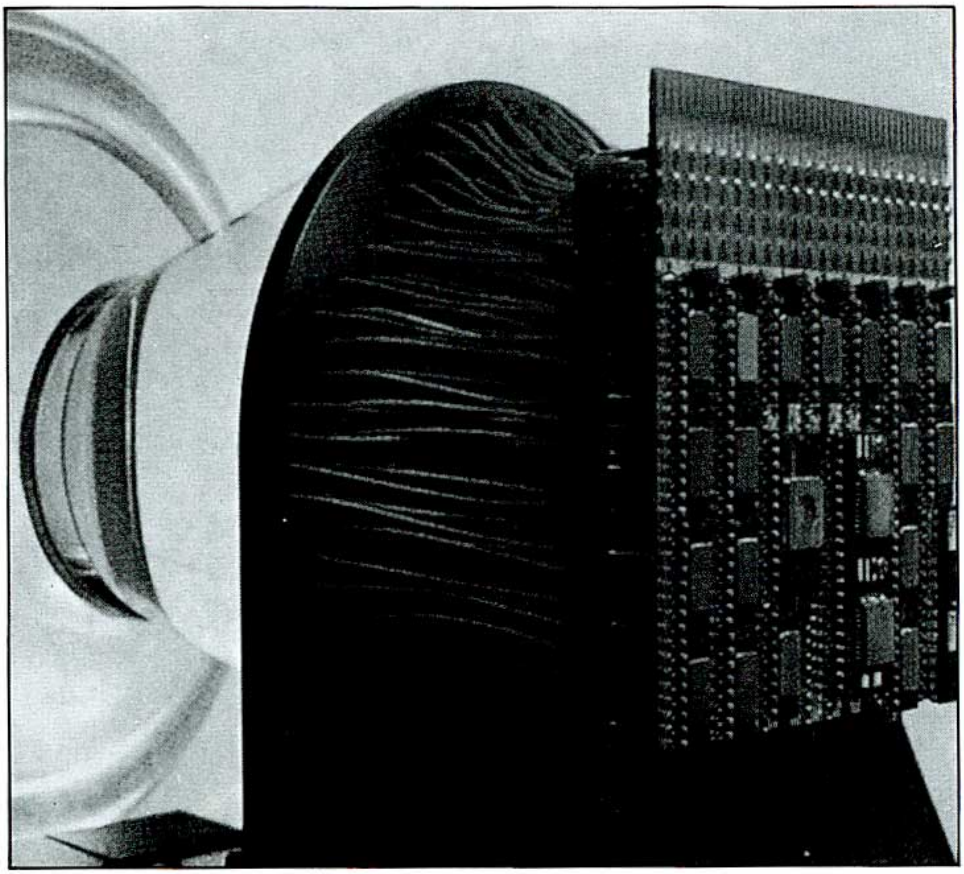
Conclusión

La IA tiene respecto a la inteligencia la misma relación que las flores artificiales respecto a las flores. A distancia pueden parecerse, pero de cerca son completamente diferentes. No creo que se pueda aprender mucho sobre las unas estudiando las otras. La IA no ofrece ninguna tecnología mágica para resolver nuestro problema. Las técnicas heurísticas no dan lugar a sistemas en los que se pueda confiar.

¿RESOLVERA LA PROGRAMACION AUTOMATICA EL PROBLEMA DEL SOFTWARE DE LA IDE?

A lo largo de mi carrera en informática he oído decir muchas veces que la solución al problema del software es la programación automática. Todo lo que hay que hacer es escribir las especificaciones para el software, y el computador encontrará un programa. ¿Se puede esperar que tal tecnología produzca programas fiables para la IDE? (...)

La programación automática siempre ha sido un eufemismo sustituyendo a programación en un lenguaje de programación de más alto nivel que el disponible por el programador. La investigación en programación automática es simplemente una investigación sobre la implementación de lenguajes de programación de más alto nivel.



Sensor óptico a bordo de un satélite.

La programación automática es factible. Se sabe desde años que se pueden implementar lenguajes de programación de más alto nivel. La única cuestión real era la eficiencia de los programas resultantes. Usualmente, si la entrada "especificación" no es una descripción de un algoritmo, el programa resultante es tremendamente ineficiente. No creo que el uso de especificaciones no algorítmicas como lenguaje de programación sea práctico para sistemas con capacidad computacional limitada y con exigencias duras de tiempo real. Cuando la entrada especificación es una descripción de un algoritmo, escribir la especificación es realmente escribir un programa. No habrá cambios sustanciales respecto a nuestra capacidad actual.

La fiabilidad de la programación automática

El uso de lenguajes mejorados ha dado lugar a una reducción en la cantidad de detalles que un programador debe tener en cuenta, y por ello, a una mejora en la fiabilidad. Sin embargo, los lenguajes de programación existentes, aunque no son perfectos, tampoco son tan malos. A menos que se utilicen especificaciones no algorítmicas como entrada a estos sistemas, no espero una mejora.

De otro lado, nuestra experiencia al escribir especificaciones no algorítmicas ha mostrado que se cometen errores del mismo modo que se cometen al escribir algoritmos. El efecto de estos trabajos sobre la fiabilidad no está claro todavia.

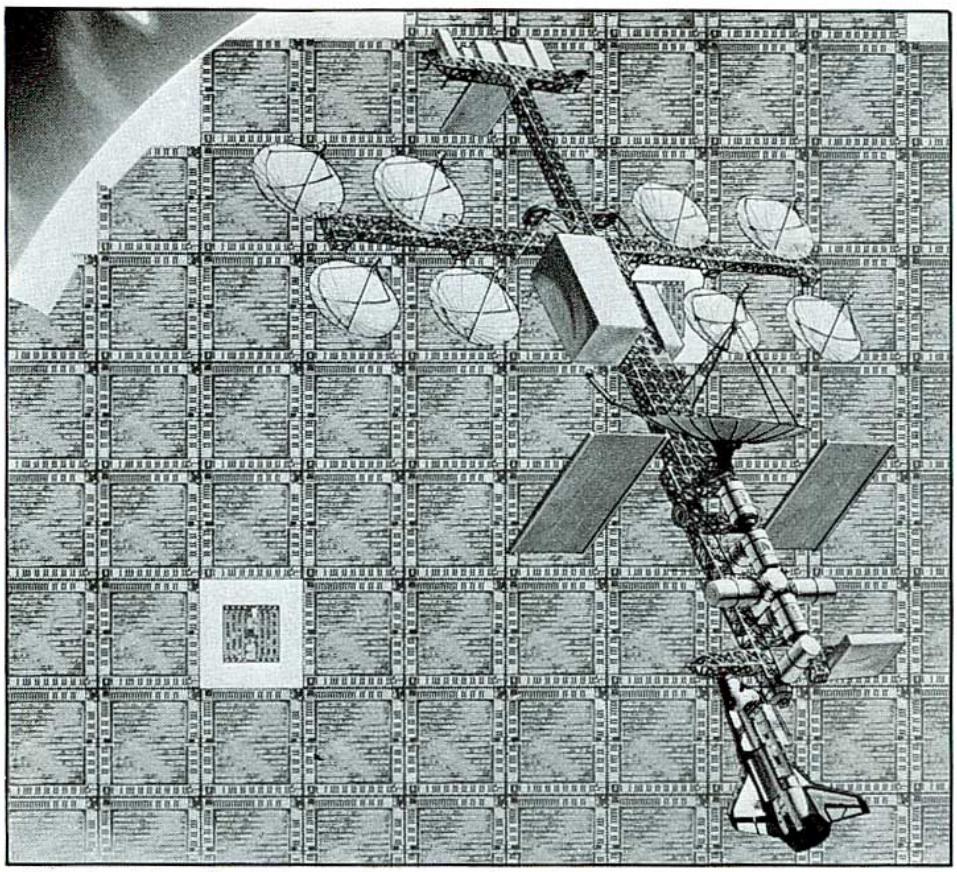
Conclusión

Creo que las declaraciones que se han hecho sobre sistemas de programación automática son muy exageradas.

La programación automática de un tipo que sea sustancialmente diferente de lo que hacemos ahora no es probable que llegue a ser una herramienta práctica para sistemas en tiempo real como el sistema de dirección de combate de la IDE. Además, uno de los problemas básicos de la IDE es que no tenemos la información para escribir especificaciones en que se pueda confiar. En tal situación, la programación automática no es ninguna ayuda.

LA VERIFICACION DE PROGRAMAS Y LA FIABILIDAD DEL SOFTWARE DE LA IDE

Los programas son objetos matemáticos. Tienen significados que son objetos matemáticos. Las especificaciones de un programa son objetos matemáticos. ¿No se debería poder probar que un programa se ajusta a su especificación?. Este ha sido un tema de investigación desde hace más de 25 años. Si podemos probar la corrección de



IDE: Informática y Electrónica para la guerra en el espacio.

un programa, ¿no podríamos probar la corrección del software de la IDE? ¿Y si consiguiésemos probar esta corrección, no se podría confiar en él en caso de necesidad?

¿Qué se puede probar?

Puede probarse que ciertos programas pequeños en lenguajes de programación especiales se ajustan a una especificación. La palabra pequeño es relativa. Los que trabajan en verificación considerarían grande un programa de 500 líneas. Al discutir el software de la IDE, en cambio, un programa tal se consideraría pequeño. Los programas cuyas pruebas he visto eran bastante más pequeños que 500 líneas, y realizaban tareas matemáticas bien definidas. Además, se habían escrito sin el recurso a efectos colaterales ("side-effects"), una herramienta importante en los programas prácticos.

Las pruebas de programas tales como un modelo del campo gravitatorio terrestre no tienen estas propiedades. Esos programas son mayores; sus especificaciones no son tan limpias o formalizables matemáticamente. A menudo están escritas en lenguajes de programación cuya semántica es difícil de formalizar. Yo no he visto pruebas para este tipo de programas.

Y no sólo son las pruebas manuales las que están limitadas a programas de pequeño tamaño con especificaciones matemáticas: los probadores y verificadores automáticos de teoremas también están estrictamente limitados respecto al tamaño de los programas que pueden tratar. Este es varios órdenes de magnitud diferente del tamaño de los programas que constituirían el sistema de dirección de combate de la IDE.

Las especificaciones

En el caso de la IDE no tenemos las especificaciones contra las que aplicar una prueba. Aunque el tamaño no fuera un problema, la falta de especificaciones eliminaría todo el sentido de una prueba formal. Si escribiésemos una especificación formal para el software, no tendríamos modo de probar que un programa que satisficiese la especificación realmente haría lo que se esperaba que hiciese. La especificación en sí misma podría ser errónea o incompleta.

¿Podemos tener fe en las pruebas?

Las pruebas aumentan nuestra confianza en un programa, pero no proporcionan una base para una confianza completa. Incluso en matemática pura, hay muchos casos de pruebas que se publicaron con errores. Las pruebas tienden a ser fiables cuando son pequeñas, claras y revisadas cuidadosamente. No son fiables cuando son grandes, complejas y no leídas por nadie aparte del autor. Esto es lo que ocurriría con cualquier intento de probar la corrección incluso de una porción del software de la IDE.

La concurrencia

Las técnicas de prueba más prácticas están restringidas a programas secuenciales. Los trabajos recientes sobre prueba de sistemas de procesos concurrentes se han centrado en protocolos con paso de mensajes más que en procesos que cooperan utilizando memoria compartida. Algunas técnicas pueden aplicarse a la memoria compartida, pero son más difíciles que las pruebas de programas secuenciales o que las pruebas de programas cuya comunicación se limita a canales de mensajes.

Programas supuestamente robustos

Uno de los principales problemas del software de la IDE es que debería funcionar con parte de sus equipos destruidos o fuera de funcionamiento por la acción del enemigo. Llevo 20 años viendo intentos de probar la corrección de programas, y sólo he visto un intento de probar que un programa proporcionaría una respuesta correcta en una situación de fallo hardware. En esta prueba se tomaron hipótesis extremadamente irreales. No tenemos técnicas para probar la corrección de programas en presencia de fallos hardware desconocidos y errores en los datos de entrada.

Conclusión

Me resulta inconcebible que alguien pueda proporcionar una prueba convincente de la corrección, incluso de una pequeña porción del software de la IDE. Dada nuestra incapacidad para especificar los requisitos del software, desconozco lo que significaría tal prueba si se hiciese. ■

David L. Parnas, *es profesor de informática de la Victoria University, en la Colombia Británica (Canadá), consultor durante muchos años de la Oficina de Investigación Naval, en programas militares y experto mundialmente conocido en ingeniería software.*

Nota: Este artículo es un extracto de otro del mismo título aparecido en "Communications of the ACM". Diciembre 1985.