NYHETSBLAD FÖR SESAM
Försvarssektorns Användargrupp för Software Engineering

RENDEZVOUS

Nr 1 januari 2004

Innehåll

Vad är SESAM?

SESAM är ett samverkansnätverk för projektövergripande och företagsneutral kunskapsuppbyggnad och kunskapsspridning inom området programvaruintensiva försvarssystem.

SESAM skall genom organiserat samarbete mellan dess medlemmar (företag, organisationer, myndigheter och utbildningsinstitutioner) främja tillförlitlighet och effektivitet i utveckling och vidmakthållande av programvaruintensiva försvarssystem.

SESAMs verksamhet att samla, skapa och sprida information och kunskap sker huvudsakligen i arbetsgrupper som verkar inom avgränsade tekniska områden och som efter behov inrättas och avvecklas.

SESAM anpassar, profilerar och förnyar sin verksamhet med hänsyn till ändrade tekniska och andra omständigheter av betydelse för intresseområdet.

När SESAM startade sin verksamhet 1988 var s k inbyggda realtidssystem (i farkoster, sensorer, stridsledningssystem m fl) dominerande inom försvaret och svensk försvarsindustri. Det fanns ett allmänt behov att kunna driva utvecklingen av programvara för dessa alltmer komplexa system på ett mer organiserat sätt, d v s att mer konsekvent praktisera vad som börjat benämnas Software Engineering. SESAMs verksamhet inriktades därför från början på användningen av det speciellt för inbyggda och realtidssystem under början av 80-talet i brett internationellt samarbete framtagna och av ISO 1987 standardiserade programspråket Ada, vilket hade utformats särskilt för att stödja tillämpning av Software Engineering principer. Den fortsatta användningen av Ada, inkl i nya (Ada 95) och framtida versioner, är fortfarande av stort intresse och betydelse för många av medlemmarna i SESAM.

Efterhand har även andra typer av programvarusystem blivit mer förekommande inom försvarssektorn, t ex på informationssystemområdet. Utvecklingen inom kommunikationstekniken, t ex via Internet snabba expansion, har också tillfört nya möjligheter och typer av problemställningar för programvaruutvecklingen. Utbudet av kommersiellt tillgänglig programvara (COTS) som man önskar kunna använda i försvarssystem har också ökat. Inriktningen på uppbyggnad av ett nätverksbaserat försvar, behovet av interoperabilitet vid internationella insatser m m har aktualiserat nya problemställningar i systemutformningen. Denna utveckling gör att nya tekniker, processer och metoder, språk och andra hjälpmedel etc för framtagning och vidmakthållande av programvaruintensiva system av intresse att behandla inom SESAMs verksamhet ständigt tillkommer.

SESAM styrs av ett Råd med representanter för gruppens medlemmar. Rådet har till sin hjälp ett Verkställande Utskott (VU) och ett sekretariat.

Rådets ordförande är Claes Wadsten, AerotechTelub, tel 013-231652 .

VU

Andersson Tommy, Ericsson Microwave Systems AB, tommy.andersson@emw.ericsson.se

Carlsson Ingemar, adjungerad, ingemar.carlsson@bredband.net

Ekman Mats, Saab Aerospace AB, mats.ekman@saab.se

Hallén Johan, FMV, johan.hallen@fmv.se

Merkell Curt, Saab Bofors Dynamics AB, curt.merkell@dynamics.saab.se

Sporre Lena, FOI, lena.sporre@foi.se

Wadsten Claes, AerotechTelub AB, claes.wadsten@aerotechtelub.se

Wåhlén Ulf, FMV, ulf.wahlen@fmv.se

Wååk Martin, FMV, martin.waak@fmv.se

Arbetet utförs i två arbetsgrupper och en intressegrupp:

Ag Metodik

Håkan Edler, CTH/Datorteknik, edler@hisafe.se

Ag Teknik

Lars Asplund, Mälardalens högskola, lars.asplund@mdh.se

Ig Programvarusäkerhet

Inga-Lill Bratteby-Ribbing, FMV, inga-lill.bratteby-ribbing@fmv.se

Vilka kan vara med i SESAM?

Medlemmarna i SESAM är företag, organisationer och myndigheter (förvaltningar, utbildningsinstitutioner etc) i Sverige med anknytning till försvarssektorn. Medlemmarna indelas i följande kategorier
- ordinarie medlemmar
- arbetsgruppsmedlemmar
- informationsmedlemmar.

Enskild person kan endast komma ifråga som informationsmedlem.

Inträde i SESAM

För samtliga medlemskategorier gäller att inträde beslutas av Rådet. För inträde som ordinarie- eller arbetsgruppsmedlem krävs status som leverantör till FMV eller FM. Dessutom krävs en skriftlig förbindelse att uppfylla åtagande som ordinarie- resp arbetsgruppsmedlem. För inträde som informationsmedlem (erhåller endast informationsbladet) krävs status som leverantör till FMV eller FM eller status som myndighet inom totalförsvaret. Rådet kan emellertid anta annan part som informationsmedlem.

För ansökan om medlemskap i SESAM vänd er till sekretariatet.

SESAM-Sekretariatet
c/o Kåsjös Kontor, Anna Kåsjö
Odengatan 28, 4 tr
113 51 STOCKHOLM
Tel: 08-510 51866, 070-716 9702
Fax: 08-510 51932
E-post: kasjos.kontor@bredband.net

Ordförande har ordet

Som ordförande kan jag nu blicka tillbaka på ännu ett genomfört år.

År 2003 har visat att SESAM är ett nätverk av företag och myndigheter med aktiviteter som berör många. Då jag själv reser mycket inom svensk försvarsindustri och liknade företag så finner jag att SESAM idag är ganska känt. En stor orsak till detta är de senaste årens seminarier som varit mycket välbesökta. Utvecklingen har varit så positiv att vi de senaste åren fått hyra större lokaler för att kunna ta emot alla som är intresserade. Jag ser nu att vi nått en storlek på seminarierna med avseende på antalet deltagare att vi inte behöver öka detta. En fortsatt ökning kommer att kräva en helt annan organisation. Det är nu viktigare att vi kan genomföra seminarier med intressanta teman så att vi kan bibehålla nuvarande storlek.

Seminarierna, tidningen Rendezvous samt våra arbetsgrupper har bidragit till att marknadsföra SESAM på ett positivt sätt. Idag är de flesta företagen i Sverige som levererar datorsystem och programvara till svenska försvaret medlemmar i SESAM. Jag vet också att några fler företag funderar på att söka medlemskap. Detta är mycket positivt.

Vår satsning på ytterligare en arbetsgrupp för programvarusäkerhet har varit mycket lyckad. Denna arbetsgrupp har varit mycket aktiv med många projekt och aktiviteter som aktiverat medlemmarna. Jag vill passa på att framföra ett tack till Inga-Lill Bratteby-Ribbing som har stor del till att gruppen varit mycket framgångsrik.

Planeringen för 2004 liknar det som varit under 2003 varför jag med tillförsikt tror att vi kommer att se ytterligare ett lyckat år. Med de tre arbetsgrupperna ”Metodik”, ”Teknik” samt ”Programvarusäkerhet” bör vi kunna få aktiviteter inom alla intressanta områden som kan beröra programvara, datorsystem och säkerhet i militära system. Aktiviteterna i SESAM blir dock inte bättre än vad medlemmarna själva gör varför jag hoppas att våra medlemmar bidrar med ”sitt strå till stacken”.

Jag vill med detta tacka alla för ett bra 2003 och hoppas på ett lika bra 2004.

Claes Wadsten
Ordförande i SESAM

Tfn. dir : 013-23 16 52
Mobil: 070-6000271
E-mail: claes.wadsten@aerotechtelub.se

Redaktionen beklagar och hoppas på läsarna

Tyvärr har arbetet med Rendezvous blivit lidande av hög belastning hos redaktionsmedlemmarna med andra göromål under året. I fortsättningen räknar vi med att kunna prestera bättre.

Nu liksom tidigare efterlyser vi bidrag från medlemmarna. Det kan gälla artiklar om intressanta projekt och produkter, eller debattinlägg om aktuella spörsmål, samt inte minst referat från de stora Software Engineering och Ada konferenserna. Det svenska deltagandet vid sådana tillfällen verkar ha avtagit de senaste åren. Desto viktigare blir det därför att de som verkligen får tillfälle att åka på sådana begivenheter försöker ta sig lite tid att skriva ner korta noteringar om intryck o d man fått, som kan delges en större krets via Rendezvous.

Red.

Ivar Jacobson får Gustav Dalén medaljen och blir Cyber Ivar

Ivar Jacobson, världskänd programvaruguru, har fått årets Gustav Dalén medalj av Chalmerska Ingenjörsföreningen för "sina banbrytande metoder för att utveckla programvarusystem för datorer".

Ivar som förstås är välkänd i SESAM-kretsen, började tillämpa sina idéer om industriell systemutveckling vid konstruktion av stora programvarusystem på Ericssons telefonväxlar mot slutet av 60-talet. 1987 grundade han företaget Objectory, som senare uppgick i Rational och som numera är en del av IBM.

Ivar är bl a upphovsmannen till Use Cases och utgjorde tillsammans med Grady Booch och James Rumbaugh de s k Tre Amigos som låg bakom framgångarna för Rational Unified Process. Ivar är också en av upphovsmännen till UML- Unified Modeling Language som blev en OMG-standard.

Tillsammans med dottern Agneta startade Ivar Jacobson för tre år sedan ett nytt bolag, Jaczone AB. Målet var att ta fram en produkt kallad WayPointer som gör programvaruprocessen med UML och RUP exekverbar. WayPointer övervakar kontinuerligt status hos UML-modellen med avseende på fullständighet, konsistens och korrekthet och baserat härpå, och de övergripande mål för systemet som användaren definierat, ger verktyget tips om hur utvecklingen bäst bör fortsätta. Syftet är att göra det roligt och lätt att bygga UML-modeller och tillämpa RUP, vilket bör öka chansen att de används på avsett sätt. WayPointer finns i ett antal olika konfigurationer för resp Use Cases, Architecture, J2EE plattform och Enterprise användning av programutvecklare, programvaruarkitekter och projektledare.

Jaczone utvecklar också Cyber Ivar, en chatterbot, eller ett slags agent, som utvecklaren kan chatta med och få råd av. Cyber Ivar avses med tiden komma att veta det mesta om programvaruutveckling och speciellt UML och utvecklingsprocesser.

Ett tillfälle att träffa den verklige Ivar var vid INCOSE Sveriges seminarium i Stockholm den 17/11 där han för ett mycket intresserat auditorium under rubriken "Use Cases & Systems of Systems" sammanfattade hela historiken från AXE för 30 år sedan och vad han lärt från den tiden och hur det påverkat hans fortsatta gärning över Objectory, Rational och "de tre Amigos", IBM till dagens verksamhet med Jaczone.

Vi är mycket glada att genom tillmötesgående från Ivar Jacobson och INCOSE Sverige kunna inkludera presentationsmaterialet från Ivars föredrag på SESAM CD2003.

Vi återger här ett kort referat av Göran Backlund, Combitech Systems, av Ivars föredrag vid INCOSE-seminariet.

Ivar Jacobson om Use Cases och System of Systems; intryck

Grundtankarna i UML (fastställdes 1997) användes på Ericsson redan 1972 under AXE-utvecklingen. Komponenter fanns då definierade med både sw och hw – det finns ännu inte i UML. Sekvensdiagram fanns också – men med alternativa sekvenser – som kommer nu i UML 2.0.

På Ericsson fick jag konstruera de första komponenterna själv – det gick inte att förklara tanken för andra, jag var tvungen att visa själv.

Ivar Jacobson, IBM Rational Software

Innan man bygger systemen – ta reda på vilka användarna är och vilka behov de har (definiera uppgiften).

UML används även för att modellera affärsprocesser; men en affärsprocess är en synonym till affärsanvändningsfall.

”Det viktiga är inte diagrammet – utan att tankarna hängs upp på detta”.

Om Aspectoriented Programming; användningsfall kommer att kunna utskiljas och generera kod, GUI och testfall – allt inom ett och samma användningsfall.

När Ivar introducerade komponentbegreppet på Ericsson för 30 år sedan möttes han av kommentaren ”Vi har inte tid att tänka igenom gränssnitten innan vi börjar programmera”. Idag anses ett sådant uttalande befängt. (Vi kan alltså vända trender och självklara sanningar – det är inte för sent!).

Vid ett tillfälle krävdes en omstart – det tog 6 månader att omstrukturera innan vi kunde gå vidare.

Det modernaste språket idag – Java – är ett 60-talsspråk, möjligen ett 70-talsspråk. Simula är det äldsta objektorienterade språket, det skapades på 60-talet i Norge. C++ skapades av Bjarne Strostrup i Danmark. (C++ är en arvtagare till Simula). Linux skapades av Linus Torvalds – en finlandssvensk. ”Not everything important in software was made in the US. On the other hand – if you want to make money of it you have to move to the US”.

Bo Sandén känd för många SESAM-medlemmar är sedan några år professor i Computer Science vid Colorado Technical University, i Colorado Springs. Bo är en känd expert på bl a realtidsprogrammering. Vi har här nöjet att återge ett bidrag från Bo inom ett mycket aktuellt ämne.

Real-time programming safety in Java and Ada

Bo I. Sandén
Computer Science
Colorado Technical University
4435 N. Chestnut St.
Colorado Springs, CO 80907-3896
U.S.A.
Email: bsanden@acm.org
Phone: (719) 590-6733
http://classweb.coloradotech.edu/bsanden

This article was first published in the Ada User Journal 23:2 (June 2002), pages 105-113 and was reprinted in Ada Letters XXIII:2 (June 2003) pages 32-46.

Abstract

Both Java and the Real-time Specification for Java contain concurrency-related constructs that are easily abused or simply misunderstood by a programmer without sufficient knowledge of multithreading. This article lists a number of those constructs and shows how they are avoided in Ada. Many of the mistakes arise when a programmer confuses exclusion synchronization and condition synchronization. The article opens with an explanation of those concepts.

1 Introduction

Few industry-strength languages include multithreading in their syntax. Among contemporary languages, Ada [1, 2] and Java [3-8] are the most prominent. The philosophy of concurrency is similar in Java and Ada[1] and is based on the classic distinction between threads (tasks in Ada) on the one hand, and shared objects on the other. This has been the dominant paradigm for practical multithreading for decades, although other models exist, such as the rendezvous paradigm of Ada83, which is still supported in Ada95. (The Ada83 paradigm is not further discussed here.) In Ada, shared objects are declared `protected`. In Java, they are instances of classes that have methods marked `synchronized`.

Although the philosophy of concurrency is similar, the attitude to safety and reliability is radically different in Ada and Java. Java threading is adequate for its original purpose: windows programming and applets. But Java is now being fitted with real-time extensions and may be applied to safety critical software. The language has many potentially abusable constructs and programmer pitfalls. This is certainly true for Java in general [9] but is particularly important with concurrent software, which is notoriously difficult to debug.

The Real-Time Specification for Java (RTSJ [10, 11]) does nothing to remove the pitfalls. It intends to

make Java useful for real-time applications by circumventing garbage collection and providing interrupt handling, not to make the language less error prone. In fact, using the new classes that represent different types of memory areas or the new construct for asynchronous transfer of control correctly is far from easy. The case that Ada is a much safer language for real-time embedded applications than Java can easily be made.

This article is intended for Ada programmers who may be taking the Ada concurrency features for granted. The purpose is to view those features against a backdrop of the pitfalls of a more traditional concurrency implementation without emphasis on safety. I point out a number of Java threading pitfalls and note how they are prevented in Ada. While I briefly summarize the Ada tasking model, the reader is assumed to have an understanding of Ada tasking and sufficient understanding of Java to be able to read small program excerpts, but is not expected to know much about Java multithreading. For a comprehensive comparison of concurrency features in the two languages, see [12].

1.1 Forms of synchronization

The traditional concurrency model with threads on the one hand and shared objects on the other relies on a distinction between *exclusion synchronization* and *condition synchronization*, described as follows.

> Exclusion synchronization is used to stop two threads from operating on the same object at the same time and thereby jeopardizing the integrity of its data. Java provides exclusion synchronization for any synchronized method. Ada provides exclusion synchronization for protected operations. A block of code that is executed under exclusion synchronization is called a *critical section*. It is bracketed by instructions that acquire and release a lock on an object.
>
> Each thread is expected to maintain exclusive access for a very short time, making it unlikely that a thread will ever find an object locked. If it does find an object locked, only a brief wait should be expected. It is even more unlikely that two threads should attempt access to the same object while its is locked. For this reason, one need not be concerned with maintaining a orderly queue of threads pending on an object lock. Implementations of exclusion synchronization typically let a thread that encounters a locked object yield the processor in the hope that it will find the object unlocked when next made running. If the object is still locked, the thread again yields the processor. With multiple processors, one often uses a *spin lock*, that is, the thread enters a loop where it repeatedly attempts access until it is successful. I shall use the term "spin lock" for both the single processor and multiprocessor cases.
>
> While a thread, `l`, is operating on a shared object `o`, under exclusion synchronization, it may be preempted by a higher-priority thread, `h`, which also needs exclusive access to `o`. Unavoidably, `h` must wait for `l` to exit the critical section. Such a situation where a higher priority thread is waiting for a lower priority thread is referred to as *priority inversion*. If `l` continues executing at its normal priority, a thread, `i`, of intermediate priority may preempt `l`. This leads to an avoidable situation where `h` is waiting for two lower-priority threads. To avoid it, `l` can be given a priority boost. One possibility is to let `l` *inherit* `h`'s priority once `h` tries to access `o`. The other possibility is to define a *ceiling priority* for `o`, which is used by any thread while it executes a synchronized method on `o`. The ceiling priority must be as high as the priority of any thread that ever operates on `o`.
>
> Condition synchronization is when a thread cannot proceed if a certain condition holds. A buffer, shared by two or more threads provides a classic example. A `Buffer` instance has the operations `put`, called by producer threads, and `get`, called by consumer threads. A thread that calls `put` must wait if the buffer is full, and a thread calling `get` must wait if it is empty.

There is no assumption that this wait will be brief. Threads may spend considerable time waiting, and must be queued, typically first-in-first-out per priority. A thread conditionally waiting for an object never holds the object locked and should not normally hold other objects locked.

Condition synchronization must be used to control access to any shared resource that is held long enough for a queue of waiting threads to form. Examples of resources of this nature range from a printer or a database record to resources in the problem domain of a control system such as railroad segments and automated vehicles in a flexible manufacturing system. Access to such a resource is handled by means of an object with a Boolean variable `busy`, say, and operations such as `acquire` and `release`. Once a thread has successfully acquired the resource and set `busy` to `true`, it releases the object lock, allowing other threads to call `acquire` and place themselves on queue.

The term "condition synchronization" gives no hint that condition synchronization is often used to control mutually exclusive access to domain resources and, in general, to resources held for a long time. The terms "competition synchronization" and "cooperation synchronization" for exclusion and condition synchronization respectively are no better [13]. The rationale here is that the producer and consumer threads must cooperate to manage an empty and a full buffer but compete over the access to the operations. Again, this ignores the use of condition synchronization to control exclusive access to a domain resource. An alternative way to characterize synchronization is to distinguish between exclusive access with *short extent* in time (exclusion synchronization) and *long extent* (condition synchronization) [14], but this plays down the conceptual difference between the two kinds of synchronization.

The distinction between exclusion and condition synchronization is crucial in real-time concurrent programming. While Ada and Java both support exclusion and condition synchronization, Ada helps the novice programmer by clearly separating the concepts syntactically. Java does not, and some of the pitfalls result from a confusion of them.

2 Ada concurrency model

Ada implements concurrency with two kinds of entities: *tasks* and *protected objects*. You define a task type, which is instantiated as any other type, or a singleton task. *Protected objects* have monitor-like behavior. They can have *protected operations* of three kinds: *functions*, *procedures* and *entries*. These are declared in the specification of the protected type[2] as for example the following:

```
protected type X is
    function F1 (...) return Type1;
    procedure P1 (...);
    entry E1 (...);
private
    --  Attribute variables
    --  Private operations including interrupt handlers
end X;
```

All protected operations have exclusion synchronization built in. The differences between protected functions, protected procedures and entries are as follows:

Protected functions are read-only. They are prohibited from changing the attribute values of the protected object and are subject to a read lock: Any number of function calls on a given object are allowed simultaneously, but not during a procedure or entry call on the object.

Protected procedures are allowed to change attribute values. They are subject to a write lock: Only one procedure (or entry) call at a time is allowed on a given object, and not during any function call.

Like procedures, *entries* are allowed to change attribute variable values and are subject to the write lock. In addition, an entry can provide condition synchronization by means of a *barrier condition*, which appears in the body of the protected type. An entry call only proceeds when the condition is true. For example, an entry Get, which is only allowed when the number (`num`) of items in a buffer is greater than zero, may be declared as follows:

```
entry Get (...) when num > 0 is ....
```

A task that calls `Get` when `num = 0` is put on a queue that is FIFO per priority. Each protected object has one queue per entry.

Any variables in the barrier condition are supposed to be attribute variables of the protected object, on which the entry operates. The values of those variables can only be changed by calls to protected procedures and entries on that object.

At the end of each procedure or entry call on a given object, its barriers are evaluated. If a barrier is found to be true, the most eligible task in the corresponding queue is activated and executes the entry body. Tasks that are already in a queue have precedence over new callers according to the principle of "internal progress first".

3 Java concurrency model

In Java, any method in any class can be declared `synchronized`. This is exclusion synchronization: A write lock per object is applied, so that only one synchronized method at a time can operate on a given object. A synchronized method in Java is similar to a protected procedure or entry in Ada in that it is implicitly bracketed by instructions that acquire and release the object lock.

Condition synchronization in Java relies on explicit tests programmed into the synchronized methods, as for example:

```
while (0 == num) {wait();}
```

If num is zero, a calling thread calls `wait` and thereby enters the object's *wait set*. There is one wait set per object, not one per entry per object as in Ada. A thread, `t`, that executes a synchronized method on an object may change the truth value of a condition that may affect one or more threads in the wait set. Before leaving the method, `t` must explicitly `notify` any such threads. The call `notifyAll` reactivates all threads waiting for conditional access to an object.

The Java thread model hides little from the programmer. This makes it quite flexible for the old hand at concurrency. Apart from the tie-in with object-orientation, the thread model is in fact very similar to what I first encountered when manipulating threads provided by the UNIVAC 494 operating system in assembler programs 30 years ago. In a sense, this makes Java more pedagogical than Ada because it exposes the details of synchronization. But by the same token, the Java model is much more error prone. Very little protects Java programmers from the consequences of their own mistakes. Unfortunately, many programmers are not shy about trying things they don't fully understand and then testing the program to see if it works. Many concurrency related bugs are subtle enough to pass most tests. An Ada programmer cannot easily make clerical errors with unintended effects on the program

behavior.

An advantage of the Java model is that it can be implemented with less overhead than the Ada model, but this issue appears to be losing much of its earlier importance. Further, a Java class with synchronized operations can be part of the inheritance hierarchy. Because a thread can hold multiple locks on the same object, a synchronized method can easily call a synchronized method in a parent class, including one that it overrides. If the parent method has a wait loop, the thread may enter the wait set.

In Ada, a protected type is not extensible, that is, it cannot be subclassed. Although Ada includes object-oriented features including inheritance, polymorphism and dynamic binding, these were not extended to protected objects [15].

3.1 Real-time Java

The Real-Time Specification for Java (RTSJ) [10, 11] is an effort to make Java useful for real-time programming. (An alternative specification is given in [16].) One premise is that a real-time program must be predictable so that the programmer can determine *a priori* when certain events will occur. This is not true for standard Java for a number of reasons. First, the garbage collector, which can interrupt any other processing, adds an element of randomness. Second, different scheduling policies cannot be imposed in standard Java. (Scheduling policies such as the rate-monotonic algorithm allow you to prove that a set of threads meet their specified deadlines.) Third, in standard Java, threads placed in a wait set are reactivated in arbitrary order, independent of when they attempted access; the wait set is not a FIFO queue.

To deal with these problems, RTSJ introduces a number of new classes, most of which are necessary for working around the garbage collector. One such class is `NoHeapRealtimeThread` (NHRT), which is a descendant of `Thread`. *NHRT* threads have higher priority than the garbage collector so are not subject to arbitrary delays. This places many restrictions on the programmer, however. For example, an *NHRT* thread cannot allocate objects on the heap. Instead, RTSJ provides for various kinds of special memory areas.

RTSJ also stipulates that threads in a wait set must be kept in FIFO order within priorities. This means that `notify` reactivates the thread with the highest priority. If there are more than one thread with that priority, the one that has waited the longest is reactivated.

RTSJ uses priority inheritance as the default control policy to address priority inversion. A priority ceiling protocol is also specified. Finally, to further support real-time programming, RTSJ allows the programmer to specify interrupt handlers.

4 Java pitfalls and their Ada solutions

Apart from the extensions provided by RTSJ, real-time Java relies on the threading and synchronization models of standard Java. Next, I discuss in more detail some features of these models from a real-time point of view, focusing on the associated pitfalls. I also discuss how each pitfall is prevented by the Ada syntax and semantics. Although programming to RTSJ is in many respects quite involved, I do not address any programming pitfalls that may appear there.

4.1 Defining and starting threads

Java provides the abstract class `Thread`, whose method `run` represents the logic that a thread performs. It corresponds to the executable part of a task body. A standard way of creating threads is to

declare a new class, `T`, that extends `Thread` and overrides `run` with appropriate processing. Each instance `to` of `T` has its own thread, which is explicitly started by means of the call `to.start()`. Once started, the thread executes `T`'s `run` method and has access to `to`'s data.

Because Java has no multiple inheritance, an additional mechanism is necessary for the case where a class, `R`, that needs a thread, already extends another class, such as `Applet`. For this situation, Java provides the interface `Runnable`. The programmer makes `R` extend `Applet` and implement `Runnable`. Instantiating `R` creates a *runnable object*, `ro`, say. To associate a thread with `ro`, you submit `ro` as an argument to one of `Thread`'s constructors and then call `start` on the resulting `Thread` instance. This is typically done in a statement such as:

```
new Thread(ro).start();
```

Java pitfalls. Once you have a class `R` that implements `Runnable`, Java gives you two ways to create multiple threads that execute `R`'s `run` method. First, you can instantiate `R` *n* times, and submit each instance once as a parameter to one of `Thread`'s constructors. Now you have *n* instances of `R`, each with a thread, so each thread has its own set of instance variables. But Java also lets you submit the same instance of `R` repeatedly to `Thread`'s constructor. The result is a subtly different case where multiple threads are tied to one object and share its instance variables. Two (or more) of these threads can directly manipulate those instance variables simultaneously, and possibly introduce inconsistencies.

Ada. Ada's model for defining and starting tasks is cleaner. You declare a task type, which is instantiated as any other type, or a singleton task. The task is started automatically, either when the first executable statement is reached after the declarations, or, if a task type is dynamically instantiated, immediately upon instantiation. Each task instance has its own private data.

4.2 Synchronized objects

Java provides all objects with the potential for monitor-like behavior, that is, approximately the behavior of a protected object in Ada. Every object has a lock variable, which is hidden from the programmer and cannot be accessed from methods on the object. Exclusion synchronization is accomplished by specifying a method as `synchronized`, as in:

```
void synchronized m() ...
```

When a synchronized method is called on some object, `o`, its code is implicitly bracketed by statements that acquire and release the lock on `o`. That is, a thread calling `o.m()` locks `o` as a whole, so that no other thread can perform any synchronized method on `o` (or execute any block synchronized with respect to `o` as discussed below). This synchronization feature is built in, which guarantees that the lock is always released when a thread leaves a synchronized method, even if this happens by means of the exception handling mechanism. I shall refer to any instance of a class that has at least one synchronized method or synchronized block as a *synchronized object*.

A Java programmer can choose to specify some but not all the methods of a class as synchronized. This has some useful applications. For example, a method that returns a constant or the value of a single attribute need not be synchronized.

Java pitfalls. The freedom to specify selected methods of a class as synchronized opens the door for mistakes. In the buffer example, the programmer may declare `get` synchronized and not `put`. This allows different threads to call `put` simultaneously. These calls may also overlap with a call to `get`.

This jeopardizes the integrity of the buffer data structure. The program may still work much of the time, but will produce occasional errors, especially when run on a symmetric multiprocessor providing true parallelism. Such errors tend to be hard to find by testing – although more easily by inspection by an experienced thread programmer. Omitting the keyword synchronized altogether, for `put` as well as `get` would further exacerbate the situation.

A programmer must ensure that the instance variables that the synchronized methods operate on are private so that they cannot be directly accessed and changed by a method operating on some other object. Even if they are private, you must ensure that they are not changed by a static method defined for the class.

Ada. All operations on a protected object require either a read lock or a write lock. You cannot include a non-protected operation in a protected object. A protected function can be used to return constants, etc., as can an unsynchronized method in an otherwise synchronized Java class.

4.2.1 Synchronized blocks

In addition to synchronized methods, Java provides synchronized *blocks*, which have no Ada counterpart. Any block in any method can be synchronized with respect to an object – not necessarily the current instance of the class where the method appears – by means of the syntax:

```
synchronized (Expression) { /* Block B */ }
```

Expression must evaluate to a reference to some object, `vo` of class `V`, say. As for synchronized methods, exclusion synchronization is implicit, so `B`'s code is bracketed by statements to acquire and release the lock on `vo`. A synchronized method and a synchronized block are both critical sections.

Consider first the case where `B` is part of some method, `m`, on class `V` and is synchronized with respect to the current object as follows:

```
class V ...
{
    void m()
    {
            synchronized (this)
            {
                /* Block B*/
            }
    }
}
```

This design is an alternative to synchronizing `m` and can be used if only parts of `m` requires exclusive access. That way, two or more threads can simultaneously execute those parts of m that are outside `B`, so concurrency may be increased. An alternative design is to make `B` into a separate, synchronized method called from within `m`.

As mentioned, the block `B` in `m` can be synchronized with respect to any object, not only the current one. In the following excerpt, `B` is synchronized with respect to object `wo` of class `W`. This means that before entering the block `B`, the thread that called `m` in order to operate on an object of class `V` acquires the lock on object `wo` of `W`.

```
class V ...
{
    void m()
    {
        synchronized (wo)
        {
            /* Block B*/
        }
    }
}
```

Synchronized blocks are useful when different threads need exclusive access to some object in order to perform their own, particular operations on it. Such an object is sometimes a printer or a window to which many threads write their own tailored outputs such as log entries as in the following example:

```
synchronized (myPrinter)
{
    // series of statements producing output
}
```

In this case, it can be inconvenient to make every possible combination of output statements into a method for the printer class.

Java pitfalls. By synchronizing blocks with respect to some object you effectively create "distributed" methods that are not included with other instance methods in the class definition. From looking at a class definition, you cannot tell whether any blocks exist that are synchronized with respect to its instances. A class without synchronized methods in its definition may appear to a maintenance programmer as an unsynchronized class even though there are blocks synchronized with respect to its instances.

Ada has no equivalent to synchronized blocks. All operations on a protected object are specified in its declaration.

4.3 Condition synchronization

The most common idiom for condition synchronization in Java is the statement

```
while (cond) {wait();}
```

I shall refer to this statement as a *wait loop*. It makes the calling thread wait as long as `cond` holds. If `cond` is true, the thread calls `wait` and thereby places itself in the wait set of the current object, `o`, and releases `o`. The wait set contains all threads waiting for *conditional access* to `o`.

The wait loop syntax is somewhat complicated by the need to handle an `interrupted` exception that can be caught by a thread while it is in the wait set. This is possible because this particular exception is thrown by a different thread than the one that must catch it. Unless the exception is propagated to an enclosing scope, a construct such as the following is necessary:

```
while (cond) try {wait();}
    catch(InterruptedException e) {
          /* Take action or ignore the exception */
    }
```

Pitfalls. The wait loop is like an incantation that should always be repeated in almost exactly that form. For example, the variation

```
while (cond) {yield();}
```

stops the calling thread from proceeding against `cond` but does not release the object. This means that other threads that are supposed to change `cond` by calling synchronized methods on the object cannot do so.

A more insidious mistake is to replace the wait loop with the quite similar statement

```
if (cond) {wait();}
```

This statement makes the calling thread enter the wait set and release the object, but only once. When reactivated, the thread continues after the `wait` call and proceeds in the synchronized method even if `cond` is true [12]. This is particularly dangerous, since `notifyAll` must often be used and relies on the wait loop. When `notifyAll` activates a thread that is not to proceed, the thread is supposed to retest its condition and return to the wait set. Substituting `if` for `while` leads to a typically transient error. Under unlucky circumstances, it can remain undetected for some time, perhaps until an additional thread is introduced.

Ada. Condition synchronization is achieved by means of entry barriers, which are built into the syntax. Their format is not susceptible to easy programming mistakes. One possible mistake is to include in a barrier condition a variable that is defined outside the protected object. In that situation, it is possible to change the value of the condition without notifying waiting threads.

4.3.1 Placement of the wait loop

The wait loop in Java most often appears at the very beginning of a critical section and is reached immediately after a thread locks the object. But it can be placed anywhere within a synchronized method or block. As a simple example of one or more statements separating the wait loop from the beginning of a method, you could count the number of calls to a method, `m`, in an instance variable `callCounter` in the following way:

```
synchronized void m()
{
    callCounter ++;
    while (cond) {wait();}
    ....
}
```

Here, `callCounter` is incremented exactly once for each call, no matter if the calling thread enters the wait set. Its value equals the number or calls to `m` including those where the thread is still in the wait set. In a slightly more sophisticated example, the statements before the wait loop could maintain a list of the thread identities of the latest n callers. One can also instrument the wait loop itself similarly by including statements before and/or after the `wait` call.

The textbook case for placing the wait loop deeper inside a critical section is when a method allocates resources to calling threads. It may turn out that the request of a calling thread, `t`, cannot be satisfied until additional resources become available. In that situation, `t` can place itself in the wait set, release

the object and wait to be notified by a thread that has released resources. Once notified, `t` continues immediately after the `wait` call with exclusion synchronization in force. If a synchronized method has one or more such `wait` calls, a thread can effectively execute it in segments separated by those calls, entering a new segment each time it is successfully reactivated from the wait set.

Java pitfalls. The syntactical freedom to place the wait loop anywhere in a critical section allows certain errors. Even if the wait loop is initially placed at the very beginning of the critical section, a maintainer can unintentionally insert statements between the beginning and the wait loop. These statements are executed exactly once by every thread that attempts access to the critical section regardless of the condition. This may be even more treacherous if there are already statements between the beginning of the critical section and the wait loop, as in the `callCounter` example. The maintainer may not realize the difference in status between statements placed before and after the wait loop.

Ada. Because protected objects in Ada are syntactically distinct, there is no easy way to include a statement such as `CallCounter := CallCounter + 1;` in an entry in such a way that it would be executed exactly once under exclusion synchronization before the barrier has been passed. (Certain elaborate maneuvers are possible if you include in the barrier condition a function call with side effects.)

An Ada entry body cannot be broken into segments where a task would execute a segment, then release the object, put itself on queue and continue with the next segment upon reactivation. In Ada, each such segment must be an entry. A task that is executing an entry can call `requeue` and place itself on the queue of the same or another entry [2]. Requeuing is sometimes considered an advanced Ada topic. An intuitive example of requeuing when a resource turns out to be unavailable is given in [17].

4.3.2 Notification of waiting threads

A Java thread that executes a synchronized method on `o` and changes a condition that may affect one or more threads in o's wait set must `notify` those threads. In standard Java, `o.notify` reactivates one arbitrarily chosen thread, `t`, in `o`'s wait set. If the call is correctly placed within a wait loop, this means that `t` reevaluates the condition and either proceeds in the synchronized method or reenters the wait set. In RTSJ, the most eligible thread is reactivated.

The call `o.notifyAll` releases all threads waiting for conditional access to `o`. This is useful when a condition has changed so that multiple threads can proceed. But calling `notifyAll` is sometimes necessary even though you want only a single thread to proceed. In standard Java, this is the only way to give preference to the highest priority thread. It is inefficient if there are many threads in the wait set, since they must all attempt access, and only one will succeed [18].

Because there is only one wait set per object, you must also call `notifyAll` instead of `notify` if an object's wait set may include threads pending on different conditions. If you change one of the conditions, you must activate all the threads to make sure that a thread pending on that condition is notified, if it is in the set. This is true in RTSJ as well as in standard Java.

When a thread calls `wait`, `notify` or `notifyAll` on an object, it must have the object locked. The wait set is a shared data structure that must be protected from conflicting access, but has no lock of its own.

Java pitfalls. Unlike exclusion synchronization, condition synchronization is not automatic; you have to explicitly notify waiting threads. An obvious pitfall is to forget to insert `notify` calls at all

the necessary places. This is particularly treacherous if a method has unusual exits, as via exception handlers. A related mistake is to call `notify` instead of `notifyAll` when threads in the wait set may be pending on different conditions.

A way to reduce the risk of forgotten notifications in standard Java is to include a timeout parameter in every `wait` call. After the given time, the thread is activated, and if the `wait` call is placed inside a correct wait loop, the thread reevaluates the condition and either proceeds or reenters the wait set. In RTSJ this can only be used if you don't care about maintaining the FIFO per priority queuing discipline.

Ada. As long as the entry barrier depends only on variables local to the protected object, the most eligible, waiting thread is automatically activated after a protected procedure or entry call on the object has changed the truth value of the condition. Waiting tasks are queued per entry, so precise notification can be achieved: If a single condition is changed, only a task waiting on that condition is activated.

4.3.3 Controlling access to domain resources

Condition synchronization is used to give one thread at a time exclusive access to a shared resource in the problem domain, such as a forklift truck in an automated factory application [14, 19, 20]. In this example, jobs on the factory floor that need the forklift are represented by `Job` threads in the software. A forklift operation may continue for several minutes and must be performed under condition synchronization because we want waiting jobs to form a FIFO queue per priority. The object controlling the forklift – instance `f` of class `Forklift`, say – typically has an attribute, `busy`, that reflects the availability of the forklift, and the synchronized operations `acquire` and `release`, where `acquire` contains a wait loop such as the following:

```
while (busy) {wait();}
```

The corresponding notification call is in `release`. Statement sequences where the forklift is operated are bracketed by calls to `acquire` and `release` whether they appear in `Job`'s `run` method or in other unsynchronized methods.

While one job is using the forklift, other `Job` threads can call `f.acquire` and place themselves in `f`'s wait set. The variable `busy` serves as the lock on the physical forklift while `f`'s hidden lock variable only serves to control the access to the variable `busy` itself.

Explicitly calling `acquire` and release in this fashion is similar to working with a semaphore, and may be counterintuitive if you have been taught that semaphores are a primitive way of controlling the access to a shared resource. A synchronized method or block is a more abstract representation that hides the semaphore operations. But when controlling access to shared resources in the problem domain, we must invert the abstraction by using a synchronized object to implement a semaphore [20].

In the example with the shared printer, we can choose whether to consider the wait to be of long or short extent. In the solution in section 4.2.1, each thread's operations on the printer are enclosed in a synchronized block as follows:

```
synchronized (myPrinter)
{
    // series of statements producing output
}
```

This exclusion synchronization assumes that the printer operations are quick. If other threads try to access the printer during the exclusive access, they spin, waiting for the lock. There is no direct way to ensure that they will ultimately access the printer in a first-in-first-out fashion.

In an alternative solution based on condition synchronization, you define `acquire` and `release` methods in the `Printer` class (or another class), introduce a variable such as `busy`, and bracket the series of statements with calls to those methods:

```
myPrinter.acquire();
// series of statements producing output
myPrinter.release();
```

Here, `acquire` contains a wait loop, and threads that must wait for the printer enter the wait set. A downside is that this solution makes the programmer responsible for inserting one or more `release` calls to ensure that the printer is released even if an exception is thrown while the output is being produced.

Java pitfalls. By convention, all critical sections should be programmed to minimize the time an object is held locked. A thread that is waiting on a condition should release its object locks and be placed in a wait set. But nothing stops a programmer from making a thread hold an object lock for an arbitrarily long time. A trivial way to do this is to call `sleep(...)` inside a synchronized method. Two other cases are described next.

Controlling domain resources. The confusion of long and short waits may typically occur in real-time applications that control resources in the problem domain. In the automated factory domain, the forklift operation may be implemented by means of the following synchronized block within the `run` method of the `Job` class:

```
synchronized(f)
{
    // Operate the forklift
}
```

This ensures mutual exclusion of jobs using the forklift and may at first seem more elegant than the solution with semaphores. But if the forklift operation continues for minutes, `Job` threads that need the forklift are not put in a wait set (and FIFO queued per priority in RTSJ) but spin until they find `f` unlocked. Which `Job` thread gets to the forklift next is then quite arbitrary. To avoid this, condition synchronization must be used.

In RTSJ, exclusion synchronization invokes the control policy to minimize the effect of priority inversion. Assume first that the default policy, priority inheritance, is in effect. If a job at priority `l` is currently operating the forklift and a higher priority job, `h`, attempts to get the lock, `l`'s remaining forklift operations will be executed at priority `h`. This skews `l`'s priority relative to any jobs with priorities between that of `l` and that of `h`. The ceiling priority protocol has an even more fundamental effect in that all forklift operations will always be carried out at the highest priority of any job.

Nested synchronized blocks. Another way of inadvertently mixing long and short waits is with nested critical sections. We can insert a wait loop in a nested synchronized block as follows:

```
synchronized(r1)
{
    ....
    synchronized(r2)
    {
        while (cond) {r2.wait();}
        ....
    }
}
```

If `cond` is `true`, the calling thread enters `r2`'s wait set and releases `r2`. But it keeps `r1` locked, and lets other threads that need access to `r1` spin rather than wait in a wait set. Incidentally, the following is also legal:

```
synchronized(r1)
    {
    ......
    synchronized(r2)
    {
        while (cond) {r1.wait();}
        ....
    }
}
```

In this case, the calling thread enters `r1`'s wait set and releases `r1` while keeping `r2` locked.

Ada. The Ada syntax is certainly clearer about the distinction between exclusion and condition synchronization. Any protected operation provides exclusion synchronization automatically. Any "potentially blocking operation", that is, essentially anything that can take time, is forbidden in a protected operation, ensuring that the extent in time of mutual exclusion is kept short. For example, you cannot call an entry of some protected object `r2` while you are executing a protected operation on the object `r1`, which would be the Ada equivalent of nested synchronized blocks.

Condition synchronization requires an entry with a barrier condition. The only way to control access to a shared domain object is by means of a semaphore object similar to the `Forklift` class in Java. A `Forklift` protected object would have an `entry Acquire` with a barrier such as `not busy` and a procedure `Release`.

In the case of the printer, if it is undesirable to define protected procedures for each different combination of printer operations, a semaphore object is the only solution permitted in Ada. It would be a protected object `My_Printer` with the `entry Acquire` and the `procedure Release`.

5 Conclusions

Java was not originally intended as a language for systems with high reliability requirements, but its popularity has prompted its use for ever wider sets of applications. The Real-Time Specification for Java removes some of the obstacles associated with garbage collection but retains many pitfalls.

Java is adequate for many kinds of concurrent software, but for critical real-time applications it remains a considerably riskier choice than Ada, which was intended for such applications. This is so because Java lacks safeguards against programming errors that are easily committed by a programmer

without a sufficiently deep understanding of concurrency issues. There is a trade off here where Java's popularity and the availability of Java programmers must be weighed against the risk exposure caused by those programmer mistakes the language readily allows.

References

[1] Barnes, J. G. P. (1998) *Programming in Ada95*, 2nd Ed., Addison-Wesley.

[2] Burns, A. and Wellings, A. J. (1998) *Concurrency in Ada*, 2nd Ed., Cambridge University Press.

[3] Lea, D. (2000) *Concurrent Programming in Java*, 2nd Ed., Addison-Wesley.

[4] van der Linden, P. (2001) *Just Java 2*, 5th Ed., Prentice Hall.

[5] Holub, A. I. (2000) *Taming Java Threads*, Apress.

[6] Hyde, P. (1999) *Java Thread Programming*, Sams Publishing.

[7] Oaks, S. and Wong, H. (1997) *Java Threads*, O'Reilly.

[8] Brinch Hansen, P. (1999). Java's insecure parallelism, *ACM SIGPLAN Notices* **34/4**, 38-45.

[9] Alexander, R. T. and Bieman, J. M. and Viega, J. (2000) Coping with Java programming stress, *IEEE Computer* **33/4**, 30-38

[10] Bollella, G. and Gosling, J. (2000) The real-time specification for Java. *IEEE Computer* **33/6**, 47-54

[11] Bollella, G. and Gosling, J. and Dibble, B. P. and Furr, S. and Turnbull, M. (2000) *The Real-time Specification for Java*, Addison-Wesley.

[12] Brosgol, B. M. (1998) A comparison of the concurrency features of Ada95 and Java, *Proc. SIGAda '98*, (*Ada Letters* **XVIII/6**,175-192).

[13] Sebesta, R. W. (2002) *Concepts of Programming Languages*, 5th Ed., Addison-Wesley.

[14] Sandén, B. I. (1994). *Software Systems Construction with Examples in Ada*. Prentice-Hall.

[15] Wellings, A. J. and Johnson, R. W. and Sandén, B. I. and Kienzle, J. and Wolf, T. and Michell, S. (2000) Integrating object-oriented programming and protected objects in Ada95. A*CM TOPLAS* **22/3**, 506-539. (Reprinted in *Ada Letters* **XXII/2** (June 2002), 11-44.

[16] International J Consortium (2000), *Specification, Real-Time Core Extensions*, Draft 1.0.14, 2 September 2002. http://www.j-consortium.org

[17] Sandén, B. I. (1996) Using tasks to capture problem concurrency. *Ada User Journal* **17/1**, 25-36.

[18] Vermeulen, A. and Ambler, S. W. and Bumgardner, G. and Metz, E. and Misfeldt, T. and Shur, J. and Thompson, P. (2000) *The Elements of Java Style*, Cambridge University Press.

[19] Sandén, B. I. (1997) Modeling concurrent software. *IEEE Software* **14/5**, 93-100.

[20] Carter, J. R. and Sandén, B. I. (1998) Practical uses of Ada95 concurrency features. IEEE *Concurrency* **6/4**, 47-56.

(Footnotes)

[1] "Ada" refers to the revised standard "Ada 95".

[2] Ada also provides for singleton protected units.

Att utveckla Blueberry har inte varit något för blåbär

Ett av de intressantaste och kanske svåraste svenska programvaruprojekten på senare år har varit utvecklingen hos Sjöland & Thyselius av "Blåbär". Det har nu börjat slå igenom på den internationella marknaden, än så länge främst för Modellering och Simulering i militära sammanhang. Vi återger här en kort beskrivning av "Blueberry", vilken också ger några intressanta användningsexempel.

Blueberry3D – Procedural Geometry for Real-time Graphics

Contact: Rune Thyselius, Sjöland&Thyselius Virtual Reality Systems AB, Stockholm

info@blueberry3d.com www.blueberry3d.com

A common concern of the real-time terrain visualisation of today is the lack of geometric detail, when the terrain model is studied close enough. This is primary not a problem of the digital terrain data used being of too low resolution. The real reason is that the terrain model is pre-generated and stored, thus imposing database size and retrieval rate limitations to the model.

We now want to introduces an alternative technique, with which most geometric detail is created only when needed (*procedural geometry*). This renders many important benefits, including unlimited detail, no repetition of geometry, and optimal system resources usage. An implementation of the procedural geometry technique, Blueberry3D, is presented, illustrating examples of the kinds of geometric detail that allows itself to be procedurally generated.

Traditional approaches

Most real-time terrain visualization packages of today rely on a simple concept. An orthographic aerial photo or satellite image is draped upon a height model. The image can vary in spatial and color resolution, and there are countless variations of the height model, but some important characteristics are common amongst all solutions.

The most serious concern is the limitations of the resolution of the raw map data. For elevation data, a resolution of tens of meters is common, and for image data no better than meter resolution is usually available for larger areas. Resolution increases, and new schemes for handling very detailed elevation data are being developed but the resolution will still be limited. At a certain distance from the virtual terrain, it will always turn flat looking and uninteresting.

One way to remedy this problem is to generate content, like foliage, for the terrain. This approach will decrease the minimum distance where the terrain looks rich and alive, but the problem remains. Pre-generated content needs to be bound in resolution due to storage space and retrieval speed limitations.

Procedural Geometry

Definition

Procedural Geometry refers to geometrical shapes created programmatically. For example, take a sphere-drawing routine. You supply it with a centre and a radius and the routine finds an appropriate set of triangles that together will, at least approximately, describe the surface of the sphere. These triangles are an instance of procedural geometry, and may later be used for the actual drawing of the sphere.

In this paper we adopt a more narrow definition. With procedural geometry, we only refer to shapes created *in real-time*, that is, shapes are created at the time they are needed, as opposed to shapes created by a pre-processing routine and stored for later. This rules out most existing

terrain visualization and related systems that use procedural geometry from our discussion, since these systems only use the wider definition.

It is important to note that procedural geometry, mainly is focused on creating content on resolutions below those of the raw terrain data. At higher resolutions, pre-processed terrain data is used for the visualization.

Benefits

Six main potential benefits from using procedural geometry for real-time terrain visualization have been identified.

1. *Unlimited resolution*. Detail down to centimeter scale and below can be created by applying fractals for computation of procedural geometry.
2. *Richness*. In practice, no repetition at all will occur, even for vast areas, by applying controlled randomness to the procedure.
3. *Natural terrain*. The unlimited resolution, fractals and random richness all serve to create natural areas that really come alive, much more than what can be achieved by traditional techniques.
4. *Decreased design time*. If made by hand, the small details in a terrain model are the most expensive to design. When procedural geometry is used, huge amounts of detail can be abstracted by small procedures using controlled random numbers.
5. *Compact databases*. The database size can be kept to a minimum by keeping most of the content stored implicitly in the program itself.
6. *Optimal system usage*. A terrain engine using procedural geometry can easily adopt the detail to comply with the hardware at hand, as there is no limit to the amount of detail available.

Problems

Procedural geometry also comes with its share of downsides. Below, three issues are listed and addressed.

1. Procedural geometry is CPU intensive. However, even the standard pc:s of today are powerful enough to create rich terrains with procedural geometry.
2. Procedural geometry demands a lot of system memory. This is mainly due to the fact that more detail than before can be created, but also that the generated shapes need to maintain extra state variables used by the generator. These variables would not be needed had the shapes been pre-created. In practice, 256 MB of memory is enough.
3. Procedural geometry needs advanced level-of-detail handling. Not all the geometry of a terrain model can be generated and used at once, that would severely overload any hardware system. Rather, a scheme needs to be devised to decide where in the terrain model a high LOD is needed and where it is not. A related problem is called “plopping”; how do we change from one level of detail to another with as little visual interference as possible? The level-of-detail handling problem can be addressed in several ways, but there is not yet a perfect solution.

Procedural geometry in the Blueberry3D system

Blueberry3D is a new terrain visualization toolkit that uses standard formats for map data in combination with several different geometry generators. The result is an easy to use tool that, with little user effort, creates rich terrain models. In the following paragraphs, some of the geometry generators in Blueberry3D are presented as examples of the effects achievable with procedural geometry in the field of terrain visualization.

Elevation

Ground elevation geometry is generated as a variation of two-dimensional *Fractal Brownian Motion (FBM)*. The “coarseness” of the surface is specified per *terrain class*.

Ground Structure

To achieve a realistic looking ground structure, several *ground layers* are used. A Blueberry3D ground layer usually maps to a real ground layer like soil, rock or sand. The layer is fitted with a set of parameters describing properties like erosion, coarseness and fertility. Each terrain class

can have its own set of ground layers.

The ground structure geometry generator takes all the local ground layers and their parameters into account when the geometry is generated. Many natural effects like rocks poking out of the surrounding soil and sea waves flushing away grass and dirt, leaving only sand, are obtainable by properly configured ground layers.

Vegetation Distribution

The distribution of vegetation items is performed by another variation of the FBM. The effect is that natural-looking glades and clusters of trees and bushes are created automatically. The fertility of the local ground is also taken into account.

Vegetation Items

Vegetation items, like individual trees and bushes, are modeled using Iterated Function Systems (IFS). A unique instance is generated by the corresponding geometry generator and controlled random numbers.

Curved Surfaces

Blueberry3D uses polynomial surfaces to model roads, rivers, walls, paths, shafts, house foundations, etc. The main idea in all cases is that the user provides a minimum of information and the system takes care of the details of the modeling. For instance, a road is modeled as a set of points in the terrain and a profile. The curved surfaces generator interpolates the points, adopts the terrain around the road to make it blend nicely and finally places the road.

Curved surfaces, like all other procedural geometry, are computed in real-time. The result of this is that, no matter at how close range the user observes the surface, it will still look smooth. Only a few real-time systems have this functionality, one example (not terrain-visualization related) is the OpenGL Optimizer from Silicon Graphics.

Conclusion

The Blueberry3D software shows that real-time terrain visualization using procedural geometry is indeed achievable on a standard home computer, with all the benefits including increased detail and variation, huge terrain models and short design time. The technology has a great potential and it is our opinion that it will play an increasingly important role in the field of real-time terrain visualization.

Customer success stories

EADS

Building large terrains to cover huge expanses, and at the same time ensure the highest possible ground fidelity at any given location is a daunting task to say the least. EADS, a Blueberry3D user, contacted the services division of the Sjöland & Thyselius Group in order to build a very large piece of terrain that should be used as part of a complex application involving numerous tools and interfaces. The requirement was to automatically generate as much of the detailed terrain features, as cost and time constraints made it impossible to manually edit the terrain.

The database covers a large part of France and includes detailed ground and road features. The work was completed in record time and the final application used Vega Prime from Multigen-Paradigm, Inc. as the scene graph and VR-Forces from Mäk Technologies, Inc. to handle the computer generated forces. Various terrain is depicted where an Unmanned Aerial Vehicle (UAV) is tasked to identify a launch area for Scud missiles.

The application was showcased in the EADS booth during the 2003 Paris Airshow in Le Bourget, France with great success.

EADS is the largest aerospace company in Europe and the second largest worldwide.

Boston Dynamics

One of the most difficult things to simulate is human interaction, Boston Dynamics has spent well over a decade perfecting their technologies within this field. Recognized as the leader in human simulation with customers all across the globe that use their software products, engineering services and knowledge for mission planning, training and virtual prototyping.

DI-Guy is one of the flagship products that Boston Dynamics offers as a third party plug in to Vega Prime. Very quickly it became apparent

that combining the procedurally generated terrain from Blueberry3D with interactive soldiers would add a new level of fidelity to ground warfare applications. An engineering effort was undertaken to build a proof of concept demo using Vega Prime, DI-Guy and the Blueberry3D Development Environment to see what the result would be like.

Boston Dynamics is now using the Blueberry3D products as part of their services offering, delivering unprecedented levels of detail with grass literally growing around the soldiers feet!

BAE Systems

BAE Systems, U.K. went through the process of classifying their Ambleside database with material information and process it through the Blueberry3D Terrain Editor in order to add more complexity. At the highest altitude there is very little difference as the two databases essentially are using the same geo-specific aerial photograph. The second altitude shows a dramatic increase in detail as large forest areas clearly can be seen stretching all the way to the horizon. At ground level, the full impact of fractal based, procedural geometry can be seen. Please note that the same source data was used, and material classification was based on the original aerial photo.

Several features such as tanks, SUV's and helicopters were added in order to build different dynamic scenarios. A large factory with several buildings and connecting roadways were included as well as a cave model for simulating underground facilities.

Programvaruomfattningen i amerikanska och svenska stridsflygplan

Att vi i Sverige var tidigt ute med datorer i flygplan känner de flesta SESAM-iter till. Men att vi faktiskt under åren hållit rätt så jämna steg med stormakterna har kanske inte alla klart för sig.

Den 17 december 2003 var det 100 år sedan Bröderna Wright flög första gången. Detta har Sveriges Mekanisters Riksförening och Flygtekniska föreningen tagit till intäkt för att ställa sig bakom utgivningen av en bok om "Flygtekniken under 100 år". Där finns vid sidan av alla andra teknologier som behövs för flygandet, en beskrivning av avionikens, eller flygelektronikens, utveckling, både allmänt och för svenska flygplan, liksom för svenska flygelektroniks historia (allt kortfattat). Bl a tog Dag Folkesson med assistans av Bengt Sjöberg och Kim Bengtsson samt med datakällor anvisade av Currie Colket i USA (f n bl a ordförande i SIGAda) för boken fram nedanstående diagram som (approximativt förstås) visar programvolymens utveckling i svenska och amerikanska stridsflygplan. Diagrammet sträcker sig från det första operativa flygplanet med en dator, Hughes rörbestyckade och trumminnesförsedda dator i F106 navigerigs/siktessystem MA-1, via AJ37 med Saabs CK37 till våra dagars JAS39C och den framtida Joint Strike Fighter med omfattande uppsättningar datorer. Som framgår av diagrammet är de svenska flygplanen väl jämförbara med de amerikanska om man mäter i

programvolym (mer program är ju inte alltid bättre, men här ger det nog en indikation om funktionsomfattningen och vad som gjorts för att göra ett komplext system hanterligt för föraren). Från Dag Folkessons föredrag vid SESAMs höstseminarium 2001 vet vi ju att svensk flygindustri varit först ute med många framstående tekniska lösningar bl a inom realtidshanteringen. I boken ges ytterligare exempel på avancerad funktionalitet och integration i främst JAS39.

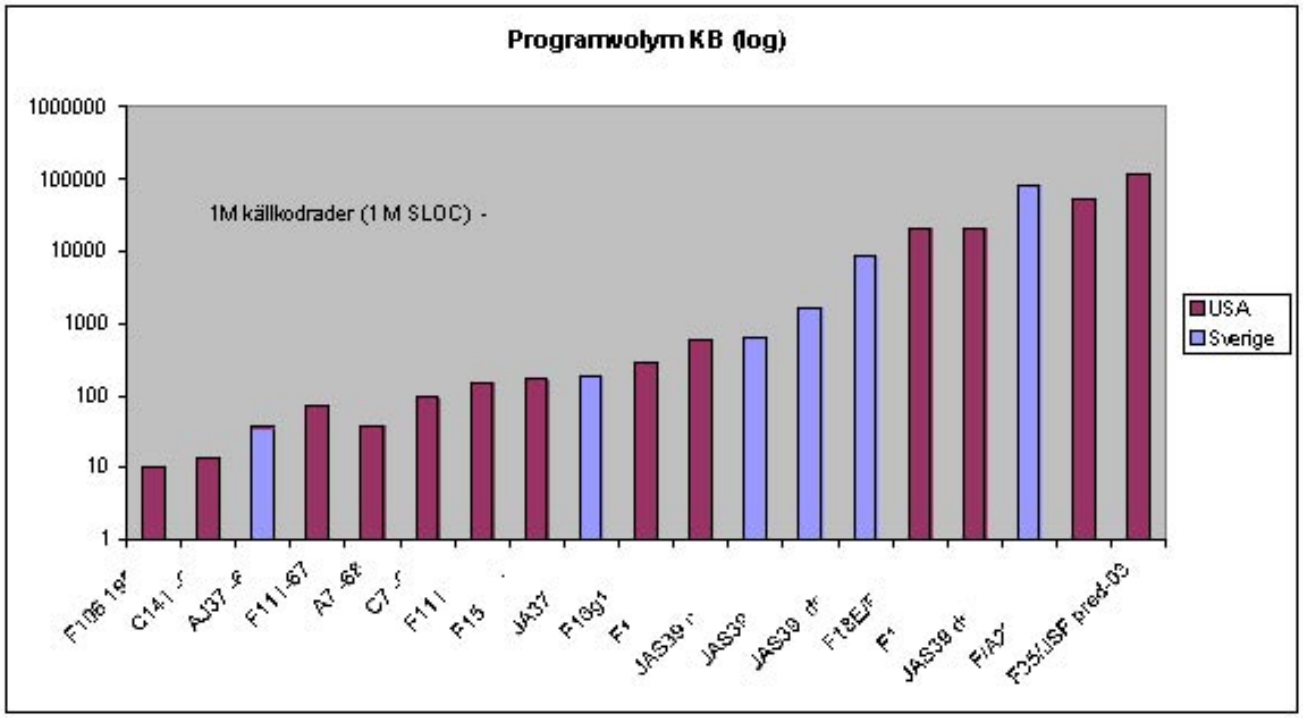

Diagram illustrating volume growth for programs in executable direct access memories of on-board computers (excluding e g data bases in bulk memories) in US and Swedish combat aircraft systems. Data for recent US a/c in SLOC taken from presentations at DoD 2002 Cost Analysis Symposium and 2003 Software Technology Conference and converted approximately to Kilobytes (KB), for Swedish a/c in KB from Saab.

(From Flygtekniken under 100 år, SMR 2003) Diagrammet återges med tillstånd av SMR.

"Flygtekniken under 100 år" som är på ca 350 sidor (varav kanske 50 totalt om avionik), utges av Sveriges Mekanisters Riksförening, www.smr.nu, e-post: smr@swipnet.se, SMR kansli 08-6679320.

I Carlsson

Årets seminarier framgångsrika

Även årets två seminarier var mycket välbesökta och fortsatte trenden från förra året. Sommarseminariet med Nancy Leveson om systemsäkerhet hade ca **126** deltagare och höstseminariet om Informationssäkerhet **147** deltagare, förhoppningsvis uppskattades även budskapen.

Vi återger nedan ett initierat och mycket läsvärt referat av Levsons föredrag som Inga-Lill på begäran har skrivit. Tyvärr har vi inte något motsvarande att bjuda på betr "Informationssäkerhet i en NBF-miljö". Där fanns det ett genomgående tema som belystes från olika aspekter, som kanske inte är så lätta att sammanfatta. Båda seminariernas presentationsmaterial, så långt vi har fått tillgång till det, finns på SESAM CD2003.

System Safety for Software-Intensive Systems

SESAM:s seminarium i systemsäkerhet för programvaruintensiva system den 20-21 augusti blev en synnerligen välbesökt tillställning och ett unikt tillfälle att lyssna till –och diskutera med– den person, som byggt upp ämnesområdet Programvarusäkerhet (Software Safety). För denna pionjärinsats har Prof. Nancy G. Leveson från MIT erhållit åtskilliga utmärkelser. Med sin 40 åriga erfarenhet av säkerhetskritiska system för industri och myndighet, samt som författare till ett flertal artiklar och böcker inom området, har Leveson en rik källa av illustrativa exempel att ösa ur.

Efter en historisk tillbakablick på datorns revolutionerande betydelse för systemutvecklingen, samt för- och nackdelar med programvarans flexibilitet konstaterade prof. Leveson att merparten av observerade felyttringar i den operativa programvaran kan hänföras till ofullständiga krav. En av bristerna med den Polar-landare som kraschade på Mars var bl a att det saknades krav på i vilket läge systemet senast behövde landningssensorernas data för säker landning. Bakom detta låg kommunikationsproblem mellan de olika experterna för autopilotsystem resp programvara, vilket ledde till att den programvarumodell, som upprättades över systemet, inte stämde med den fysiska designen. Sensordata inför landning kom därför att sändas då landaren befanns sig 40 cm ovanför Mars yta i stället för de nödvändiga 12 metrarna.

Trots olika systemsäkerhetsanalyser, testtekniker etc uppstår systemsäkerhetsproblem.

För programvara går det inte (som för hårdvara) att kvantifiera olyckssannolikheten (vilket många försöker göra i sina felträdsanalyser). Det är inte heller praktiskt genomförbart att testa alla exekveringsvägar i ett programvarusystem. Hög tillförlitlighet (***Reliability***) är inte en garanti för systemsäker-het (***Safety***). Tillförlitlighet är en komponent-egenskap, medan systemsäkerhet är just en systemegenskap, där systemets omgivning och användning har betydelse. Ofta visar sig säkerhetsproblemen ligga i interaktionen mellan systemkomponenter, som var för sig kan uppvisa ett tillförlitligt beteende. Ett exempel är det prov som utfördes för att se om attackjaktplanet F/A-18 kunde nå viss höjd, vilket det också gjorde. Dock levererade dess mekaniska system attityddata med högre frekvens än programvaran kunde hantera. Vid en annan F/A-18-flygning provsköts en vingmonterad missil, vars mekaniska spärr inte frigjorts. Incidenten inträffade lyckligtvis på hög höjd, där piloten kunde bemästra de 300 pund i extra dragkraft under ena vingen, som avfyrningen plötsligt medförde.

Att ensidigt koncentrera sig på största möjliga tillförlitlighet är m a o inte effektivt. Tyvärr är alltför många av de systemsäkerhetsstandarder som finns för programvara inriktade mot tillförlitlighet (bl a IEC 61508, DO 178B), vilket också avspeglar sig i SIL-begreppet (***Safety Integrity Level***) –ofta kopplat till kvantifierade sannolikheter (t ex 10^{-9}), för att en olycka kan inträffa.

En vanlig orsak till mjukvarurelaterade olyckor är inkludering av återanvänd programvara. Klassiska exempel är Ariane 5-raketen, den datorstyrda acceleratorn Therac-25 för strålbehandling samt den senaste ATC-programvaran för södra England (baserad på ett amerikanskt system, som visade sig negligera lägen öster om Greenwich-meridianen, dvs 0^{o}). Olyckor uppstår, när de förutsättningar på vilka programvaran bygger, inte längre är uppfyllda.

Ett primärt säkerhetsproblem för datorbaserade system är dock en alltför hög grad av flexibilitet. En av systemsäkerhetsingenjörens uppgifter är därför att identifiera de konstruktionsbegränsningar, som är nödvändiga för att upprätthålla systemsäkerheten, samt att försäkra sig om att systemets och programvarans konstruktion inte bryter mot dessa.

En genomgång av olika systemsäkerhetsbegrepp följde med exempel på en enkel dörrstängningsautomatik, där identifierade riskkällor (***hazards***) omformulerades till konstruktionsbegränsningar.

Ett metod- och verktygsstöd (***SpecTRM***) avsett för såväl kund/beställare som systemleverantör vid utveckling och analys av säkerhetskritiska

system presenterades. Detta underlättar kundens dokumentation av systemets ursprungliga syfte/ målsätt-ningar (***intent specifications***) –vilka annars kan vara svåra att återfinna för den enskilde utvecklaren. Avsiktsspecifikationerna samt den därur härledda systemarkitekturen visar sig också vara väl lämpade för återanvändning (något som däremot inte gäller kod, se ovan). Systemet beskrivs i 7 olika vyer, vilka tas fram av olika roller inom kund- resp leverantörsorganisation (t ex projektledare, systemingenjör, arkitektur- resp komponentutvecklare). Ett formellt baserat och exekverbart kravspråk (***SpecTRM-RL***) i form av logiska "and-or"-tabeller, ger möjlighet för den applikationskunnige (t ex en pilot) att kunna förstå och kontrollera detaljerade krav utan att behöva behärska ett formellt språk. Det ger också möjlighet till automatisk analys av kraven m a p fullständighet (t ex robusthet, determinism), motsägelsefrihet, efterlevnad av konstruktionsrestriktioner etc. Spårbarhet mellan olika typer av säkerhetsrelaterad information (från högnivåkrav till detaljkonstruktion och kod) understöds. SpecTRM illustrerades m h a antikollisionssystemet TCAS. Detta underhålls m a p den modellering som gjorts i SpecTRM –ändringar införs på kundnivå i kravspecifikationerna varifrån leverantören har att härleda dessa vidare ned i systemet för realisering av en ny version.

De traditionella tekniker som står till buds för att genomföra olika typer av systemsäkerhetsanalyser (HAZOP, FMECA, FTA osv) har ursprungligen tagits fram för mekaniska system. De flesta är inriktade mot tillförlitlighet, dvs att undvika felyttringar (***failures***) snarare än mot systemsäkerhet och att undvika olyckor. Direkta samband mellan händelser uttrycks i felträd eller händelsekedjor mellan felkällor och felyttringar. Vad de därmed missar är att analysera icke-linjära samt indirekta samband (t ex återmatningar), säkerhetsbrister i interaktion mellan systemkomponenter resp operatörer, mänskliga misstag eller fjärmanden från fastlagda procedurer, ursprungsorsaker (***root causes***) etc. Nancy Leveson har sedan ett par år arbetat på att ta fram en mer systemorienterad riskkälleanalys, ***STAMP***, som modellerar och analyserar ett system m a p dess olika styrmekanismer (***control struc-tures***), för att finna eventuella bidragande olycksfaktorer (snarare än att analysera risk-källor, olycksutfall).

Centralt för varje styr- och reglersystem är någon typ av styrslinga (***control loop***) samt en återmatningsmekanism (***feed-back loop***) som returnerar aktuellt status: information nödvändig för fortsatt styrning (väsentlig för dynamiska system som har att reagera och anpassa sig till förändringar i system och omgivning). Ofta är det just i feed-back-mekanismerna som brister kunnat påvisas i många av de olycksscenarier som analyserats. För att i ett säkerhetskritiskt system förhindra aktivering av kvarstående (men ofta oundvikliga) systemsäkerhetshot (t ex bekämpning av eget / allierat flyg i ett stridsläge p g a misslyckad identifiering) krävs styrning från olika nivåer i systemhierarkien i form av säkerhetsrestriktioner, som läggs på underliggande system och handhavare samt olika typer av uppföljningar av att dessa följs.

STAMP:s modellering av övervakat system, dess olika styr- och återmatningsmekanismer (sensorer, operatörer) är applicerbar inte bara på ett system, dess omgivning samt aktörer, utan även på de utvecklingsprocesser som används vid systembygget och den organisation samt de beslutsfattare som är involverade i processerna. Faktorer, som ej beaktas i de traditionella analysteknikerna.

Modelleringen har applicerats bl a på antikollisionssystemet TCAS (fallet ***mid-air collision***), US Air Force oavsiktliga nedskjutning av egna arméhelikoptrar i Irak 1994 (***friendly fire*** i flygförbudszon) samt fallet med den år 2000 förorenade vattenreservoiren i Kanada (med hälsovårds- och miljömyndigheter, lagstiftning, etc som styrande instanser samt ett vattenprovlab för kontroll av reservoirerna och kommunens tillsyn av labbet som återmatande instanser). STAMP har även tillämpats på en modell över processinteraktioner vid systemutveckling samt systemdrift (med styrmekanismer vid utvecklingen i form av lagstiftning, industriella standarder, certifieringar, leverantörens regelverk och användarinstruktioner samt feed-back-mekanismer via leverantörens granskningar, analyser, prov, problemrapporter, beställarens leveranskontroller samt användarens olycks-/ incidentrapporteringar från driftmiljön).

Leveson illustrerade hur man modellerar de

restriktioner en överliggande nivå i en hierarkisk struktur ålägger en underliggande nivå för styrning och kontroll av denna samt hur denna modell kan utnyttjas för att finna brister i styrnings- och återmatningsmekanismerna på olika nivåer (från formella ledningskanaler mellan olika handhavare till kommunikationskanaler mellan olika delsystem). Modellen kan även användas för att uttrycka effekten av olika administrativa förändringar (t ex budget ökningar/nedskärningar) på olika faktorer i ett systems säkerhetsprogram. STAMP är fortfarande under utveckling och kommer att publiceras i bokform (denna gång fritt tillgänglig på nätet).

Två-dagars seminariet avslutades med en översikt över principerna för hur man bygger in systemsäkerhet i sina mjukvarusystem i allmänhet samt i människa-maskin-interaktionen i synnerhet (mycket av detta finns att läsa i Leveson's klassiker "Safeware: System Safety and Computers", ISBN 0-201-11972-2, 1995). Att bygga säkra programvarusystem kräver både specialkunskap och erfarenhet för att få fram enkla och hanterbara system, där säkerheten inte lagts in i form av olika skyddssystem i efterhand, utan byggs in redan under konstruktion.

Overheadmaterial från seminariet kommer att finnas tillgängligt på den CD, som SESAM producerar och skickar ut till alla som deltagit i något av SESAM's seminarier under året.

Verksamheten inom IG Programvarusäkerhet 2002-10-24 – 03-12-16

En intressegrupp i programvarusäkerhet har efter beslut av SESAM rådsmöte inrättats.

Gruppen bildades under SESAM:s höstseminarium i oktober 2002 och har sedan dess genomfört möten/seminarier med interna och inbjudna talare över olika teman inom Programvarusäkerhet. Ytterligare möten/seminarier fram till första kvartalet 2004 har förberetts.

Möten och seminarier:

Mötesform	*Datum*	*Tema*	*Kommentar*
Gruppmöte #1	02-10-24	• OS i säkerhetskritiska tillämpningar	3 externa talare.
Gruppmöte #2	03-02-04	• Teknikprov inom FoTA P12 • Systemsäkerhetsanalysmetoder inom olika företag	9 talare varav 2 externa.
Extra seminarium	03-03-26 --27	• Safeware:s SpecTRM-verktyg	2 Leveson-doktorander
Gruppmöte #3	03-05-07	• SpecTRM-prov, • Design by contract, • Återanvändning o interoperabilitet	Enbart interna talare
Leveson-seminarium	03-08-20 -- 21	• System Safety for Software-Intensive Systems	Seminarium öppet även för icke-SESAM medlemmar
Gruppmöte #4	03-09-18	• Tidsstyrd programmering/nätverk	5 externa talare.
SESAM:s höst-seminarium	03-10-21 -- 22	• Gruppmöte #5, Ag-Redovisningar • Informationssäkerhet i en NBF-miljö.	SESAM Gruppmöten + Öppet SESAM-seminarium.
Gruppmöte #6	04-02-11	• Programvara i UAV:er	5 externa talare.

SESAM:s hemsida:

Under rubriken ”Arbetsgrupper” på SESAM:s hemsida har en struktur för IG Programvarusäkerhet lagts in, där dokumentation i form av inledande förutsättningar (upprop, programförklaring) samt mötesagenda med presentationsmaterial och mötesnotiser finns tillgängliga. Öppet material har hämtats från det projekt som utgjort mönster vid bildande av Ig Programvarusäkerhet –det under 2002 avslutade FoTA P12: Överföring till industrin av programvaruteknik för säkerhetskritiska system– ett projekt, vars karaktär av specialinriktat nätverk kombinerat med kunskaps- och teknikutbyte samt teknikprov visat sig vara en framgångsrik modell för hur tekniköverföring kan befrämjas mellan industri, företag, organisationer och högskolor. Till den information som förts över och sedan uppdaterats hör förteckningar över ”Utbildning och kurser”, ”Intresseorganisationer”, ”Publikationer”, ”Verktyg” samt ”Projekt”.

Tekniköverföring/-utbyte:

Detta har realiserats främst genom diskussioner och info-utbyte under möten och seminarier. Framöver förväntas även teknikprov utförda inom medlemsorganisationerna kunna leda till erfarenhetsutbyte både bland dem som utför provningarna och de som enbart tar del av utfallen, men i gengäld agerar som granskare/bollplank. Gemensamma frågeställningar kan formuleras och fördelas mellan olika provare. Angreppssätt och realisering av tidiga prov kan komma senare prov tillgodo varigenom dubbelarbete undvikes och ett snabbare genomförande möjliggörs. Såsom medlemmar i en icke-vinstdrivande och utbildande organisation (SESAM) har IG-deltagare getts möjlighet att utan kostnad prova en produktionsversion av verktyget SpecTRM. Två företag har valt att prova verktyget genom att formulera examensjobbsuppgifter, något som stimulerat utbytet mellan deltagande industrier och högskolor.

Medlemmar:

Antalet medlemmar är f.n. 21 personer –inberäknat ordföranden för Ag Metodik resp. Teknik– varav flera från företag, som inte tidigare ingått i SESAM (se Medlems-/Intressentlista).

Programvara i obemannade farkoster 11/2 -04

SESAM:s intressegrupp i Programvarusäkerhet håller sitt nästa möte onsdag den 11 februari 2004, på FMV, Banerg. 62, Filmsal C, Stockholm.

Förmiddagen är i vanlig ordning vikt för gruppmöte. Eftermiddagen, som är öppen även för icke gruppmedlemmar, ägnas åt att belysa system- och programvaruaspekter inom ett visst område. Temat denna gång är ”Programvara i obemannade farkoster”.

Preliminärt program:

12.30 Inledning, *Inga-Lill Bratteby-Ribbing, FMV*

12.35 Övergripande krav för flygning med UAV:er, *Lars Sundlin, LFV*

13.10 UAV-verksamhet i Sverige, *Stefan Tenor, FMV*

13.45 Programvarupartitionering efter kritikalitet, *Rikard Johansson, SAAB*

14.20 Programvaruutveckling för Sharc Technology Demonstrator, *Carl-Olof Carlsson, SAAB*

14.55 Kaffe

15.20 Spaningsfordon för bebyggda områden, *John Folkesson, KTH*

15.55 Autonoma undervattensfarkoster, *Maria Pettersson, Saab Underwater Systems*

16.30 Diskussion

17.00 Avslutning

Anmälan till eftermiddagsseminariet lämnas senast mån den 9/2 genom att uppge namn, företag samt personnr till Christina Gunnarsson, FMV (e-post: christina.gunnarsson@fmv.se, fax: 08-782 5788). Gruppmedlemmar kan anmäla sig på vanligt sätt till undertecknad.

Inga-Lill Bratteby-Ribbing
Ordf IG Programvarusäkerhet

SESAM Kalender

2004-02-06	VU-möten, kl 1000
2004-03-25	
2004-05-27	
2004-08-26	
2004-09-30	
2004-11-25	
2004-04-22	SESAM Rådsmöten, kl 1000
2004-10-19	
2004-10-20	SESAM höstseminarie