Safety Critical and High Reliability Software: A Case Study in Real-Time Design

Dr Tony Hedge, Benthic Sciences LLP (tony@benthicsciences.co.uk)

Abstract

This paper examines some typical design problems from the perspective of maritime systems. Maritime systems are commonly required to integrate data from a dozen or more "sensors", each quite complex in their own right. Serial data links (RS-422 or RS-485) are still by far the most common interconnection techniques. Careful consideration must be given to synchronisation issues, and we look at some of the appropriate real-time design techniques for ensuring integrity is maintained, and at the role that appropriate tools can fulfil.

Foreword

The introductory notes for this event describe its aims as "exploring the approaches used to develop embedded software in some of these different safety-critical applications, with a view to establishing any common approaches and identifying opportunities for sharing best practice and development tools and techniques." This paper does not set out to break new ground, but rather strives to put into a real-life context some very old and well-established problems, with old and well-established solutions, and to ask some questions: As an industry, do we handle such problems very well? What are the best current techniques for addressing these problems? Am I really the only one who cares?

Background

Benthic Sciences LLP provides support to a range of clients to support the design, development and testing of software in real-time embedded systems. Our precise role will vary from project to project; in one, we may be delivering a design against which other individuals or teams will go on to code, to develop test specifications and to test. In another, we may be in the test role, testing someone else's design and code. This separation of roles, implicit in a good safety-related development methodology, demands that each team must be able to communicate effectively; as a development partner we are in a good position to see how well, or otherwise, this demand is met!

For illustration we will use a rather 'generic' V-shaped development process model, as shown below[1].

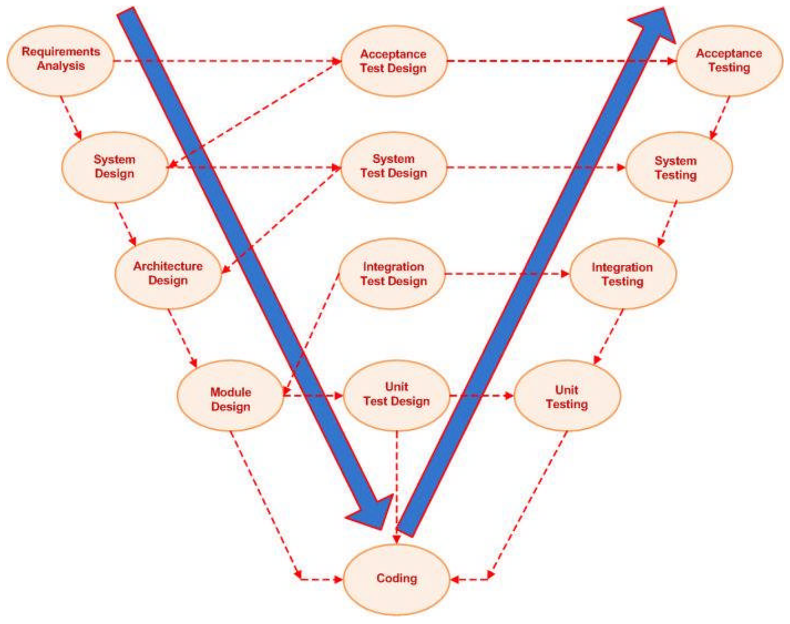

Figure 1

The general idea behind such a process is that the left (or should that be downhill slide?) side of the V represents the translation from specification to code, whereas the uphill struggle on the right represents testing, from individual modules at the bottom, to functional testing of the complete system at the top. At each ‘level’, we are in theory testing on the right what we developed on the left. I’m not promoting it as ideal, but it is commonly understood, or at least commonly claimed as such. This isn’t my diagram, and I might label it somewhat differently, but the original author clearly has both real experience and a sense of humour - whatever path you take through the dashed lines takes you back inexorably to the bottom!

Ideally we ought to be able to test what we develop on the left **before** we reach the corresponding level on the right along the ‘blue’ timeline. We ought to be able to test our requirements specification for consistency and completeness **before** we try coding it. We ought to be able to test our design (eg via a simulation) **before** we get to test its implementation in real code on real hardware.

I work mostly on the left hand-side of the ‘V’, so that is the area on which I will concentrate.

On of our principal areas of activity is the maritime sector. Embedded systems in this sector are characterised by a high number of asynchronous serial channels, connecting sensors such as GPS, radar, speed logs, gyros, inertial navigator systems, meteorological sensors, echo sounders etc. In the defence sub-sector, we can add to this list: periscopes, sonars, electro-optic sensors, telemetry links and more. RS-422, and occasionally RS-485, are the dominant interconnection standards in both the commercial and military sub-sectors, with Ethernet, CAN-bus and MIL-STD-1553 encountered, but still relatively infrequently.

Let us take a typical defence scenario as an illustration, because it clearly highlights the consequences of some common design errors. I will use a fictitious but realistic example for what I hope are obvious reasons.

A Typical (but Fictional) Problem

In a typical defence scenario, most of the embedded systems of the type we are discussing cooperate with the purpose of building some form of tactical ‘picture’:

- Where is the vessel?
- Where (and what) are the friendly, hostile and neutral entities within its vicinity?
- Where will everything be in ‘n’ minutes time?

Decisions may be taken on the basis of that tactical ‘picture’ – decisions that in a real combat scenario may lead to engaging one or more of those entities with one or more weapon systems.

Consider a fictional but representative system. A number of sensors are providing data for a number of ‘tracked’ entities. Data may be typically the bearing and possibly range of a few contacts from a sonar, updated once every few tens of seconds, and several tens of tracks each with range, bearing, course, speed and identity from a radar, the whole data set updated every few seconds. Data is typically transmitted as a single message per ‘track’.

The role of this fictional system is to assess and grade the threat of each track, and transmit the current 10 highest threats to a threat display system.

At the Requirements Analysis stage, the text books suggest these might be represented in a UML use-case diagram something like this:

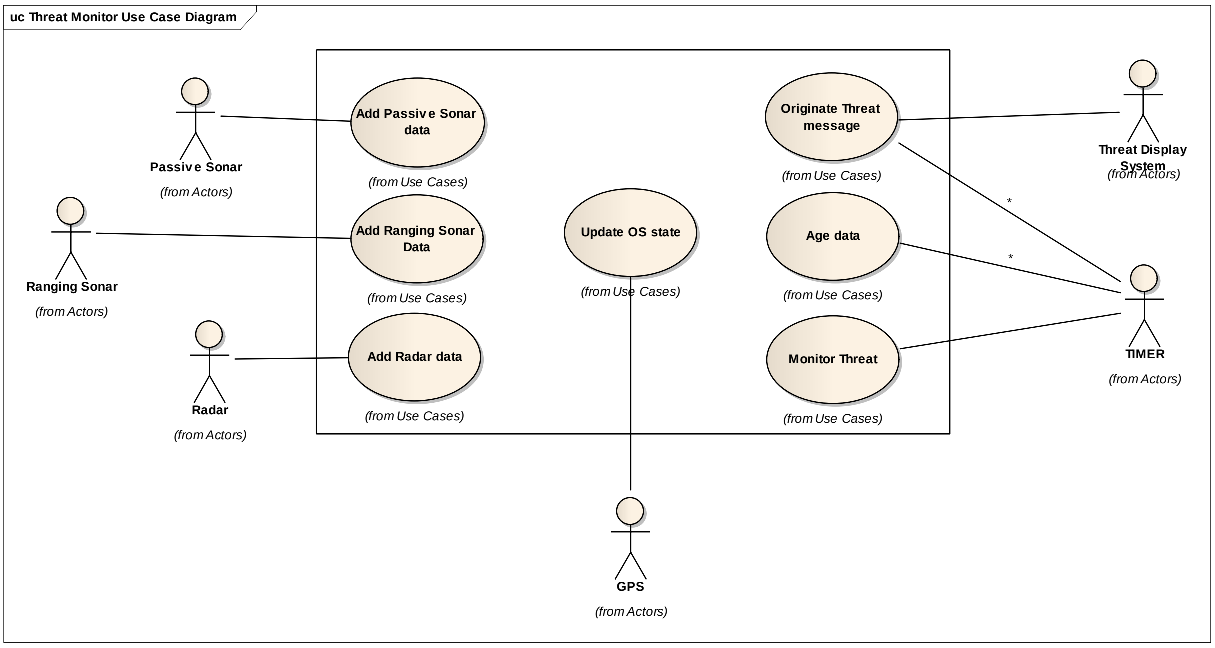

Figure 2

We need to apply some appropriate method to develop the design. We may, depending upon the design method, identify the “significant domain objects” along the way, as something like:

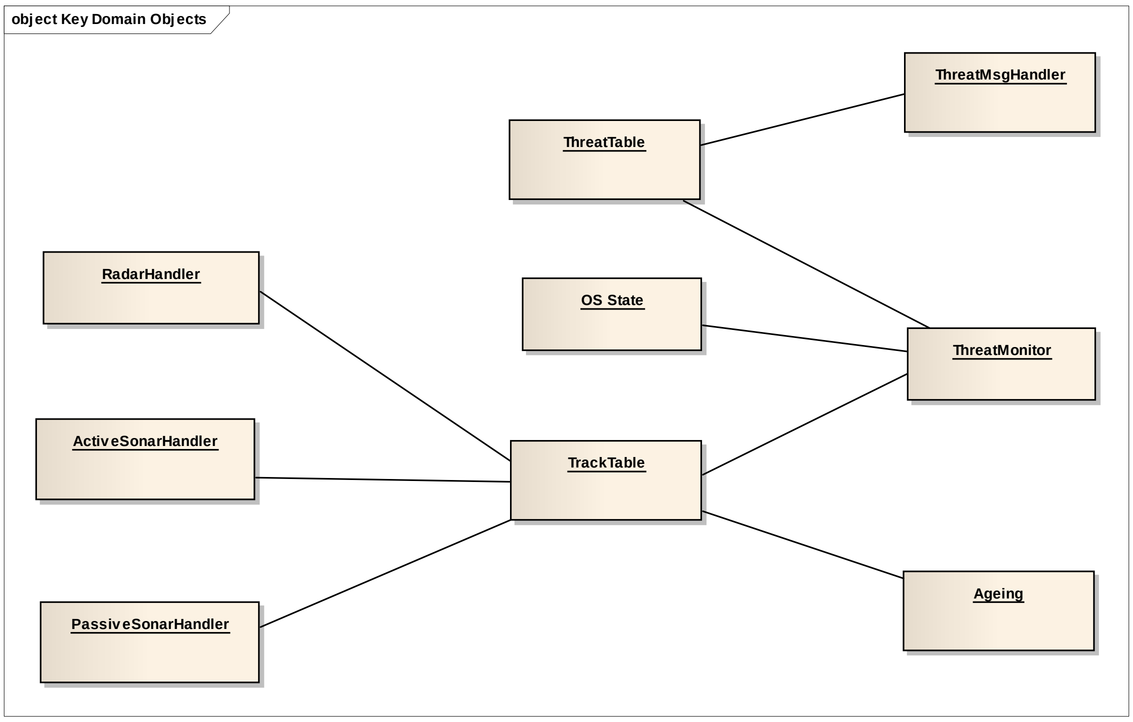

Figure 3

Again, depending upon the design method, we may arrive at some form of Collaboration or Communication Diagram, or more likely, several. This one, for example, shows the collaborations involved in processing a radar track.

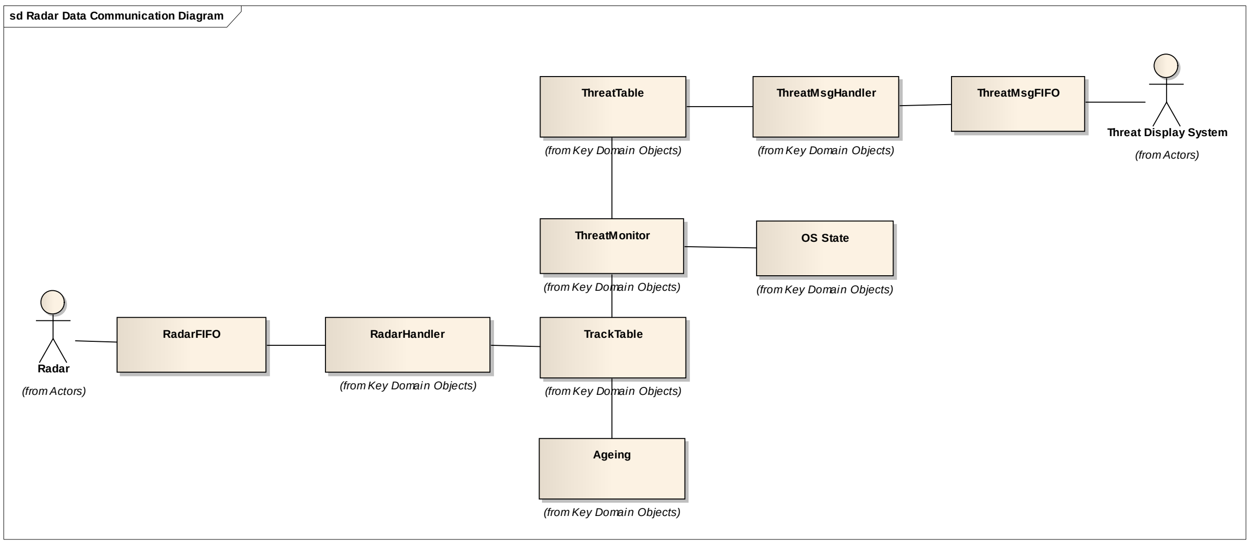
sd Radar Data Communication Diagram
ThreatTable
(from Key Domain Objects)
ThreatMsgHandler
(from Key Domain Objects)
ThreatMsgFIFO
Threat Display System
(from Actors)
ThreatMonitor
(from Key Domain Objects)
OS State
(from Key Domain Objects)
Radar
(from Actors)
RadarFIFO
RadarHandler
(from Key Domain Objects)
TrackTable
(from Key Domain Objects)
Ageing
(from Key Domain Objects)

Figure 4

A Method for Deriving the Design?

But how do we do it? How do we, as designers, get from Figure 2 to Figure 4? How do we know it is any good? How do the V&V team on the right know it is any good? What are the techniques we use? What are the techniques we should be using?

In the Design phase we will need to go a few stages beyond our Communication Diagrams of the style of Figure 4. We will identify messages between objects, methods or functions, and some of the internal structures. Typically we may by this stage have a few dozen diagrams to represent the Design.

Having completed the Design phase, there are tools that can help us test and improve it: analysis tools, executable models within MDD toolsets etc.

Once we are into the Coding phase, we are well supported by modelling tools, coding standards, compilers, static testing tools, software metric tools, automatic documentation generators etc , and beyond that there are dynamic testing tools to take us through into the Testing phase. I’m not suggesting it is easy, but there are plenty of tools to support us.

But what is there to support us actually **within** the Design phase? How do we go about it it? How do we make this leap from Specification to Coding, that we call Design?

From my own experience, we (that is 'we' as an industry, excluding 'us' as a company of course!) generally do it pretty badly. But this isn't just a rhetorical question that I'm about to answer for you – it is a genuine question to all delegates and speakers at this conference – this is what the "Knowledge Transfer" bit of EKTN is about!

What I can answer is how we (as a company) do it, and look at some of the benefits and some of the pitfalls. But first – why does it matter? As long as we can get to something on paper which will give our Coding team something to work with, why does it matter how we do it? Surely there's lots of stuff out there to do it with? Isn't that what UML is for?

To answer the last one first – no! UML is an excellent (in my opinion, your opinions may vary!) tool for assisting with and representing design. It lets us draw all sorts of useful diagrams to analyse and express our design; it doesn't tell us how to do it. It doesn't provide us with what I would call "a method of deriving the design".

Back to "why does it matter?" It matters because in a real-time system, this is the stage at which we can and should resolve synchronisation issues. Data that originates together has to stay together, be used together, and anything derived from it also needs to stay together. That is a requirement that is best met by good design rather than by clever coding. This may be blindingly obvious to all of us in this room – I hope it is. But if it is so obvious, why is it one of the commonest errors we find in such systems? Not only is it one of the commonest, it is one of the hardest to find. The system may pass a particular test a hundred or a thousand times, before throwing up a single error. Code reviews, static analysis and dynamic testing around the "bottom of the V" will not reveal such problems. Because they are introduced "high on the left of the V", they will turn up, at best, high on the right. Sometimes so high on the right, it's turned from a 'V' into a hockey-stick.

Furthermore, these errors pose a safety risk in many systems, in both commercial and defence sectors. In the exaple system, consider the implications of combining a longitude from one update from the GPS with a latitude from the subsequent update; or an identity flag (from a friendly 'track') from one update with the range and bearing (of a hostile track) from the previous update.

These errors happen in real systems. But why? We've had the programming tools we need to resolve synchronisation issues since 1968 when Dijkstra gave us the semaphore[2]. We've added to the toolset steadily ever since, with mutexes, mailboxes, queues, interlocked operations, reader-writer locks etc. But they are difficult to use, and prone to errors, if we leave it until the Coding phase. We need to address synchronisation earlier, so that it is "embedded" in the design.

An Old Solution to an Old Problem?

Wouldn't it be useful to have some sort of approach which specifically addressed synchronisation issues, and gave us:

a. a means of design representation;
b. a method of deriving the design;
c. a way of constructing the software so that it is consistent with the design;
d. a means of executing the constructed software so that the design structure remained visible at run time;
e. facilities for testing the software in terms of the design structure?

Note that we aren't talking about Coding here, just Design. Ideally the above approach would be language independent, and portable between execution environments.

That isn't my list – for a start I've never really even understood what (d) even means! It comes from the Introduction to "The Official Handbook of Mascot"[3]. Mascot (Modular Approach to Software Construction and Test) began life at the Royal Signals Research Establishment in the early 70's and was probably the dominant methodology for real-time design in the UK defence sector from the mid-70's through to the late 80's.

Looking at it in 2009, it has a number of shortcomings:

- a graphical notation unsupported by any current design tools;
- a textual notation that is horribly Pascal-like (apologies to Pascal enthusiasts!);
- it uses only a tiny subset of synchronisation primitives;
- people stare at you as if you have a propeller on your head when you mention it.

But what it retains, and is still relevant today, is "the method of deriving the design" which delivers a design which is, from experience, rigorous and robust in terms of synchronisation issues. It's a method that a designer can follow; he or she can refer to it without having to invent it and describe it. The V&V team can look at the design and compare it to the design rules to check compliance.

To condense several hundred pages into a few bullet points, the essence of the "Mascot method" can be summarised as:

- The design is broken down into Activities and Independent Data Areas (IDA's).
- Activity = Thread or Process
- Synchronisation mechanisms should not be embodied in Activities but in communications elements (Intercommunication Data Areas or IDA)
- Mascot Activities NEVER communicate with each other directly but always through the intermediary of an IDA.
- An IDA is a passive element; it is never scheduled.

- The independent threads of execution of Activities pass through the IDA coding.
- Several such threads may simultaneously be active, or temporarily suspended, within an IDA.
- Synchronisation should take place ONLY in the access procedures of IDAs.
- Synchronisation primitives are limited to JOIN, LEAVE, STIM, WAIT, WAITFOR.

Looking at our Collaboration Diagram again, but this time showing the objects as Mascot Activities and IDA's via a UML prototype, and adding a bit more detail, we get:

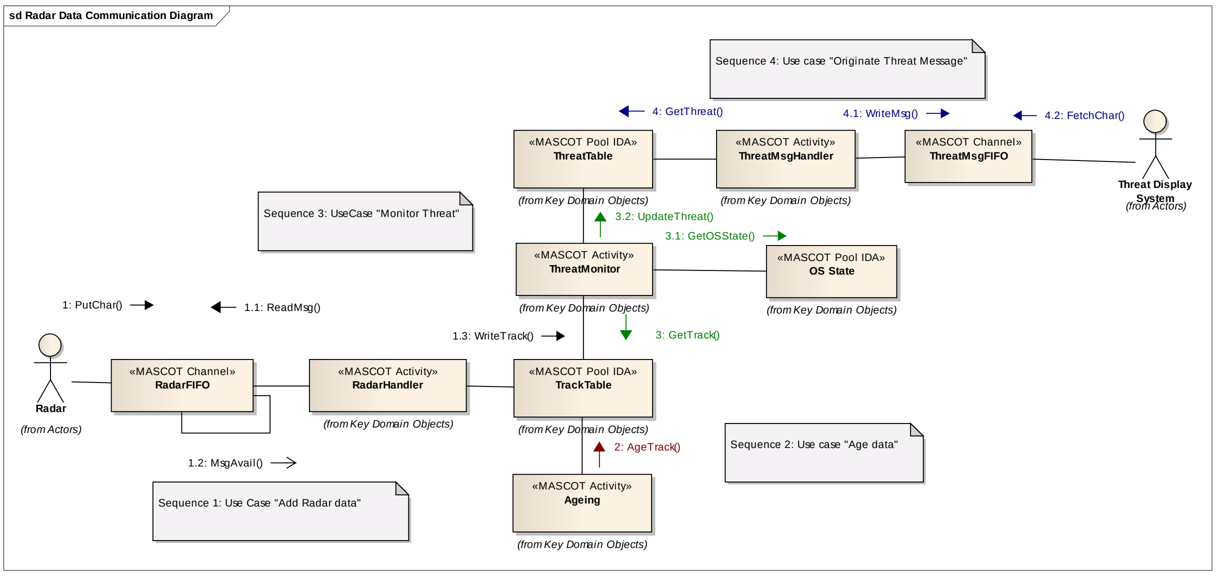
sd Radar Data Communication Diagram
Sequence 4: Use case "Originate Threat Message"
4: GetThreat()
4.1: WriteMsg()
4.2: FetchChar()
«MASCOT Pool IDA» ThreatTable
(from Key Domain Objects)
«MASCOT Activity» ThreatMsgHandler
(from Key Domain Objects)
«MASCOT Channel» ThreatMsgFIFO
Threat Display System
(from Actors)
Sequence 3: UseCase "Monitor Threat"
3.2: UpdateThreat()
3.1: GetOSState()
«MASCOT Activity» ThreatMonitor
(from Key Domain Objects)
«MASCOT Pool IDA» OS State
(from Key Domain Objects)
1: PutChar()
1.1: ReadMsg()
1.3: WriteTrack()
3: GetTrack()
Radar
(from Actors)
«MASCOT Channel» RadarFIFO
«MASCOT Activity» RadarHandler
(from Key Domain Objects)
«MASCOT Pool IDA» TrackTable
(from Key Domain Objects)
Sequence 2: Use case "Age data"
2: AgeTrack()
1.2: MsgAvail()
Sequence 1: Use Case "Add Radar data"
«MASCOT Activity» Ageing
(from Key Domain Objects)

Figure 5

Other Mascot enthusiasts (yes, there are some[4]) will spot that I'm using some Mascot 2 terminology here, rather than pure Mascot 3! I still find the concept of Pools and Channels extremely useful.

The required Synchronisation primitives are, as previously noted, a tiny subset of what most modern execution environments can offer. Whilst that can (and rightly should) be seen as a limitation, it also makes the Mascot "Design Method" portable to almost any execution environment. Table 1 (at the end of this paper) illustrates how the Mascot primitives translate to three environments in which I have implemented Mascot designs.

By retaining the essential "method of deriving the design" from Mascot, but using more modern tools to represent that design (UML), we have a proven and robust means of translating Specification into Design that firmly embeds good design principles. It works well with C, C++, and Ada (and I've heard of it being applied to Java[5]). It adapts extremely well to Object Oriented Design, but is by no means limited to it. It has a proven pedigree in the defence domain. I suspect the essential "method of deriving the design" has passed into many companies' own internal Design Guidelines purely because of the familiarity with it of many of us "mature" designers. Maybe you already use the Mascot "method" without knowing its name or origin.

The Shortcomings

There are, inevitably, shortcomings in this approach. It is much too "hand-draulic"; it relies far too much on manual checking - we do not have any tools with which to test the compliance of the design against the Mascot rules (static testing); we do not have (or I haven't yet found) a good way to communicate the use of synchronisation techniques via UML to the Coding team; it demands that we teach the approach to the Coding team; and lastly, but by no means least, we are relying on an approach that fell into disuse, at least as a published "standard" some twenty years ago.

We also need to consider the future. Twenty years ago Mascot was being taught in universities, at least at MSc level. What are we teaching about real-time design methods now? I'm not actually interested whether or not the next graduate with whom I work knows how to design yet another Internet Bookstore Ordering System!

More Modern Alternatives?

If I knew of a better approach, I would not only adopt it, I would shout about it from the roof-tops! I have looked, but I haven't yet found anything that instils the same level of synchronisation 'discipline' into the design.

UML is becoming so established as the *de facto* modelling language that I would hope to retain it as the means of communicating the Design from one phase to the next. So the Object Modelling Group (OMG), which maintains UML, would seem a natural source, but so far I have found only one candidate technology: MARTE (Modeling and Analysis of Real-time and Embedded systems)[6]. MARTE is relatively new (2007?), but seems to be attracting the interest of most of the UML tool vendors and of some significant players in the safety-critical arena (eg Alcatel, Lockheed Martin and Thales) To quote from OMG (sorry about the spelling!):

"The new standard provided by the Object Management Group, called the MARTE profile, addresses a broader scope than its predecessor (the SPT one). MARTE tackles all the activities of the two classical branches of the V cycle, i.e. modeling and validation & verification. Modeling capabilities have to ensure both hardware and software aspects of RTES in order to improve communication/exchange between developers. It has also to foster the construction of models that may be used to make quantitative analysis regarding hardware and software characteristics. "

I'm still trying to assess the strengths of MARTE. It certainly seems to promise the right capabilities for representing synchronisation in the design. It also seems to promise capabilities for analysing the design. I can't yet say that I have found anything corresponding to what I would call a design method, but that may well be because I'm still struggling to understand the documentation!

I've looked at the major commercial UML tool suites, but haven't found anything with the design rigour we are seeking, at least not anything applicable to real-time embedded development. They support design and analysis, and some even simulation, but don't seem to offer a "method of deriving the design".

I can firmly recommend a book by Bruce Powel Douglass of I-Logix ("Doing Hard Time"[7]). It is an excellent source of advice, but I would argue that that he discusses strategies and techniques for design, rather than anything approaching a method in my terms.

I would stress here that my failure to find what I think we need does not imply it doesn't exist. Maybe I haven't looked hard enough yet; I get paid to design – looking for better design tools isn't usually chargeable to the client!

Conclusions

I believe there is a shortcoming in the way we, as an industry, treat the Design phase of real-time embedded systems. These shortcomings can introduce errors which are difficult to detect, expensive to resolve, and are frequently discovered only late in the Testing phase, if then. If unresolved they can pose a significant safety hazard.

The design methods we need to resolve these shortcomings are well-established, but do not seem to be well supported by current tools and technologies. Or perhaps they are well supported, but I haven't found them yet!

By presenting what I see as these shortcomings, I hope to stimulate discussion about appropriate solutions. What do you use as "the method of deriving the design"? What are its benefits and shortcomings? Does it integrate well with the rest of your development lifecycle, especially Validation and Verification? I certainly have a lot of questions. Do you have the answers?

Table 1 – Mascot Synchronisation Primitives in Real-Time Environments

MASCOT	**MultiTask** **(US Software Inc, c. 1990)**	**μc/OS-II** **(Micrium Inc, 1999)**	**Windows CE 5.0** **(Microsoft, 2004)**
JOIN	getres *mutex*	OsSemPend *semaphore*	EnterCriticalSection *critical section*
LEAVE	relres *mutex*	OSSemPost *semaphore*	LeaveCriticalSection *critical section*
CHECK	chkres *mutex*	OsSemQuery *semaphore*	n/a
STIM	setevt *event*	OsSemPost *semaphore*	SetEvent *event*
WAIT	wteset *event*	OsSemPend *sempaphore*	WaitForSingleObject *event*
WAITFOR	wteset with timeout *event*	OsSemPend with timeout *semaphore*	WaitForSingleObject with timeout *event*
TIMENOW	n/a	OSTimeGet *ticks*	GetTickCount *milliseconds*
DELAY	n/a	OSTimeDly *ticks*	Sleep *milliseconds*

References

[1] Wikipedia, author “M ajth” – public domain.
[2] “Co-operating sequential processes”, Dijkstra, E.W. in “Programming Languages”, (ed.) Genuys, F., Academic Press, London,1968.
[3] “Official Handbook of Mascot”, Joint IECCA and MUF Committee on Mascot, 1987.
[4] http://www.mascot-devel.org/index.html, by Richard Taylor.
[5] MASCOT2002, http://www.object-forge.com.
[6] MARTE, http://www.omgmarte.org.
[7] “Doing Hard Time”, Bruce Powel Douglass, Addison-Wesley, Reading, Mass.,1999

For more information on MASCOT try http://async.org.uk/Hugo.Simpson/