Who's Afraid of Objects for Real-Time?

Onno van Roosmalen

Eindhoven University of Technology

In the coming years a further rapid increase in the number of real-time embedded applications is to be expected. The importance of this branch of computing is generally recognized. Our government considers it of vital economic importance to stimulate the use of embedded software, as appears from a recent 'Technologieverkenning Embedded Software' [1] (with the explicit mention of *software* instead of *systems*). Closer to home; the Computer Science department of EUT has recently proposed the establishment of the Eindhoven Embedded Systems Institute (EESI) at Eindhoven University.

When discussing software development and increase in software production, one cannot refrain from considering object orientation. Object orientation is a powerful software development paradigm that is getting more widely accepted. Currently, every new development in object technology is pervading the IT community faster than the previous one. Java is beating it predecessor (C++) in the rate at which it is being accepted. UML is awaited by many with impatience.

Object-oriented methods are not yet widely applied to the construction of real-time systems. In particular embedded systems with stringent space and time constraints are often being built using conventional programming technology. If object orientation is used at all for such applications, one resorts to established design and programming languages (e.g. OMT and C++) that are not particularly tuned to the needs of real-time systems. The real-time languages that are formulated in the research community do not lead to substantial progress in the state of the practice. Is research not providing the right solutions or is it simply some steps ahead?

Object orientation for real-time embedded applications however is not (yet) moving fast. Major manufacturers of embedded systems are hesitant to commit to OO and are at best in the stage of exploring the possibilities. A recent seminar organized by the CME (the Dutch Centrum voor Micro-Electronica) [2], where Java was strongly advocated as a language for implementing embedded applications, was received by many with scepticism or at best a wait-and-see attitude.

Object Orientation: It is the Best for Real-Time, It is the Worst for Real-Time

There are some undisputed advantages to the use of OO compared to other approaches (e.g., SA/SD). The most important ones are the following: (1) it reduces the semantic gap between software development stages (i.e., it does not require a shift in modelling concepts moving from one phase to another), (2) it increases maintainability through modular continuity and encapsulation of hardware dependencies, and (3) it increases reusability of software components. All these advantages are at least as important for real-time embedded systems as they are for 'standard' applications. Most real-time systems can be modelled after the physical systems they control, thus obtaining an even more seamless path from specification all the way down to implementation and testing. A lesser sensitivity to changing requirements is of particular importance for the development of families of systems. Embedded systems often do occur in families. The possibility to isolate platform dependencies through encapsulation can also be very useful for embedded systems since one could more quickly exploit hardware developments. WindowsNT is a good example of how an object-based paradigm can lead to high portability

and flexibility. Looking at all these advantages, one may wonder what the problem is with using OO in embedded systems.

The price to pay for OO is a substantially higher and less predictable usage of resources. In addition to devouring memory, OO application are usually slower than their conventional counterparts (this may vary strongly with implementation language, though). Two major causes of unpredictability are dynamic binding and the heavy (almost exclusive) use of dynamic object creation and destruction requiring frequent allocation and deallocation of memory. The high and unpredictable resource usage is very disabling when developing real-time systems. Such systems are usually subject to tight resource and timing constraints.

The high resource usage of OO may ultimately be more detrimental to its application to embedded real-time systems than its unpredictability. The reason is that there are many other sources of unpredictability, in fact numerous ones at all levels of a computing architecture. (1) The hardware; e.g., instruction prefetching and pipelining, the use of caches, bus contention on multiprocessor systems. (2) The operating system; e.g., services that supply necessary abstractions for nowadays highly complex systems such as memory management (memory allocation, paging, etc.), and optimization such as lazy evaluation techniques to reduce OS overhead. (3) Language implementation; e.g., (apart from the mentioned allocation and deallocation of memory for dynamic data structures) compiler optimizations. (4) The application and its input; e.g., number of iteration through loops, recursion depth. At all these levels the situation seems to persistently get worse. Although embedded systems are currently implemented on platforms that intentionally do not supply many of the mentioned facilities, the tendency is towards the use of standard platforms. The expectation that the rapidly increasing availability of processor and memory capacity will make guaranteeing real-time performance easier may be entirely misplaced. User's demand for increase of functionality results in a push of real-time applications to the technical limits (consider e.g. multimedia applications) and real-time constraints may actually become more severe. This will be further aggravated by the fact that embedded real-time systems will become less embedded and more and more open. The question is not *if*, but *how* we will learn to live with unpredictability.

It is to be expected that in the course of time new sophisticated mechanisms to deal with memory constraints and timeliness requirements will be proposed. At the same time one will have to use design and programming language concepts for real-time systems, favorably object oriented, that will be durable and can employ such new mechanisms. In my opinion it is, therefore, important during the development of languages to find the best concepts from a *software engineering point of view* rather than the best concepts from a *language implementation perspective*. My impression is that many new language concepts are developed disregarding this point. This may very well be the explanation of the failure of such new concepts to penetrate into the software engineering practice.

OO Languages for Real-Time

Two important types of primitives for object oriented real-time programming languages have been studied extensively in recent years: concurrency and timing primitives [3].

There are, by now, many proposals for including concurrency in object-oriented languages. One important group of such proposals are the *Actor* languages. However, actor models for concurrent object systems are hardly used in practice. Instead a language like Java is gaining wide popularity but uses simple multithreading. It is likely that for embedded systems such concurrency model is preferable for its simplicity, uniformity, and lower overhead.

Including time in programming languages in a way that is syntactically concise, semantically simple and that would allow formal verification of programmed timing behavior, is also a longstanding issue. We will consider it in more detail in the next sections.

Time and Composability

Already in the early seventies, Dijkstra and Wirth [4] stressed the importance of problem decomposition and stepwise refinement as programming strategies. Normally, there are no major problems in applying these strategies: the functional behavior of an object can be made to depend strictly on the explicitly declared interfaces offered by other objects. Thus, based on the interface properties, objects can be combined to form more complex behavior. This is called *composability* and it is essential to enable software reuse. However, if one considers timing behavior of an object, implementation details of other, concurrently executing objects may become important. For example, in a multitasking system the time required to complete a certain computation is sensitive to claims on the processor(s) made by objects that solve completely independent concerns. I will refer to these other objects as the *context*. Hidden coupling, i.e., coupling that is not explicitly on the interface between components, will be called *context dependence* here. Thus, certain timing aspects of objects, in particular execution durations of their methods, may be context dependent. In addition,

timing behavior usually depends on the platform.

Normal, i.e. non-real-time, programming languages provide a context and platform-independent way of describing algorithms. That is, the design decisions and the resulting program are not influenced by details of context and execution platform but solely depend on the specification of the system or subsystem that is to be constructed. This abstraction from platform and context is a prerequisite for composability and reusability. Since composability is an important aim of OO, it makes no sense to formulate a *real-time object-oriented* design method or programming language that does not offer such abstraction.

One can distinguish two types of context and platform dependencies: (1) those that can be explicitly identified in the program text, and (2) those that implicitly influence program design. Many existing real-time design and programming languages are not free of these types of context and platform dependencies. I will illustrate this with two examples.

Example 1 In 1996 I attended a presentation about a HRT-Hood toolset that was developed for ESA [5]. In the toolset, timeliness issues can be addressed early in the design stage by having the designer estimate execution times. A time budget for each deadline (i.e., time available to a deadline) is introduced that must be divided by guestimation over the various high-level instructions (procedures) that must be carried out before the deadline. If this is done for all objects competing for a resource, a schedulability analysis can be carried out. There is no harm in making such estimates to get an impression of the feasibility of obtaining an implementation on a given platform. However, such facilities are actually intended for guiding the subsequent design process and there is the danger of introducing platform dependencies within the design that ultimately hamper the reusability and portability of the designed objects. It is, for instance possible that the subsequent design uses the restricting assumption that two objects reside on the same processor. (Note that a schedulability analysis can only be carried out when all of the objects competing for a processor are known and sufficiently detailed.)
End Example 1

Example 2 Consider a control system that must generate a proper response when an observable in the environment under control reaches a critical value. The observable is probed by the control system through polling. This type of problem is very common in real-time control. One could think of an observable like the methane level in a mine. If this level becomes too high (i.e., reaches a well-defined critical value) all electric equipment (like a water pump) must be shut off to prevent explosions. Formulated more generally, a critical condition C_i on the (polled) environment value S_i requires a response R_i within deadline d_i, i.e.

$$C_i(S_i) \textbf{ at } t \rightarrow R_i \textbf{ in } [t, t + d_i)$$

for all t before which the environment was not critical, i.e.

$$\underline{E}\, t_1 < t : not\ C_i(S_i) \textbf{ during } [t_1, t)\ .$$

Because various such conditions might exist, we use a subscript i to distinguish them. A typical solution adopted in many programming approaches (see for example [6]) is to introduce for each critical observable an object that carries out a periodic inspection of the sensor measuring the environment value.

```
class Mi;
...
thread Ti priority Qi;
do
    with period Pi do
        read(Si); if Ci(Si) then Ri fi
    od
od
...
end -- class Mi
```

The class M_i yields active objects with a thread T_i that reads a sensor value and conditionally takes action R_i. The thread T_i is released periodically with period P_i and has execution priority Q_i. A good choice for the process priorities is the rate monotonic assignment, i.e., the process with the shortest period obtains the highest priority.

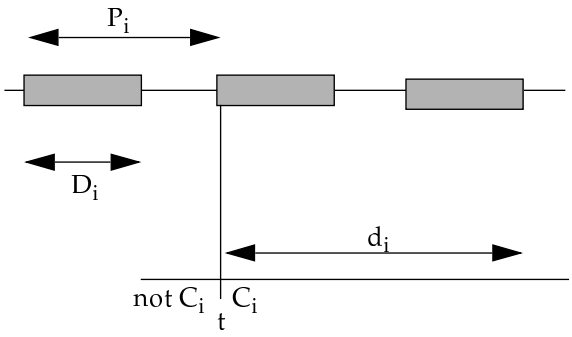

Figure 1: A strictly periodic thread that polls a sensor.

To understand the relation between the deadline d_i and the period P_i, consider Figure 1. The grey areas in the figure indicate the time that the released process can be in execution. During this period it may execute or wait for higher priority processes to relinquish the processor. The maximum size, D_i, of these time slots can be computed when both the complete process set and the time required by the processor to execute the process' statements is known. The worst case with regard to the timeliness of a response R_i occurs when the critical condition

C_i starts at t, just after the process was released and performed a read action. The first occasion the control system is able to detect this condition is after the next release and it requires then at most a time D_i to effectuate the response. Thus the condition on the process' execution is: $P_i + D_i < d_i$.

The above solution has a couple of platform and context dependencies.

- Relative process priorities must be selected and must be hard coded into the program. Thus making processes depend on each other, although completely independent critical conditions may be monitored by them. A process can not be reused, as is, in another context.
- A choice must be made for a division of d_i (the entity that is given in the problem specification) into a period P_i and a deadline D_i for the released thread. When P_i is made smaller, the system load is increased, when it is made larger the available time D_i for the process to execute is reduced and the number of platforms that can execute the program is reduced. Thus, the programmer is enticed to take platform properties into consideration.

End Example 2

Although the foregoing describes just examples, the following shortcomings of present real-time languages and methods are more general.

- Many current real-time programming methods force the programmer to make unnecessary design decisions that put additional timing constraints (i.e., constraints not directly derivable from the problem specification) on the execution of programs.
- Programming solutions must often explicitly be tuned to a particular property of the platform (scheduling regime, processor speed).
- To prove the correctness of a program, platform properties must be made explicit and described separately. It is not sufficient to consider just the program to prove that the specification is satisfied.
- Correctness (in particular satisfaction of timing requirements) can usually not be established for a program component in isolation. This hampers the systematic construction of program through composition.

Platform Independence

Recently, Jozef Hooman and I [7] have described in detail a language extension to deal with timing requirements, that solves the aforementioned problems. It yields programs or program components that can be proved correct independently of a context and platform. Using the timing annotation that is the basis of this approach, I will briefly sketch the solution of the problem described in Example 2.

```
class Mi
...
thread Ti
do
    [t1 :=0];
    forever do
        read(Si) [t2:=t1+di; <t2; ?t1];
        if Ci(Si) then Ri [<t2] fi
    od
od
...
end -- class Mi
```

The timing annotations (in square brackets) express timing constraints that are to be satisfied when executing the statements they belong to. In the example they should be interpreted as follows. The execution moment of each sensor reading is measured `[...;?t1]` and recorded in the timing variable t_1. The timing constraints following from each of these time measurements fall in the next iteration of the forever loop as I explained earlier using Figure 1. This is effectuated by using a second timing variable t_2 which takes the value t_1+d_i using t_1 as obtained in the previous iteration through `[t2:=t1+di;...]`. There are two constraints. First, each read action should be performed within time d_i after the previous one (regardless of whether a response R_i is necessary). This is enforced by `[...;<t2;...]` which demands the execution of the read action to take place before the present value of t_2. Second, if a response R_i is required, its execution moment should also take place before the current value of t_2. Correctness of programs like the one above can be established formally.

Note the differences between our approach and the standard one described in the previous section.

- Only constraints that follow directly from the original problem specification are introduced. The only timing parameter in the program, d_i, is directly taken from this specification.
- Any execution of the program that satisfies the indicated constraints will yield the specification. A strictly periodic execution is possible but not necessary.
- The constraints are formulated on execution moments of statements, not on program blocks (e.g., a complete thread body).
- There is no reference to the particulars of an execution mechanism. There are no priorities or other scheduling directives.

As may be clear from this example, there is no guarantee that an implementation of a program exists on a given platform. In the implementation generation phase it must be established that the selected platform is powerful enough to satisfy the timing constraints that are expressed in the program. This

usually comes down to compiling and scheduling the program. The main difference between this implementation generation and the compilation step in non-real-time system development is the possibility to conclude that no implementation can be found for the selected platform.

Conclusion

I suggest the following list of properties that object-oriented real-time design and programming concepts must satisfy to be successful (in order of importance). They must (1) be efficiently implementable, (2) have simple semantics, (3) be concise (few and orthogonal concepts), (4) have a proper abstraction level (in particular, implementation mechanisms should be hidden completely from the programmer), (5) allow reusability of designs and programs across platforms (even when time is included), and (6) have a formal semantics and prove system.

Many proposed real-time languages have little impact because their primitives do not satisfy the majority of the above criteria. In particular the abstractions are not well enough chosen and semantics very complex. Although Java cannot (yet) be called a real-time language, it already scores high on many of the above points. I believe that its success can be largely attributed to that.

Finally, let me give a (partial) answer to my initial question. Who's afraid of objects for real-time? Sun Microsystems definitely isn't!

References

[1] Report for the Dutch Ministry of Economic Affairs, (in Dutch), *Embedded Software, Onderzoek naar de mogelijkheden en knelpunten bij de toepassing van embedded software door het midden- en kleinbedrijf*, Prisma & Partners b.v., Warnsveld, March 1997.

[2] Handouts seminar *Java in Embedded Systems*, at Zeist, 22 mei 1997, Centrum voor Micro-Electronica, Postbus 1001, Veenendaal.

[3] An interesting overview is given in: W. van der Sterren, *Design of a Real-Time Object-Oriented Language*, M.Sc. thesis, Department of Computer Sciences, Twente University, February 1993.

[4] N. Wirth, *Program Development by Stepwise Refinement*, Comm. of ACM, Vol. 14, No. 4, pp 221-227, 1971.

[5] European Space Agency Contract Report, contract No 11234/94/NL/FM(SC), *HRT-HoodNICE: a Hard Real-Time S/W Design Support Tool*, Intecs Sistemi S.p.A., March 1996.

[6] Y. Ishikawa et al., *Object-Oriented Real-Time Language Design: Constructs for Timing Constraints*, Sigplan Notices, Vol. 25, No. 10, p289, 1990.

[7] J. Hooman and O. Van Roosmalen, *A Programming Language Extension for Distributed Real-Time Systems*, Computing Science Report 97/02, Eindhoven University of Technology, 1997.

Dr. Onno van Roosmalen studied physics at the University of Nijmegen and did his PhD-research in theoretical nuclear physics at the "Kernfysisch Versneller Instituut" in Groningen. He obtained his PhD in 1982. After having filled positions at California Institute of Technology and Yale University he switched to computer science and joined the Distributed Systems group of the Computer Science department at Eindhoven University of Technology in 1987.