1 Introduction

Of course everybody is special! But, among software developers, none more so than developers of real-time software. Theirs is a struggle not merely against the tyranny of literal-minded computer technology but also against an even mightier foe: the messy, unfriendly, unpredictable, driven-by-Murphy’s-law physical world. Slowly and painfully, over decades, they first learned how to overcome software technologies that seemed to be designed for any domain but theirs, even reaching a point where they were able to turn these to advantage. Furthermore, almost uniquely among software practitioners, real-time developers started introducing elements of true engineering for these technologies, such as various forms of schedulability analysis and performance engineering.

And then, just as things seemed to be moving well, new technologies emerged sowing confusion and shattering this established order. First came the object paradigm. Like everyone else, real-time developers could not ignore the onrush of this technological wave that promised so much, but which, like all previous software technologies, never took into account their needs. How should this new paradigm be used for real-time applications? What should be done with all that good stuff that was developed around earlier generations of software technology? Even as these issues are being pondered, yet another technological wave has struck: UML. Once again, the payback potential is impossible to ignore: a common language providing higher-levels of abstraction, widely taught and used, supported by a large selection of tools.
At the same time, real-time developers are facing yet another onslaught from a different direction: With the advent of the Internet and the symbiotic relationship that our society has with computer technology, the demand for highly dependable real-time systems is increasing in leaps and bounds. This explosive growth is matched by an even greater growth in scope and complexity of applications. Systems of several millions of lines of “high-level” language code are no longer considered exceptional. Many are now in the tens of millions of lines of code. Yet, the vast majority of real-time developers use techniques and technologies that are twenty to thirty years old. It is clear that these are inadequate for dealing with the next order of magnitude of application complexity that is emerging and that more powerful techniques based on higher-level abstractions are required.

So, what should the real-time community do with UML? Does it help or hinder? Given the lack of widespread adoption of practically all real-time languages (e.g., Real-Time Euclid, PEARL, Flx) or even real-time dialects of standard programming languages (e.g., RTC++, Ada-95), is a “real-time” UML the right thing to do? If so, what should it look like? If not, what is the alternative?

The panelists were asked to provide their views on these and related issues. (Note that due to poor planning by the moderator, not all the panelists had the opportunity to submit their position papers in time to make the proceedings.)

2 Prof. Theodor Tempelmeier: Real-Time UML

Object-Orientation

Object-oriented design is currently the best way of building systems, and this also applies to real-time/embedded systems. Encapsulation, abstract data types, etc., and other concepts such as generics or templates, are a must when developing complex embedded systems\footnote{The over-cautious approach of embedded C++ [2] is hardly understandable to the author.}. The author is reluctant, though, to accept inheritance as a dominating design principle and prefers object composition instead, where possible.

As for analysis, the situation is less clear. In many cases the object-oriented paradigm may also be reasonable for analysis, but on the other hand there are situations in which the sheer algorithmic complexity suggests an “old-fashioned” functional decomposition approach. For instance, this may be the case in very complex control systems, e.g. in a flight control system\footnote{Note that this holds only for analysis. The design should be object-oriented anyway.}.

In a still wider context, systems engineering (in its original meaning, i.e. not restricted to software systems, cf. [5]) poses again the problem of overwhelming complexity. For large and nontrivial systems from unfamiliar application domains, it is just difficult to understand, what the system does or has to do. Furthermore, in many real-time/embedded systems the hardware/firmware architecture is not fixed from the beginning. Even application-specific hardware, generated from some VHDL descriptions onto FPGAs, has become feasible\footnote{[11]} and is used in practice. From personal project experience, a functional decomposition into subsystems seems preferable in such cases.\footnote{Note that this holds only for analysis. The design should be object-oriented anyway.}