

# ESL Design Education – How much software do we need?

Rolf Ernst

Technische Universität Braunschweig  
Han-Sommer-Str. 66, 38106 Braunschweig, Germany  
[r.ernst@tu-bs.de](mailto:r.ernst@tu-bs.de)

## EXTENDED ABSTRACT

Usually, ESL is considered as microelectronic design raised to a higher level of abstraction. Like in hardware design, it starts with a function specification which is then implemented in hardware and related software components, together providing the required functionality. Implementation considers given physical constraints of, e.g., performance and reliability, and optimization goals, such as cost or power consumption. This is the typical ESL approach to embedded systems design. The required skills for this approach are a good understanding of hardware architectures, abstract transaction-level modeling (TLM), verification, and insights into system level physical properties and their optimization. At this design level, knowledge of the application domain (signal processing, control engineering, ...) helps to find adequate solutions.

At the other end of the spectrum, general purpose computing uses benchmarks to optimize processor systems to typical expected load situations, using well developed performance modeling and quantitative analysis techniques. These techniques are specialized to processor design and are now part of the regular computer engineering curriculum.

The mentioned skills are clearly important for system design and should be part of ESL education, but they are insufficient. In a growing number of embedded system designs, software development is no more employed as a hardware design companion, subject to an overall design process that starts with a unified system description. It rather comes as a discipline that creates its own layered architecture for a new system quality enabling flexibility, dynamic adaptation, resilience and evolution. The exploitation of these layered software architectures, however, very much depends on the underlying hardware and its organization.

To give a practical example, automotive design has traditionally focused on supply chains where the function was specified by the automotive manufacturer (the OEM) while the suppliers developed the respective electronic control units (ECUs) including hardware and software.

Today, each major automotive function has its own ECU leading to a large number of 50 and more networked ECUs in a car. Any update or enhancement goes through incremental steps of the original design process accompanied by stringent and costly test procedures. The new automotive software standard, AUTOSAR ([www.autosar.org](http://www.autosar.org)), assumes a layered software architecture that allows migrating and evolving software components. This is a major challenge to the underlying run time environment (RTE) and hardware platform given that automotive functions are for a large part subject to real-time and safety requirements. Currently, the standard is still evolving in part. Whoever implements and optimizes a hardware platform for such a standard, its components, and RTE should have competence in the principles of software architectures and objects, of signaling mechanisms and real-time system principles. This is even more important for the upcoming generation of multicore processors that expose complex timing behavior.

How can we reflect this trend in the curriculum? Certainly, a basic instruction in software engineering is very helpful to get insight in the software paradigms and development practices. That, however, is not sufficient. CS and EE education are usually limited to very basic application models, such as single FSMs and signal flow graphs, possibly Petri Nets or Statecharts. Challenging combinations of such models of computation are rarely addressed. The resulting dependencies, activation and timing rules which are fundamental for software implementation, for platform load and timing are even less covered. To be efficient, real-time systems design assumes knowledge of these data. Even where real-time is not required, predictability and platform robustness are design goals that require knowledge of these mechanisms.

At our university, we have developed a curriculum that tries to address many of these aspects, but it is far from being complete. Given the complex nature of ESL design, it leaves freedom to combine application, software and hardware topics and stresses practical lab exercises. The talk will outline selected contents of the curriculum.