High-Level Programming Languages are Too Low-Level

A Position Paper

Raj Rajkumar
Departments of ECE and CS
Carnegie Mellon University
raj@ece.cmu.edu

It's All Relative

High-level programming languages are a misnomer in the construction of software systems in general and high-assurance software in particular. High and low are relative terms, and today's so-called high-level programming languages (such as Fortran, Pascal, C, C++, etc.) were "high-level" relative to the norm of the 60s where assembly language programming was common, and entire operating systems were written in assembly language. Times have changed, better technologies have been created and expectations have increased. We posit that two reasons underlie the core problems associated with these so-called high-level programming languages. First, software systems are larger, and are more complex. Secondly, programming abstractions must match the complexity of the software systems being built. In other words, as complexity grows, higher and higher levels of abstraction ***must*** be used. This statement is recursive, one generation's high-level abstraction will be the next-generation's low-level abstraction. And that is how it should be!

Why Not Low-Level?

A traditional argument against the use of higher levels of abstraction is that they impose additional performance overheads which can be costly in resource-constrained systems. For example, compilers for high-level programming languages adopt translation rules and conventions that may lead to sub-optimal performance relative to hand-coded assembly-language programs. This counter-argument does not bear much weight at best, and may even be false. First, with Moore's Law, overhead ratios drop exponentially over time. Secondly, while reaching the performance of hand-coded code was long considered the holy grail of compiler techniques and optimizations, in many specialized domains, automated compilers can actually do better than manually written assembly code. Superior dataflow analysis, control flow analysis and VLIW parallelization techniques have been invented and supported. Lastly, and likely most importantly, hand-coded code is extremely hard to read, understand and maintain. In other words, productivity and verifiability become major problems and infeasible over time. This exact set of arguments applies to why high-level programming languages must be considered too low-level.

The Next Higher-Level of Abstraction

The question then becomes "If high-level programming languages must be considered to be lower level today, what is the higher-level language to be used"? We posit that model-based design is the abstraction with which software systems must be built. We argue that model-based approach is essential to the design, development and validation of high-assurance software. Such high-assurance requirements are critical for many application domains including medical devices and systems, control of critical infrastructures, aerospace, avionics and air traffic control. A "model compiler" (or set of compilers) will translate the system models to the so-called high-level programming languages, which in turn can be compiled to the underlying instruction set architecture.

Model-based Design for High-Assurance Embedded Systems

In the context of software, the model is a precise description of the semantics of the system. First of all, semantics are multi-dimensional, and must describe functional or logical behavior and any operational modalities. Functional behaviors may be modeled, for example, using state machines for event-based systems, dataflow graphs for stream-oriented and signal-processing systems, feedback control diagrams for control systems, graph topologies for wireless sensor networks, and real-time tasking models for hard real-time systems. For high-assurance embedded systems as used in applications such as medical devices, para-functional characteristics must also be specified. These para-functional aspects can include timing behavior, fault-tolerance and reliability, and security properties. Secondly, the properties of the deployment target also can be captured, as are mappings between the software components and the hardware entities. We refer to the latter as deployment characteristics.

Once the semantics have been captured, various kinds of analysis (such as schedulability analysis and reliability analysis) can be carried out on both functional and para-functional aspects. Analysis can also be complemented but not replaced by simulation approaches. The best model-based approaches will allow analytical properties to be composed, not require complete global knowledge, and exploit partitioning technologies that isolate independent sub-systems from one another.

Deployment characteristics will play a significant role in performing detailed quantitative analyses. First, deployment characteristics will be used to perform complete code generation (generating code in desired "high-level" programming languages). Dependencies on operating systems, communication protocols, middleware and specific programming languages will be dealt with by the model compiler. Different back-ends will deal with a host of deployment configurations and their constraints. Secondly, a specialized model compiler test-generator backend will allow the injection of software and hardware faults, and test vectors to validate the behavior of the system when components or subsystems fail. This feature is particularly crucial for safety-critical systems such as high-assurance medical devices. Finally, measurement and profiling tools will be valuable in obtaining quantifying such elements as the worst-case, average-case and best-case values for execution times and spatial resources consumed. Analytical approaches to obtain these target-dependent values represent a long-term research goal.

From a software and system engineering point of view, groups of software components and/or hardware entities can be recursively composed to form larger components, subsystems, systems and systems of systems.