A Design Revolution – Using UML to Generate both Software *and* Hardware

Introduction

Imagine that you have just spent somewhere between $80,000 and $120,000 on the latest and greatest in luxury automobile transportation. You immediately drive your new acquisition to the office and proudly display it to your colleagues. It is in front of these friends that you experience the first taste of the horrors to come when – much like in a comedy movie – you power-up the state-of-the-art quadraphonic sound system and the windshield wipers come on.

Over time, you discover more and more "niggles," such as the fact that – at least once a month – you have to pull over to the curb on the way into work in order to allow your automobile's various computer systems to reboot. This invariably occurs just as the junior office assistant passes you by waving from the comfort of his vintage Trabant (see sidebar).

Eventually you can take no more. You decide to return to the dealership to have your car's software reinstalled to discover that – for reasons known only to themselves – the designers opted for the communications equivalent of an incredibly low-speed analog dialup connection as opposed to some form of high-bandwidth broadband access. The result is that it takes 8 to 10 hours to upgrade your system software, which is obviously a major inconvenience.

Widely considered to be the *"World's Worst Car,"* the East German Trabant celebrated its 50th anniversary in 2007. A contemporary of the USSR's first Sputnik, the two-stroke Trabant had no fuel pump, no fuel gauge, and no oil filter.

As steel was in short supply at the time of its creation, the Trabant's vegetable-fiber body was formed from compressed cotton waste bonded with resin (a prototype formed from compressed cardboard became the *"first soggy car in history"* when it was accidentally left out in the rain).

Believe it or not, all of the above is true (apart from the office assistant being silly enough to own a Trabant) – these are documented cases associated with some of today's most prestigious automobiles. As many influential people have discovered to their cost, their high-end, high-priced (and high-complexity) luxury vehicles actually turn out to perform the role of "beta platforms" that are used to debug the latest electronics capabilities before these functions are eventually disseminated to a wider (cheaper) audience.

A key part of the problem is the lack of an executable system model that lets the designers ensure that they have accurately captured the full requirements and interactions of a design and that the entire system will function as planned. This article first briefly reviews the concept of the non-proprietary industry-standard Unified Modeling Language (UML). Also discussed are the ways in which a new generation of UML-based tools – known as *executable* and *translatable* UML, or xtUML for short – allow system architects and system designers to capture and verify both the hardware and software portions of a design at a high level of abstraction, and to then automatically generate the corresponding hardware and software implementations.

The Unified Modeling Language (UML)

Representing systems at a higher level of abstraction allows designs to be captured and verified quickly, concisely, and efficiently. UML is a general-purpose modeling language that includes a graphical notation used to create an abstract model of a system. System architects and designers can use a graphical UML interface to represent common concepts like classes, components, generalization, aggregation, and behaviors.

UML allows designers to visualize, understand, and verify the workings of a complex system. It acts as documentation, an executable specification, and an unambiguous means of communication between different members of the design team and across multiple design teams. Of particular interest is the fact that UML is extensible because it is possible to add new keywords and descriptions. Remembering that UML is intended as a method for documentation and communication, it makes sense for users in a particular design domain to be able to use the jargon with which they are most familiar. If one is in the "widget" industry, for example, it is possible to augment the UML "grammar" with new "nouns" and "verbs" that facilitate communication with – and understanding by – other members of the "widget" fraternity.

By the early 1990s, there were an abundance of high-level modeling languages, each with their own advantages and disadvantages.

In 1994, Grady Booch, James Rumbaugh, and Ivar Jacobson (widely known as the "Three Amigos") started work on what was to become the Unified Modeling Language (UML), which combined three of the most promising approaches of the time.

The draft UML 1.0 specification was proposed to the international, open membership, not-for-profit computer industry consortium Object Management Group (OMG) in January 1997, and UML 1.1 was officially adopted by the OMG in November 1997.

In the field of software engineering, UML has become a standardized specification language for object modeling. It's important to note that UML is not restricted to modeling software – it is also commonly used for tasks such as business process modeling, systems engineering modeling, and representing organizational structures. Until recently, however, UML has not made great inroads into describing both the software and hardware portions of today's complex digital electronic systems such as System-on-Chip (SoC) devices. These silicon chips can contain hundreds of thousands (or millions) of logic gates coupled with millions (or tens of millions) of lines of embedded software, where the software components include system initialization routines, the hardware abstraction layer, a real-time operating system (RTOS) and associated device drivers, all the way up to embedded application code.

Conventional UML Solutions

Conventional UML solutions are very powerful when it comes to representing the software portions of a design, but they are lacking when it comes to the hardware side of the fence. To understand why this should be so, it is first necessary to understand that UML modeling can be visualized as having two different aspects:

- A highly-abstract general-purpose modeling language that includes the graphical notation used to create an abstract model of the behavior of the system. This may be thought of as capturing the high-level "what we want to do" view of the system (as a point of reference, this is at a significantly higher level of abstraction than C/C++).
- An "action language" that is used to capture the lower-level *"how we're going to do it"* view of the system.

One consideration is that the existing UML specification doesn't actually include a definition of the action language. Instead, this has been left to the developers of UML-based tools and technologies. Typical action languages are implementation languages, (such as C or Java), and effectively constrain the user to working at that level with implementation language constructs, etc. (this should not be viewed as a characteristic of UML, just a characteristic of the way in which UML has been implemented in many cases). The problem is that such an environment is now focused on a particular implementation (e.g. software), and it is very difficult to re-target any portion of the system to an alternative implementation (e.g. hardware).

Another issue is the fact that UML (including whatever action language is used) is supposed to form an executable specification. Many UML-based environments have this capability, but some are harder to use than others. In the case of a UML-based environment intended to represent software systems, for example, it is common to have to translate the entire model into an equivalent C/C++ representation, compile this representation, and then execute it. As a side-product of this approach, it is necessary to have a largely complete representation of the system in order for the compilation and execution to be successful. This can be awkward in the early stages of the design when the system architecture is fragmented and incomplete.

Executable, Translatable xtUML solutions

Now let's consider a more advanced environment for getting real-time embedded applications right the first time, with high quality, using code that represents both the software and hardware portions of the design conforming to any chosen convention or standard.

The key to such a tool's success is that its action language must support a higher level of abstraction than C/C++. This makes it easy to translate these high-level representations into lower-level implementations. In the case of software intended to run on a general-purpose microprocessor core or a special-purpose DSP core, for example, the high-level UML representations (including actions described using the action language) can be translated into one flavor of C/C++. By comparison, in the case of the hardware portions of the design, the high-level UML can be translated into a different flavor of C/C++ that is geared for use by a C-to-RTL synthesis engine such as Mentor's Catapult C (Figure 1).

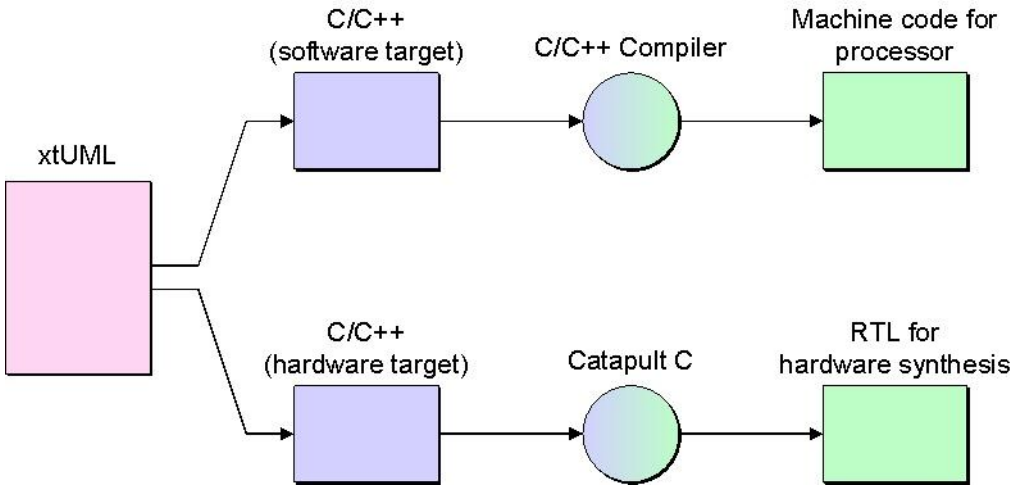

Figure 1. xtUML-based software and hardware flows.

The key point here is that the original model is created using the same high-level action language. All the designers need do is to specify which portions of the system are to be implemented as software and which portions are to be implemented as hardware, and the tools then generate the required outputs accordingly. Of course, this means that the designers can easily re-target different portions of the design to alternative implementations as required.

The above capability explains the 't' ("translatable") portion of the xtUML term. Meanwhile, the 'x' qualifier refers to the fact that these representations are truly "executable." In this case, the environment uses an interpreted model, in which both the UML and its underlying action language are interpreted and executed "on-the-fly". This has a number of advantages, including the fact that it is not necessary to completely describe the system before analysis and debugging can commence.

One way to visualize this is by comparing a program written in C/C++ to one created in BASIC. The C/C++ program must be largely complete (at least to the level of having stubs in place of any procedures, functions, and subroutines) before it can be compiled and run. By comparison, in the case of BASIC, it is possible to enter one or more statements and immediately start interpreting and executing these statements without having to have an entire system in place.

Of course the action language featured by an xtUML environment is at a much higher level of abstraction than BASIC, but the interactive model is very similar. Users can quickly make and evaluate changes on-the-fly, including the ability to modify the values of variables and parameters and immediately observe the results. In addition to fitting well with the way in which engineers prefer to work, it's easy to see that this usage model provides a tremendous boost to productivity.

Summary

This implementation-level independence allows users to quickly and easily experiment with different "what-if" partitioning scenarios (*"now let's try implementing* this *function in software and* that *function in hardware"*) to realize optimal designs in terms of performance and efficiency.

Using a modern xtUML-based environment and tools results in designs that perform the way in which they were intended. This is of great advantage to many industries, not the least automotive manufacturers whose top-tier users are growing tired of acting as beta testers for the latest and greatest "experimental" systems. Imagine the triumph of activating the radio and being able to enjoy one's desired music without the accompanying beat of the windscreen wipers – welcome to the 21st century!