The Concept of Virtualization (Simulation)

The concept of *virtualization* (i.e., *simulation*) is a generally applicable concept that goes beyond virtual machines for programming languages, and its importance has been increasing. Let *X* be any kind of programming language (high-level, intermediate, native machine languages), or digital device (e.g., digital audio systems, thermostats, controller chips for telephones and automobiles), or operating system. The full functionality of *X* can in principle be simulated by a piece of software program.

Software simulators of digital devices are useful for experimental prototyping prior to production of actual hardware devices. Software simulators are easier and less expensive to build, and easily modifiable for experimental testing. They also serve as *executable specifications* of hardware devices. This idea has led to the development of *hardware description languages* and the methodology of *hardware/software codesign*. A programming language can be designed specifically for the purpose of design and precise description of digital hardware devices (including hardware CPUs). Programs written in such a language are software simulators usable for experimental prototyping, and they can also be "compiled" into hardware pieces such as field-programmable gate arrays or VLSI circuits.

A software simulator of an operating system *X* can be built which runs on top of another "real" operating system *Y*. Then the user can enjoy benefits of two operating systems at once: one is the "real" OS *Y*, the other is the "virtual" OS *X*. For example, application software developed for the OS *X*, such as graphic windows and database systems, can be run under the virtual OS *X*.

Indeed, the concept of virtualization/simulation has been frequently used in computation theories involving various types of automata and Turing machines. For example, the fact that the deterministic Turing machines have exactly the same computing power as the non-deterministic Turing machines can be proved by constructing a simulator (i.e. virtual machine, interpreter) for any given non-deterministic Turing machine inside a deterministic Turing machine. Likewise, the fact that the one-tape, one-head Turing machines have the same computing power as the multi-tape, multi-head Turing machines can be proved by constructing a simulator for any given machine of the latter type inside that of the former type. What is called a universal Turing machine is nothing more than an interpreter of Turing machines implemented in a particular Turing machine, completely analogous to an interpreter of a programming language *X* implemented in *X* itself, for example an interpreter of Java written in Java itself.

An alternative method to prove equivalence of computing power is "compilation": Show that each non-deterministic Turing machine can be translated (compiled) into an equivalent deterministic Turing machine, and show that each multi-tape, multi-head Turing machine can be translated into an equivalent one-tape, one-head Turing machine, etc.

There is a significant theoretical insight: The successful implementation of a simulator of a language/device *X* inside a different language/device *Y* proves that the theoretical computing power of *Y* is at least equal to that of *X*. This proves, however, nothing about comparative *efficiency* of *X* and *Y*, or of the simulator of *X* and the "real" *X*. In the latter comparison, simulators of *X* have turned out to be no more efficient (usually slower) in execution than the "real" *X*. This appears to be a fundamental, universal principle of digital machine simulation. For example, a software interpreter of a language *X* is slower than a direct, hardware implementation of it; a software simulator of a hardware digital device *X* is slower than *X* itself; a simulator of an operating system *X* implemented on top of another operating system is no more efficient than a direct implementation of *X* (assuming that both operating systems are implemented on the same hardware platform).