Meta-level compilation (MC) involves using compiler extensions to catch domain-specific errors. These errors can be described as higher-level “system rules” that must be obeyed in order to achieve good performance or correctness. Enforcing such rules has been a problem in traditional programming systems. Manual inspection is tedious and error prone, testing requires a large time investment for sufficient coverage, and formal verification depends upon models that are difficult to construct and may not correspond exactly to the code. Static analysis of source code solves many of these problems, but compilers don’t know the system-level semantics and are thus unable to check for these sorts of rule violations. MC attempts to remedy this situation by allowing system developers to write compiler extensions so that the compiler can verify that the system rules are being followed.

The compiler extensions are written in metal, a high-level state machine language. The metal program describes possible program states and the transitions between them; entering an error state at any point during the analysis indicates a possible rule violation. In order to use a metal program, it is first compiled by the metal compiler and then dynamically linked into the C++ compiler when the actual program source code is compiled. Metal allows for either flow-sensitive or flow-insensitive analysis, so either can be chosen for a given rule depending on the nature of the rule. For flow-sensitive analysis, pruning is used so that the analysis does not take prohibitive time.

Most of the paper is devoted to presenting examples of the compiler extensions and the results of using them on Linux, OpenBSD, Xok, and FLASH. The rules generally fall into the categories of describing legal orderings of operations, or describing contexts in which certain operations may occur. The first few examples from the paper can be checked locally within a function: “assertions should not have side effects,” “memory should not be used after it is freed.” However, some rules require a global context, such as: “kernel code cannot call blocking functions with interrupts disabled.” In order to use metal for this global checking, two passes over the source code are needed, one to determine all functions that call functions that may block, and a second to compute the transitive closure of this set of functions. Finally, as a bonus, MC can be used to find opportunities for optimizations in addition to finding bugs. An example in the paper shows how MC can find places in FLASH programs where instruction count can be reduced.

The results presented in the paper are encouraging; MC could find a variety of bugs in four real software systems. One of the first things to consider in evaluating a system like this is whether the burden on programmers would be too great. From the information presented in the paper, it seems like metal would be acceptable, since the specifications are not very long and need only be written once for each property to check. The metal language also looks like it wouldn’t be exceedingly difficult to learn, but it would likely take some experience to write more complex extensions. Another issue is the tradeoff between soundness and completeness. Their analyses did produce a non-trivial number of false positives, but they were able to explain the reasons for them. Some way of removing false positives automatically without source modification would be nice, but of course this should not be at the expense of missing actual bugs. Also, as the authors stated, the prototype implementation seemed a little clumsy because it was based on a C++ compiler even though they were checking systems in C. Porting to a C compiler, as they planned to do, would fix this problem.

Overall, it seems that MC could be used to catch quite a large class of programming errors. It is only a matter of time before we see tools such as MC in widespread use to improve software reliability.