Checking System Rules Usign System-Specific, Programmer-Written Compiler Extensions[1] A Few Billion Lines of Code Later, Using Static Analysis to Find Bugs in Real World[2]

Yoon-Kah Leow

March 29, 2011

1 Paper Objective

Meta-level compilation is a technique that introduces meta-level information to the process of compilation to check for system-rule violations. The paper provide innovative methods to dynamically link meta-level semantics to the compilers in the form of checkers. These checkers take after the form of high-level state machines defined by a language named, *Metal*. Static analyses are employed while the checkers scan the code for system-level violations. The authors also provide insights in the commercialization of the fruit of their research, *Coverity*, describing hiccups that they encounter while advertising their research-based product in the commercial context.

2 Motivation to use Meta-level Compilation as System-Rules Checker

There are currently 3 main approaches to system-rules checking. However, each of them has their disadvantages which makes system-rules checking a pain.

- **Formal Verification**
  Formal verifications constructs a specification based on the rules of the source-code that it is verifying. These specifications are often abstractions that attempts to mirror the system-rules. However, due to the large variations in system-rules that exist in a language, these abstractions are often too simple with missing features. Furthermore, it is also a huge task to construct such an abstraction to a fine detail as it requires up-to-date experts in the specific system to accomplish it. Although this approach allows one to spot violations which is hard to detect with other means, it requires too much effort to define a comprehensive specification.

- **Testing**
  Testing requires developing test-cases to verify each execution path. This method scale poorly as the number of execution paths increase exponentially with the code size. Hence, testing will miss a lot of areas in the code especially when testing engineers are unfamiliar with the code base that they are testing. Furthermore, locating the error in the code is hard as test-cases do not provide pinpoint the exact failure.

- **Manual Inspection** Developers manually examine the code for errors in manual inspection. Although this approach can spot errors at all level of semantics, it is often tedious with large

code bases. Furthermore, the efficacy of this method varies widely with the person that is performing the check.

During compilation, the compiler is binded closely to the system at code-level granularity. This allows the compiler to reach areas in the code that is hard to spot by other means. Despite this advantage, as there are no requirements for the compiler to understand the code's higher semantic levels, the compiler remain ignorant to the meta-level of the code. Meta-level compilation makes use of this advantage by introducing mechanism to facilitate system-wide checking. Due to the nature of compilation, a single checker can easily reach the entire source-code. The authors also introduced definition languages, rule checking techniques to ensure a much pleasant yet thorough testing experience.

3 Meta-level Compilation

A state-machine language *metal* is used to define extensions (i.e., system-rules checkers). These extensions are dynamically linked into the GNU g++-based compiler, *xg++*. Static analysis is performed based on rules defined in a checker.

3.1 Defining a Checker using *Metal* language

The following information is required in order to define a checker in *Metal* language.

- Declare any argument format derived from the code used for pattern-matching.
- Define patterns observed within the code that triggers a state transition.
- Define discrete states with possible transition paths that is associated with a pattern.

3.2 Caveats of Meta-level Compilation Checker

- As meta-level compilation is based on static analysis, it cannot spot run-time errors.
- As meta-level compilation is based on a checking mechanism (i.e, in contrast to formal verification), it cannot spot system-rules that is used in a unique way.
- False positives are inevitable as static analysis is inherently localised checking.
- As the dynamic compiler, *xg++*, is based on *g++*, it has compatibility issues with some relaxed C types and GNU extensions.

4 Meta-level Extensions

4.1 Checking for Assertion System-rule Violation

4.1.1 Expected System Behavior

- Debugging assertions should be turned off in production code.
- Expressions that evaluate to *false* in assertion is a bug.

4.1.2 *Metal* Checker Implementation

- Traverse through the code and search for pattern {*assert(expr);*}.
- Upon pattern matched, *expr* is recursively applied to the state-machine.
- The state-machine checks for expressions that resembles a function call, assignments, and pointer arithmetic. An error is flagged when such an expression is found.

4.2 Checking for Pointer Usages

4.2.1 Expected System Behavior

- System call routines must validate application pointers before use.

4.2.2 *Metal* Checker Implementation

- Traverse through the code and mark all pointer variables.
- Unmark each pointer variables after an assignment or passed as an argument to a function which expects an uninitialized pointer.

4.2.3 Deployment Observations

- Detected 18 errors in the exokernel.
- The lack of global semantics leads to false positives when the code implements an unique way of verifying null pointers.
- Kernel backdoors leads to false positives as the checker assumes a universal approach.

4.3 Checking for Memory Allocation and Deallocation

4.3.1 Expected System Behavior

- Returned pointer of *malloc* must be checked before use.
- Cannot use deallocated memory.
- Always deallocate memory before exiting an execution.
- Memory allocated needs to be larger than the object.

4.3.2 *Metal* Checker Implementation

- Traverse the code and initialize each pointer variable to an unknown state.
- Checks for allocation, deallocation and freeing of each pointer variable and assign the states not_null, null and freed respectively.
- Errors are flagged when pointer variables in the null and freed states are used.
- Errors are flagged for each pointer variables in the not_null state when code exits.
- Warnings are flagged when pointer variables in the freed state are deallocated.

4.4 Global Analysis Extensions

4.4.1 Expected System Behavior

- The kernel will deadlock if a blocking function is called after interrupt disable or acquiring a spinlock.
- A race condition is possible if a blocking function is called before the module's reference count is set properly during dynamic module loading.

4.4.2 *Metal* Checker Implementation

- Traverse the code and mark every potential blocking function.
- Generate flow graphs for each basic block while traversing the code.
- Generate a global call graph with the resultant flow graphs generated.
- Traverse the call graph and search for all functions that invokes a blocking a function directly and indirectly.

4.4.3 Deadlock *Metal* Checker

- Initializes the state of all function calls to clean.
- Traverse the code and switch state to enable when it encounters a interrupt disable or spinlock acquire. The state is switched to disable vice versa.
- Search the list of blocking functions before entering state enable or disable.

4.4.4 Deployment Observations

The deadlock checker is used to test Linux and 79 potential deadlocks are found. Local errors are mainly caused by unawareness of the system behavior in inexperience developers. Global errors (i.e., errors that affect the entire kernel) are caused by code traceability issues due to large code size that developers have to verify. This also imply that it is desirable to have an automation tool like meta-level compilation to check for errors if the code size is large.

4.5 Optimization Checker Implementation

The authors demonstrate the capability of meta-level compilation by extending their tool to improve highly optimized FLASH machines' cache coherence protocol. Checkers check for the following items.

- Check for irredundant buffer spaces during message sending.
- Check for default buffer lengths that are larger than required.
- Check for duplicating message headers and recommend an XOR operation over consecutive messages that can recycle the same header.

4.5.1 Deployment Observations

Some of the potential optimizations will require a major code change. Potential errors might be introduced if a developer who is unfamiliar with the code works on implementing them. This implies that it depends on a situation whether a developer should choose to implement the recommendations from the checker and a good bug to fix is probably one which is easily accessible and uncontroversial.

5 Coverity Commercialization Experiences

Coverity is a bug-finding tool based on the authors' research. They commercialize this tool and gain invaluable insights to the current attitude of the commercial world to performing code testing using a 3rd-party bug-finding tool.

5.1 Reseach Versus Commercial

- In research context, a small number of programmers work on a single code base and programmers expects that errors in a code is inevitable.
- In the commercial context, a large number of programmers are required to work on a variety of code bases and programmers assumes expected results from a 3rd-party bug-finder tool.
- In the commercial context, programmers are often ignorant to the code that they are testing. Due to this lack of interest, confusing bugs flagged by the tool are often neglected and denoted as a false positive.

5.2 Difficulty Faced by Coverity Developers

- Coverity developers have no access to the actual code for testing due to confidential reasons.
- Customers expect a different testing environment from Coverity.
- Coverity is subjected to variations in flavor, version compatibility, and proprietary compiler extensions as customers often associate correct code with successful compilations.
- Coverity developers realize a huge variety of compiler variations in the commercial context that it is almost an impossible task to support them all.
- It is critical to ensure that Coverity is bugs-free to eliminate false negatives.

5.3 Advices to Bug-Finding Tool Developers

- A hard-to-understand bug is not a good bug as it will lead to confusion and misinterpretation.
- Do not fix out-dated bugs that has already exist for a long time as it might introduce more bugs.
- Modifications to Coverity should try to limit "churn" in order to guarantee trust-worthiness in the customers.

6 Conclusion

Meta-level compilation poses a revolutionary approach to bug finding. Despite that this approach does not guarantee 100% bugs-free, the easily scalable architecture has demonstrated commendable results over existing complex systems like Linux, OpenBSD and Xok exokernel. However, the commercialization of Coverity also impose very different expectations from the commercial context on the tool. In my personal opinion, I feel that meta-level compilation is in the right direction of creating a the next generation bug-finder tool due to the close coupling to unreachable large code bases during compilation. However, the *Metal* high-level state-machine definition language still requires quite a lot of system/code domain knowledge to create extensions/checkers. As a company often has a dedicated testing and validation department to perform extensive tests, test cases are often created based on marketing requirements which are often too high-level. Hence, this tool might still require cross-departmental support from the research and development department to define useful customized checkers pertaining to the system being tested.