Automatic Verification — What We Can Get

Bogdan Rusu

Department of Computer Science, West University Timisoara
bogdan@info.uvt.ro

Abstract

The talk tries to figure out some fundamental aspects regarding the structural assemblage of a desirable automatic verification tool passing under the surface of the comparison between projects like **Metal** and **Ballista**. [1]

The aim of this talk is rather to point out some comparative aspects in considering both formal and more straightforward techniques based methods of investigating the correctness of code in an automatic fashion, focusing mainly on CMU **Ballista** and Berkeley **MC-Metal-xgcc** projects, picked because they represent two variants which are different in they goals and underlying principles, but still share a few intriguing similarities in the background. Possibilities, limitations, directions of generalization are discussed while some more fundamental issues — like the role played by code - data duality (as a source of generativity), the simulation statute, the presence of order and chaos in program sources, the meaning of abstracting code elegance by (co)evolutionary mechanism run on short examples and counterexamples (if possible), the power and the decoys of automating the flaws detection task, the tradeoff with the amount of false positives and the increasing difficulty of avoiding undecidability, the incompleteness of analysis and the opposition between bug identification and certifying the correctness of a code[2] — are tried to be captured through some suggestions. A few characterizations of the properties of the component actions of an automatic verification method and some classification are made. A particular interest is devoted to key-concept genesis in a certain approach — for example to the "belief propagation" interesting idea in **Metal**.

Dawson Engler's project at the Stanford Univ., MetaCompilation was intended to intercept *Linux* kernel bugs (considered due to the high programming discipline of the team and the code age to be the most reliable piece of software available in the most error-free world of open-source classical operating systems) by extending the gcc compiler with kernel-specific information (from where the meta-compilation name) by a language called **Metal** (*Meta*Language) — in some way a sort of intermediary between static and dynamic testing. **Metal** programs, which describes finite state automaton, are compiled and transformed into *C* programs and then executables, dynamically linked to gcc. Then, when **xgcc** is executed for compiling the kernel, the extensions written in **Metal** are sequentially executed (each extension analyze each compiled function). The finite automaton is executed in the abstract by compiler on the program (on each procedure). Conceptually, the compiler executes every possible path in procedure and makes the finite state transitions until meets the indicated patterns — if one path leads to the error state, the compiler will notify this through an error message. In fact, **xgcc** don't follow all the possible paths, but simultaneously compute all the states in which the automaton can be in every of the program points by a method called abstract interpretation. The analysis described in **Metal** are intra-procedural, which imply incompleteness of analysis and the false positives problem. The next step was to automatically infer the rules — in a certain form (pattern) class — by statistical means (one of the advantages of the desired system was the possibility of detecting flaws in a code unknown in details to the researchers), which raises

[1] *Keywords and phrases:* Model checking, automatic verification tools, model abstraction, abstract interpretation, belief propagation, metacompilation, comparison between automatic verification systems, code correctness.

[2] Both examples, **MC** and **Ballista** are merely concerned with error detection than (as opposed to) assuring correctness certificates.

on another level the more evolved question (have pointed out the presence of some rules, but still don't know if these rules are complied or not) if the use of some construction does not imply the fact that the programmer has a certain belief about how the code may look alike — for example, if one compare a pointer with zero, one expect that such a pointer can have a zero value, but if the pointer is dereferenciated immediately after declaration, this means that the programmer don't expect a zero value for the pointer in this point (these two beliefs are in contradiction). **Metal** defines a few simple rules regarding the propagation of beliefs into the program (for finding which of them are conflicting) — which seem to be the most subtle, elegant and original concept in this contribution.

Philip Koopman's **Ballista** project from Carnegie-Mellon Univ. (*not* from Computer Science Dept., but from the Complex Software Systems Group of Electrical Engineering and Computers Dept.) deals with testing of modules (which gives services to others programs through defined interfaces) by fault injection techniques; they have verified in such a way operating systems kernels (mostly, but not only *Unices*), *C* standard library and *ORB* (Object Request Broker) component implementation in *CORBA*. (These packages share a certain dose of paranoiac suspicion (mistrust) regarding the programs demanding their services in a system which runs.) **Ballista** (the name of an ancient weapon was adopted for describing the way a module is bombarded with data) sees modules like black boxes and generate a few hundreds or thousands combinations of illegal parameters. It creates a separate process (which executes the command asking for the service using the illegal parameters): if the process terminates, then the received code is returned; else, the master process consider after a while that the test process is no longer active (alive); for each test the results are recorded on disk (in case of some disaster), preventing their loss. A classification of failures is introduced, forming the acronym `CRASH`: catastrophic, restart, abort, silent, hindering (which adds to the correct case, of course) — the tool is intended for comparison between systems which implements the same interfaces (like all the *Unix* flavors relating to the implementation of the functions described by the POSIX standard). Now, the main idea is to classify all the function arguments types (20 data structure types are enough to test all the *Unix* services on all platforms); **Ballista** keeps also a list of exceptional file descriptors values (like FD_CLOSED, FD_OPEN_READ, FD_DELETED, FD_NONEXIST, FD_EMPTY, FD_PAST_END, FD_TERM, FD_MAXINT, FD_NEG_ONE, etc.) — for generating a parameter with the value FD_CLOSED (a closed file), it will call a procedure which opens and then close a file and will return the obtained ID. This is the first simple elegant idea: to get the test cases from the data types. Then a problem raises: how do you know which of all possible values combinations for arguments are illegal and which are not ? A voting system is established — if more equivalent (assuring same functionality) systems are tested, if none of them does not reject a combination (by generating an error), then the combination is considered to be correct. Another heuristic is coined for handling the silent errors: if at least one of the systems report an error, then the combination is considered illegal and it is said that the others silently fail. (This has surely counterexamples like the next one: *QNX* does not allow spaces in filenames — valid in POSIX standard — and the functions called with such cases will fail; the situation will lead to considering this combination as a silent error for all the others systems, which is wrong, because in fact they will not report the inexistent error because they run correctly...) The most unhandable are the hindering errors (if one writes 0 bytes into an inexistent file, the standard does not specify which error has to be returned: if the size test is made first, the function will succeed, since it has indeed written 0 bytes, but if the function finds first that the file does not exist it may return an error).

Based on this principles, the system has detected a variety of bugs, producing also some relatively surprisingly results (different types of surprises!) in regard with the major operating systems kernels et oth. The system has also obvious limitations (it catches only the simplest errors, due to the smallest pieces of code). **Ballista** does not intercept errors like the ones related to the heavy load of the system, the ones depending to the system internal state (like memory leaks) or (especially) the ones originated into the interaction between many components (*the* source of chaos :-) . . .) — deadlocks caused by the execution of functions in a certain order will escape (because each test evaluates one single function at a time). It seems that the greatest power and the greatest limitation of **Ballista** had the same source: the test modules are viewed as black boxes — the conditions related to the error generation are taken into account, but the correct answer is in fact not known.