Precise and Scalable Program Analysis at NEC Labs

Aarti Gupta and Franjo Ivančić
NEC Labs America, Princeton, USA

The Systems Analysis & Verification Department at NEC Labs engages in foundational as well as applied research in the areas of verification and analysis of software and embedded systems. We have developed several tools and frameworks for scalable and precise analysis of programs, some of which are now used within the company on large software projects. This brief report summarizes the challenges we faced and our attempts in making them usable.

Overview of Current Research

Starting from SAT-based model checking, which we had successfully applied to large hardware designs, we have focused on adding static program analysis and testing-based dynamic techniques to develop a portfolio of verification and bug-finding tools for source code programs in C/C++. More recently, we have also investigated testing and simulation-based techniques in model-based design development of software systems. Our current research activities are broadly classified as follows:

- *Sequential program verification*: We have developed F-Soft, a cooperative, staged framework of various static analyses, abstract interpretation, and model checking techniques. It can check C/C++ programs for runtime errors (pointer access violations, buffer overflow, memory leaks, string errors), standard library API usage, and other user-defined assertions. An in-house product based on F-Soft, called VARVEL, is currently in use in NEC.
- *Concurrent program verification*: We target finding concurrency-related bugs in multi-threaded C/C++ programs, such as data races, deadlocks, atomicity violations, and missed notifies. A tool called CoBe (Concurrency Bench) leverages a combination of precise dataflow analyses and symbolic model checking. A different tool, called Fusion, combines dynamic and static verification techniques. Given a test case, it executes the program to derive a Concurrent Trace Program (CTP), which is then used to explore alternate thread interleavings by using static analyses and SMT-based checks. Another tool, called Contessa, explores the space of input values and thread interleavings in CTPs.
- *Embedded system/software analysis:* We utilize symbolic execution techniques to check robustness of simulations for Simulink models and to improve their test coverage. We have also extended F-Soft to handle floating point numbers more precisely, to check for numerical stability and other related issues in embedded software.

For more information and related publications, please visit
http://www.nec-labs.com/research/system/systems_SAV-website.

Challenges and Lessons Learned: Technology Perspective

It is well-understood that precision and scalability are conflicting requirements in general. We believe, however, that both accuracy in program modeling and efficiency of analysis are crucial to make verification usable in practice. Plus, with the advancements in SAT and SMT solvers, there is hope for

larger models with increased precision, since these solvers are effective at ignoring semantically irrelevant facts. Therefore, we believe that *precision should not be given up too early*. Rather, it is good for a verification platform to provide many "knobs" for making this tradeoff, both in terms of program modeling and in verification techniques.

Although automated abstraction-refinement techniques (e.g. predicate abstraction and refinement) also aim to make verification more efficient by using coarse abstractions, and then refining on demand when higher precision is needed, in practice, sometimes it can be difficult to recover global accuracy by using local refinements. In our experience, predicate abstraction-refinement did not work very well for program properties related to memory safety (pointer validity, array buffer overflow, memory leaks, etc.) because the number of pointer-alias predicates typically blew up, or multiple refinement iterations were costly. We found that a custom memory model for the program with a global pointer analysis was more useful in retaining an adequate level of precision in static analysis.

Another fact of practice is that with millions of lines of code and thousands of automatically-instrumented properties, there is no silver bullet among techniques. Therefore, we believe it is useful to provide a *staged framework of cooperative techniques*, where cheaper analysis techniques are used first, to discharge easy-to-prove properties and simplify the model for downstream application of higher-precision analysis. This has worked well for us in F-Soft, as shown in Fig. 1. We use abstract interpretation to automatically derive program invariants over increasingly more precise domains, such as the interval domain first, and the octagon domain next. These invariants are used to prove correct about 70-80% of properties related to memory safety. They are also used to simplify and reduce the model, on which the remaining properties are checked by a SAT-based bounded model checker, to report violations. Indeed, in VARVEL (the product based on F-Soft), abstract interpretation is mainly used for proving properties, whereas model checking looks for relatively simple path sensitive proofs and otherwise focuses on finding concrete witnesses. The important point is that these cooperative techniques work on the same underlying model of the program, including the memory model. Therefore, information (e.g. invariants) derived using cheaper techniques can be utilized effectively to improve performance and scalability of the more expensive techniques (e.g. model checking).

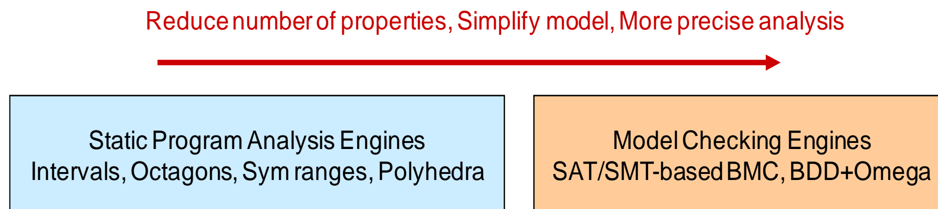

Figure 1: Staged Framework of Cooperative Verification Techniques

A cooperative analyses framework has also worked well for us in verifying multi-threaded programs. In CoBe, we use partial order reduction and synchronization constraints to derive a transaction graph that captures all relevant thread interleavings. This bootstraps the application of abstract interpretation to derive invariants, which are used to further refine the transaction graph by removing unreachable

nodes. For data race detection, this static analysis framework serves to reduce the number of false warnings captured by an initial lockset analysis, and facilitates the use of model checking on the remaining warnings to generate concrete error traces. In Fusion too, we use a common symbolic model for a concurrent trace program (CTP), derived from a given test execution of a multi-threaded program. We model the inter-thread data flow, concurrency primitives, and correctness properties uniformly as happens-before constraints in our model. These can be then tuned to different analyses – from static cycle checks to eliminate warnings, to precise handling of data flow in SMT-based exploration to generate concrete error traces.

Finally, our efforts for Simulink/Stateflow models are also based on extracting a symbolic model from a concrete simulation, and applying analyses with varying precision (interval arithmetic and affine arithmetic) to check robustness of the simulations and to improve test coverage. Although these efforts are still far from a full verification, there is much scope for applying ideas from verification to complement and enhance testing/simulation efforts.

Another useful experience for us has been the development of a layered infrastructure, with a vertical silo for each verification application. A silo consists of a top layer of modeling/abstraction techniques for model extraction – these are interfaced with front-ends for standard programming or modeling languages. The middle layer comprises verification techniques applied on the model, e.g. abstract interpretation, model checking, symbolic execution, etc. The bottom-most layer consists of core back-end engines based on symbolic solvers, such as SAT/SMT solvers. The shared infrastructure enables re-use across different applications, and promotes the use of cooperative analyses. Another big advantage is that different layers can independently keep pace with advancements in analysis techniques or back-end solvers. In particular, we have benefitted greatly from the efforts of the SAT/SMT community. At the same time, we have found that it is not unusual to obtain an order of magnitude performance improvement in the back-end solvers (including our own) by exploiting application-specific encodings and heuristics within a vertical silo. We believe a continuing dialog between verification application providers and back-end solver developers is crucial for creating usable verification solutions.

Challenges and Lessons Learned: Industry Application & Technology Transfer

For adoption in an industry application, tool applicability/usability and justifying a business rationale are big challenges.

Our first challenge in this direction was to avoid the *expectation of a push-button tool*. Even with completely automated techniques (no interactive theorem-proving!), we believe that verification tools work best by involving a user in the loop (like a check-debug-refine cycle familiar to developers), as shown in Fig. 2. There are many reasons for this, including large code size necessitating verification of smaller units, missing code, library functions, etc. Any of these may lead to a reported bug being declared false by a user, with subsequent refinements required in the program, specifications, or the environment models. The verification tool provides great value by automating the difficult tasks of searching the state space and generating counterexample traces for bugs.

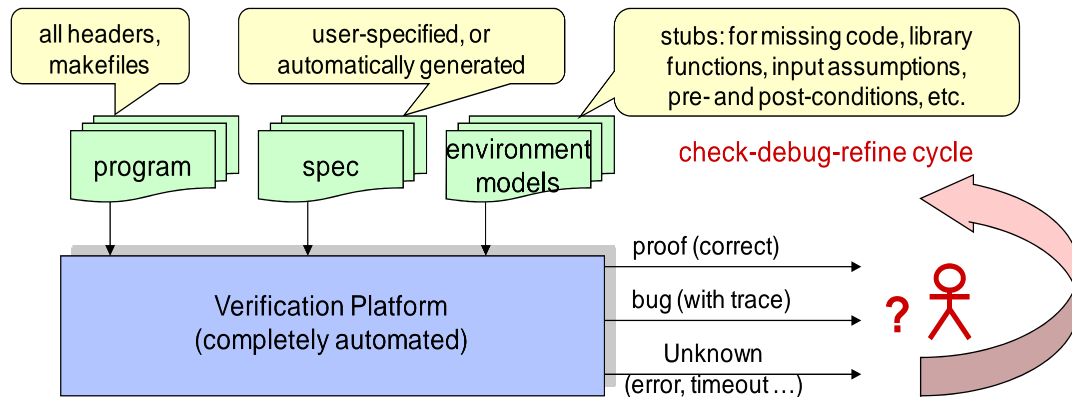

Figure 2: User in the Verification Loop

After it became clear that our users were interested mainly in bugs, we added many checks in stubs for standard libraries (to detect API-usage bugs) and developed some helper tools to reduce the false bug rate. One of these tools, called SpecTackle, performs light-weight static analysis to automatically infer *likely* pre-conditions and post-conditions for memory safety properties. These can be used to tune the environment conditions that affect the false bug rate. We also hope that these pre-/post-conditions will help users gain familiarity with contract-checking – although our tool can easily handle contract-checking, in practice it has been tough to get it off the ground. Another enhancement was to add a new abstract interpretation domain to our staged cooperative framework to perform a self-limitation analysis. This analysis uses patterns of likely false bugs, and declares related properties as don't care properties. Finally, our collaborators in NEC Japan developed a post-processing tool that analyzes the specific witnesses reported by the verification tool. It classifies them into a set of ranked bins and removes duplicate bug reports.

In terms of tool usability, we found that it was as important for the witness (error trace) presentation to be intuitive, as it was for the witness to be real. Even real bugs are likely to be declared "false" if they are complex, or difficult to understand. Therefore, a large amount of current development effort (by our collaborators in NEC Japan) is devoted to improving the witness presentation in our tools. It is also not clear how much of the "instrumented verification model" to expose to the end-user, or how much witness information to provide. To accommodate differences among user expectations, our tool provides various levels of witness abstraction, starting from a short natural language explanation highlighting two lines in the source code, to a witness path that is data-sliced, to a full path through the program execution with information on data values for all variables.

The front-end compilations required to get the verification tool started is another big challenge. As researchers, we can usually make changes in header files, etc., to make the code compile using CIL (thanks to Prof. Necula at UCB). However, in an industry setting, this can fail due to many reasons – legacy code requiring older compiler versions, different operating systems, optimizations, etc. A verification tool is expected to work, no matter what code is thrown at it. We believe it should be well-integrated with the build and compile environments for software development. Publicly available compiler platforms for extracting higher-level verification models would be extremely useful.

In terms of developing a business rationale and application context, there are many questions to consider: Who is the user of the tool (developer, QA team, verification experts)? When and how often is the tool used (nightly/weekly, only before code shipment, to find bugs, to track proofs)? The challenge is to integrate the verification tool into a working flow that provides total cost reduction (a need for highly-skilled verification users may not provide enough cost savings!). In our case, we engaged in technology transfer with collaborators in NEC Japan, who developed a product called VARVEL based on our F-Soft prototype. They provided an in-house verification service to other business units, and successfully applied VARVEL to find many bugs in large software projects (some with more than a million lines of code). More recently, VARVEL is being incorporated into a centralized automated facility for statically checking all software projects in a business unit.

Our experience in technology transfer is probably similar to other such efforts in the industry. Verification technology is not the easiest to develop and to use, and a product development team requires diverse skill sets. While verification performance continues to be an important issue, tool usability is equally (if not more) important. The business perspective and needs also evolved over time as our "customers" gained a better understanding of the strengths and limitations of the verification technologies. In this direction, it was very useful to hold internal technical workshops to inform them of the state-of-the-art and new trends. It is not surprising that to make continuous progress in technology transfer, consistent thought-leadership and frequent education of business users are essential.