Technology Transfer: Formal Analysis, Engineering, and Business Value

Ralf Huuck

NICTA and UNSW
School of Computer Science and Engineering
Sydney, Australia

Abstract

In this work we report on our experiences on developing and commercializing *Goanna*, a source code analyzer for detecting software bugs and security vulnerabilities in C/C++ code. Goanna is based on formal software analysis techniques such as model checking, static analysis and SMT solving. The commercial version of Goanna is currently deployed in a wide range of organizations around the world. Moreover, the underlying technology is licensed to an independent software vendor with tens of thousands of customers, making it possibly one of the largest deployments of automated formal methods technology. This paper explains some of the challenges as well as the positive results that we encountered in the technology transfer process. In particular, we provide some background on the design decisions and techniques to deal with large industrial code bases, we highlight engineering challenges and efforts that are typically outside of a more academic setting, and we address core aspects of the bigger picture for transferring formal techniques into commercial products, namely, the adoption of such technology and the value for purchasing organizations.

While we provide a particular focus on Goanna and our experience with that underlying technology, we believe that many of those aspects hold true for the wider field of formal analysis and verification technology and its adoption in industry.

Keywords: static analysis, model checking, SMT solving, industrial application, experience report

1. Introduction

Formal methods have come a long way from being a niche domain for mathematicians and logicians to an accepted practice, at least in academia, and to being a subject frequently taught in undergraduate courses. Moreover, starting out from a pen-and-paper approach, a range of supporting software tools have been developed over time, including specification tools for (semi-)formal languages such as UML, Z or various process algebrae, interactive theorem-provers for formal specification, proof-generation and verification, as well as a large number of algorithmic software tools for model checking, run-time verification, static analysis and SMT solving, to name a few [14].

Despite all the effort, however, there has been only limited penetration of formal analysis tools into industrial environments, mostly confined to the R&D laboratories of larger corporations, defense projects or selected avionics applications. The use of verification tools by the average software engineer is rare and typically stops at formal techniques built into the compiler or debugger.

In this work we present our experiences from developing the formal-methods-based source code analyzer *Goanna* [17, 16] and the technology transfer of moving the tool from a research prototype to a fully fledged commercial product that is used by large organizations around the globe. In particular, we report on bringing verification techniques such as model checking [28, 8], abstract interpretation [9] and SMT solving [12] to professional software engineers.

Email address: `ralf.huuck@nicta.com.au` (Ralf Huuck)

We explain why creating a successful software tool is far more than good technology and good bug detection, why engineering challenges need to be addressed realistically and why technology only plays a partial role in business decisions.

The goal of the work is to give some realistic insights into the opportunities and challenges of delivering software tools out of academia to some industrial setting and to explain what formal verification technology can deliver to end users, who are not experts in the field.

The remainder of this paper is structured as follows: In Section 2 we give a short introduction to our static analysis tool Goanna and some of the key design decisions that have been driving its development. Next, we give a high-level overview of Goanna's underlying technology in Section 3. This includes some of the tools and techniques used as well as their capabilities and limitations. In Section 4 we highlight a range of engineering challenges faced in creating an industrial strength tool. These are often rather different from the challenges in a more academic setting and can essentially prevent the adoption of any new technology. On top of this, and most importantly, any successful technology transfer requires a good value proposition to the end user. This means that purchasing a new software tool needs to solve a particular need and pay off. We explain some of the key underlying factors that drive these decisions in Section 5. Finally, in Section 6 we close our observations with some lessons we learned in the process.

2. The Tool: Goanna

Goanna is an automated software analysis tool for detecting software bugs, code anomalies and security vulnerabilities in C/C++ source code. Goanna has been developed at NICTA, first as an internal tool for research purposes and for support of internal mission-critical projects, and later as a commercial product that is available through the technology spin-off Red Lizard Software[1]. The tool is in continuous development and is used by many large corporations such as LG-Ericsson, Alstom or Siemens on a daily basis. Additionally, the underlying technology is OEM licensed to a major independent software vendor for programming of processors in embedded systems.

One of the original goals of the earlier project was to make verification technology applicable to large-scale industrial systems comprising millions of lines of source code. As such, there were a few general guidelines: First of all, any analysis tool has to be simple enough to use that it does not require much or any learning from users outside the formal methods domain. Secondly, the application of the tool has to match the typical workflow of the end-user. This means that if the end-user is accustomed to doing things in a particular order, those steps should remain largely the same. Moreover, run-time performance of any analysis should be similar to existing processes, which in terms of software development is often driven by compilation or build time. Finally, and most importantly, a new analysis tool should provide real value to an end-user. This means that it should deliver information or a degree of reliability that was previously not available, making the adoption of the tool worthwhile.

Goanna is designed to be run at compile-time and does not require any annotations, code modifications or user interaction. Moreover, the tool can directly be integrated into common development environments such as Eclipse, Visual Studio or build systems based on, e. g., Makefiles. To achieve acceptance in industry, all formal techniques are hidden behind a typical programmer's interface, all of C/C++ is accepted as input (even, e. g., Microsoft specific compiler extensions) and scalability to tens of millions of lines of code is ensured.

To achieve this, some of trade-offs had to be made:

Verification vs Bug Detection. While using a range of formal verification techniques, Goanna is not a verification tool as such, but rather a bug detection tool. This means that it does not guarantee the absence of errors, but the tool does its best to find certain classes of bugs. This means that while the techniques and algorithms developed are correct, the abstraction they are working on is not necessarily safe, i. e., the abstraction is not guaranteed to be a safe (over/under-)approximations at

[1] `http://redlizards.com`

all times. However, this also means that not all bugs are necessarily found, i.e., there might be *false negatives,* and some of the bugs found can be spurious, i.e., there can be *false positives.* In particular, unlike in verification we do not give a guarantee that there are no more bugs in a program after a successful run nor that every bug is a real one.

This approach is based on practical reasons. For instance, function pointers in C/C++ are notoriously hard to deal with. Any analysis that safely over-approximates function pointers' behavior quickly will warn that something is unsafe, i. e., can possibly go wrong. Warning that possibly anything could have happened after a manipulation of a function pointer is unrealistic and will create unacceptable noise for the user of such a tool. Allowing to miss certain instances of violations, i. e., giving up soundness is common [10, 24, 11]. Another option would be to keep soundness, but limit the accepted C/C++ constructs and usages [13, 3]. The latter, however, is often unrealistic in an industrial context.

Checks and Check Tuning. Goanna comes by default with a fixed set of pre-defined checks for errors such as buffer overruns, memory leaks, NULL-pointer dereferences, arithmetic errors, or C++ copy control mistakes as well as with support for certain safety-critical coding standards such as MISRA [27] or CERT [31], totaling over 200 individual checks. Finding the right balance between precision, speed and the number of supported checks is very much an engineering art supported by verification and abstraction techniques. We spent a lot of time and effort tuning the analysis to software bugs we have seen in the field and that represent a realistic class of issues. Moreover, we also tune the abstraction to avoid both false positives as well as false negatives as much as possible. However, this is a complex topic and while we present some technical approaches to false positive elimination in Section 3, a much deeper discussion on false positives and how to address them is presented in earlier work [18, 23].

An emphasis on the practical application of source code analysis by sacrificing soundness and instead focusing on the best possible results is also highlighted by Engler et al. [15]. At the same time this underlying philosophy is at the core of a range of commercial products that originated from various universities and research labs [10, 24, 22] .

3. Technology

This section explains some of the underlying technologies of the Goanna tool. The specific details and formal definitions can be found in earlier publications on our core model checking approach [17], the details on inter-procedural analysis [4], and the formal treatment of counterexample-guided trace refinement using SMT solvers [23].

3.1. Core Idea

Central to our approach is to treat static analysis as a model checking problem. This idea was inspired by Schmidt [30, 29], where the connection between μ-calculus and static program analysis is explored. Our basic ideas are illustrated in Figure 1: On the left there is a simple C program and on the right there is its corresponding initial coarse abstraction. The abstraction consists of the program's structure as extracted in a control flow graph (CFG) and atomic proposition, which are essentially labels of points of interest. The CFG and the labels are computed automatically based on predefined check patterns.

In the example in Figure 1, the points of interest are the control locations where memory is allocated, where it is used and where it is freed. Those locations can be determined by pattern matching on the parsed code. Once the abstraction has been automatically built it is possible to state typical static analysis queries as temporal logic formulas. For instance, consider the following Computation Tree Logic (CTL) formula:

$$\mathbf{AG}(malloc_{\mathtt{p}} \Rightarrow \mathbf{AG}(free_{\mathtt{p}} \Rightarrow \mathbf{AG}\neg used_{\mathtt{p}}))$$

This formula means: for every allocated memory (`malloc`) we ensure that after a path to a matching freeing of memory (`used`) we do not encounter the use of a pointer to that memory (`free`). The CTL operator **A** refers to all paths and **G** to globally, meaning all states. Variances of the same check can easily

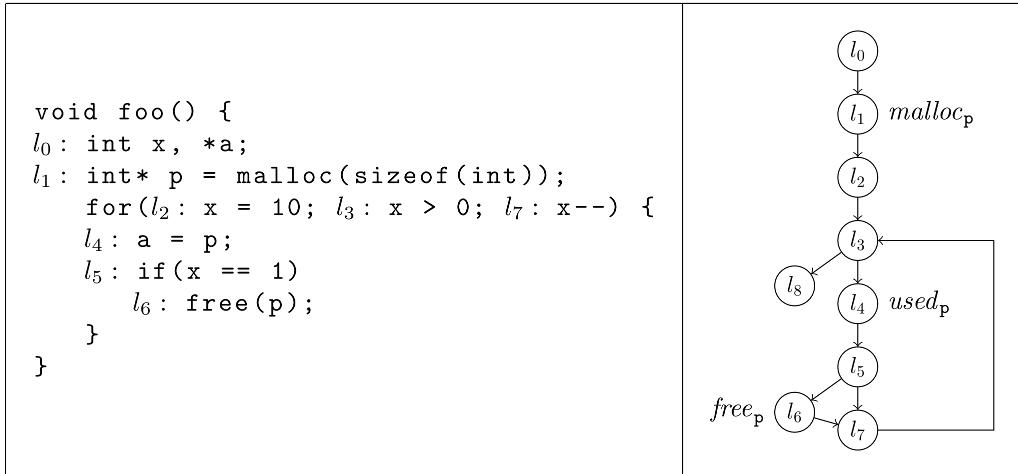

Figure 1: Example of an annotated CFG for a function `foo`. The locations are also annotated in the listing.

be achieved by switching to existential quantifiers, stating for instance that usage should not happen on at least one path, otherwise we can deduce that there is a violation on every path.

All the elements above can be predefined as templates. The labeling to generate the atomic propositions is predefined as search queries on the abstract syntax tree (AST) of the parsed code in the form of (tree) pattern matches. The individual checks are CTL templates where the results from the atomic proposition generation are filled in, and the transition system is defined by the structure of the CFG.

The Goanna analyzer ships with a large library of predefined checks containing around 200 different CTL properties to identify various classes of potential C/C++ software bugs, including NULL-pointer dereferences, copy constructor problems, concurrency locking issues, memory leaks and security vulnerabilities. The full list can be found online[2].

3.2. Advantages

Using CTL model checking for static analysis might be unusual to a certain degree as the worst case complexity is higher than in traditional static analysis using flow equation solving. In practice, however, there a number of factors that indicate that this approach is favorable:

- The initial coarse abstraction consists of the control flow graph and atomic propositions representing points of interest. This means that the state space is typically very small as there are no concurrent components nor detailed data structures that might lead to state explosion. A single CTL check on this abstraction typically only takes a tiny fraction of a second in analysis time.
- Unlike traditional static analysis little effort is required to write a new check. It is sufficient to define the pattern matching rules for points of interest and the CTL property. There is no need to define transfer functions and join operations, or to manage a fix-point iteration algorithm.
- The current approach is amenable to additional software model checking techniques such as abstraction/refinement. Since we are already in the model checking world we can use the current abstraction as a starting point to add more semantic information whenever needed. This is explained in Section 3.3.
- The analysis is by default flow-sensitive, i. e., the CTL model checking algorithm by its very nature considers every possible path without merging information at joins as in classical static analysis. This is done efficiently without enumerating each path individually.

[2] `http://redlizards.com/resources/user-manuals/`

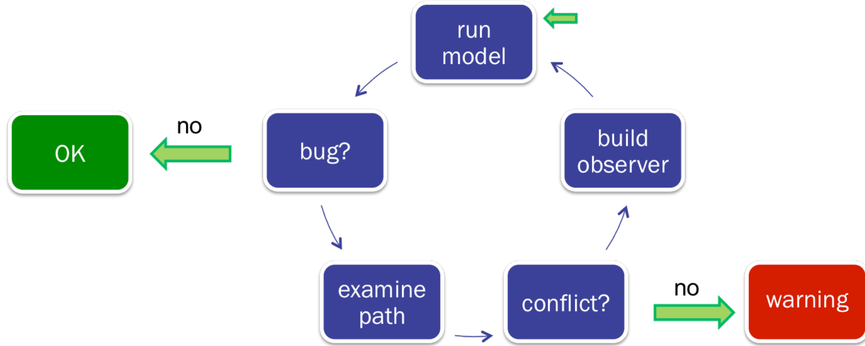

Figure 2: SMT-based trace refinement loop

- Several CTL model checking queries can make use of each others' results. By looking up sub-formulas it is possible to reuse information across different CTL formulas on the same structure. More details on this are given in Section 4.

The idea of using an automata-based approach to static analysis is also followed in the Uno tool [21] and its later development in Orion [11].

3.3. Path-based Abstraction/Refinement using SMT

One of the drawbacks of the abstraction presented above is that it is often too coarse, leading to false positives. This can be seen for instance in the example in Figure 1: The actual program semantics only frees the memory in the last loop iteration. This means that the memory is no longer used after the free-operation. This fact is lost in the abstraction, leading to a false positive using the naive approach outlined earlier.

To remedy this situation we use a language refinement approach, whenever we encounter a *potential* bug, i. e., obtain a counterexample from the CTL model checking phase. The details of the refinement approach are published earlier [23]. Here we give the core ideas:

1. Whenever we get a counterexample we examine if this counterexample is real or spurious by subjecting it to a detailed analysis in an SMT solver. In the SMT solver we closely model the C semantics and investigate if the constraints on the counterexample path are satisfiable.
2. If the path cannot be satisfied, we know the counterexample is spurious. We compute the unsatisfiable core of the counterexample, which relates to the actual *cause* of the unsatisfiablility, and create an excluding observer automaton that is ruling out the set of paths with the same cause.
3. We run the original model with the observer automata in parallel and repeat the process until we either do not find a new counterexample or we cannot refute it.

The idea of this abstraction/refinement approach is depicted in Figure 2. The advantage is that the refinement happens on-demand, i. e., whenever a potential bug is found. Moreover, the refinement only adds observers ruling out infeasible traces, keeping the original model the same and the overall state space relatively small. Most importantly, the refinement approach is something that is not easily possible in a classical static analysis setting and provides a real advantage in delivering better results. Recent approaches using interpolants for verification go into the same direction and are a field of future research [19].

Apart from these core techniques involving model checking and SMT solving we use a range of other methods that we briefly touch on in the next section.

3.4. Other Techniques

While the core techniques are good to find deep bugs in an efficient manner, there are limitations when it comes to data range checking for buffer overflows and alike, for pointer handling and dealing with millions of lines of C/C++ code at a time. This is addressed as follows:

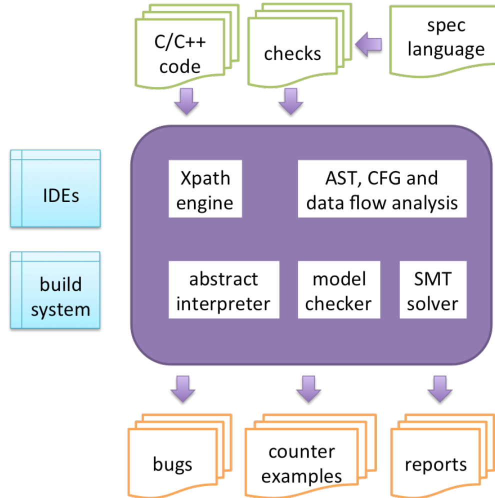

Figure 3: High-level Goanna Architecture

Abstract interpretation. We use a classical abstract interpretation framework to approximate data ranges for integers and pointers. The abstractions are based on an interval semantics and are computed in a separate phase before the results are integrated into subsequent analyses. While there are more precise domains for abstract interpretation, including octagons and (convex) polyhedra, in practice it appears that intervals are quite sufficient, especially in conjunction with our abstraction/refinement approach. Most importantly though, interval abstractions are easy and cheap to compute.

Pointer Aliasing. The tool contains a pointer alias analysis to improve the analysis results both in terms of minimizing false negatives as well as false positives. To achieve this, we are building heap graphs that for each location in the program approximate the actual pointer structure, i. e., which pointers are currently aliases of each other and point to the same memory location. This enables us to detect dangling pointers as well as for instance double-free operations, where `free` is called twice on different pointer names, but addressing the same memory [2]. The analysis is done separately and the pointer aliasing structures are kept in memory for other analysis steps.

Modular Analysis. In order to scale to millions of lines of code, Goanna uses a function-modular approach to handle inter-procedural information. This means that for each function we compute an input/output summary and only propagate that summary information along the call-graph for future computations. For instance, we track which pointers related to parameters dereferencing a pointer, or which variables require an incoming pointer to be non-NULL. Typically, the summary is in the form of a classical lattice, i. e., a pointer is in {`null, not_null, maybe_null, unknown`} with the usual ordering on it.

Incremental Analysis. An incremental analysis is used to avoid re-analysis of a whole project if only some code parts changed between two runs. Instead, Goanna analyses the changed code and related dependencies only. Additionally, there is support for parallel analysis and running Goanna on distributed servers, if such a setup exists.

Figure 3 depicts the high-level architecture of our software analysis tool. At the core are the different analysis engines for abstract interpretation, model checking, SMT solving and pattern matching. As input the tool takes C/C++ projects and a set of specifications that are written in our own domain-specific language and predefined for standard users. Goanna can be run on the command line, integrated with IDEs and can be embedded in the build system. It outputs warning messages classified by severity, displays error traces based on counterexamples, and can create summary reports.

Overall, the integration of various formal verification techniques into Goanna makes the tool rather different from other static analysis tools. At the same time it is also different from traditional software model checking tools by sacrificing some of the latter's semantic depth and focusing on more generic bug detecting capabilities. We believe that Goanna provides some realistic middle ground to address deep software issues in a practical manner.

4. Engineering

In the previous section we explained some of the underlying formal verification techniques used and the overall approach to static analysis as implemented in Goanna. There are, however, a range of challenges that go beyond the actual core technology to create an industrial strength product.

Our earliest implementation was a typical academic prototype that was in fact more a proof of concept than a complete tool. For that we extracted some intermediate format out of the gcc compiler, we parsed that format, built the necessary data structures and transformed them into the input language of the NuSMV model checker [7]. Finally, we used to call NuSMV and parse the output back. There has been a whole range of problems with this approach that became apparent with early industrial adopters:

Parsing. Key to analyzing C/C++ is the ability to parse the code, which is already a substantial barrier. Our original approach using gcc had several issues: gcc's internal format was neither well documented, nor consistent or even stable over various version numbers. A better choice would have been LLVM [25] with a cleaner interface, but LLVM was still in its infancy at the time. But more importantly, industrial projects often do not use gcc (and do not compile with gcc), but have their own compiler variants such as the Microsoft compiler, the ARM compiler, the IAR compiler and so on. Each compiler has its own interpretation of C/C++ and its own non-standard extensions. For those reasons we decided to go with the commercial EDG compiler[3] that supports various dialects to a high degree. Still, being able to parse all language aspects including compiler extensions and C++ templates does not mean that it is straightforward to handle those constructs correctly. It takes significant effort to deal with them in the underlying implementation.

Build Integration. Being able to parse code is one challenge, being able to integrate into existing build infrastructures is a different issue. Large commercial build systems have two major obstacles: Hardly anyone is allowed to touch them and hardly anyone still fully understands their inner workings. Often build systems have been growing over time. They might have started out as standard makefiles, but then got patched by calling Perl scripts or other shell scripts, include special feature builds for various target environments, call different compilers and they might not even build completely if legacy code and third party libraries are involved.

As a tool vendor it is almost impossible to require a change to the build system, and managers are understandably highly sensitive about any process changes. As a solution we developed a non-invasive build wrapping that channels calls to the compiler through our build recording mechanism, which can be used to run with the actual analyzer at a later time. This also includes some auto-configuration for different compilers and target environments on the fly to detect the right settings, right include files and so on. This in itself is a major piece of work and support.

IDE Integration. Similar to build integration we also provide a Goanna variant to integrate with common IDEs to make the uptake of the tool as simple and straightforward as possible, and to enable a seamless adoption by software developers, who are not experts in formal techniques and are also not experts in build system integration.

As above, the integration effort is non-trivial. Starting out with support for Eclipse and Microsoft Visual Studio we branched out into various embedded compiler IDEs. Each have their own slightly

[3] http://www.edg.com

different concept that require a lot of time of engineering, re-engineering, debugging and testing. Often this also requires to accommodate minor changes between different versions, where certain APIs are no longer available or have changed.

The above challenges present a rather high entry hurdle to the industrial application of software analysis. While not all of those items need to be addressed simultaneously, they certainly impact uptake and support requirements. Often these issues prevent a successful application of the core analysis.

The next set of issues are related around performance and scalability to large code bases. While academic prototypes are often good enough if they can demonstrate the underlying techniques on some example code bases, industrial projects quickly require to deal efficiently with many millions of lines of code. For instance, in the telecom industry, Android is now a popular platform that gets included in the build process, pulling into every nightly build around 6 million lines of pure C/C++ code, which translates to 43 million lines of code after preprocessing. This number does not even include the applications on top of it.

Commercial requirements often dictate that the analysis finishes "in time", which means at least over night for such large code bases. In general, this will require analysis time that is similar to compile time or a close multiple of it. Below we report on our experience of scaling our basic analysis technologies to those requirements.

Pattern Matching. As mentioned earlier, our CTL model checking approach uses pattern matching to detect points of interest that get translated into atomic proposition for the CTL formulas. This pattern matching is done on the parsed program's abstract syntax tree (AST). For finding patterns in the AST we use an XPath-based query language [20], which was originally designed to efficiently describe queries in XML documents.

However, since we have a wide range of different CTL properties and each CTL property might depend on atomic proposition described as complex XPath queries, the number of queries might be fairly high. While each individual query might only take tenths of a second or milliseconds the sheer number of queries can add up. For instance, for the Android code base Goanna evaluates close to 500 million XPath queries. As are result, the queries have a non-trivial performance impact taking up to 20% of the overall runtime.

To deal with this, we implemented a range of caching stages to avoid re-computation of sub-expressions and hardcoded some of the most critical expressions. Still, performance optimization is ongoing and often it is not immediately clear if some new optimization brings a performance improvement or might hit the garbage collector in our implementation language OCaml and decreases performance.

Model Checking. The models that are generated in Goanna are very atypical for the model checking problems encountered in academia. In our case the state space is mostly small with a few hundred or a few thousand control flow locations, but there are a large number of CTL formulas to check on the same structure. Large means that it is not too unusual to check for several thousand CTL formulas on a single file. The reason is that in the worst case every class of checks applies to each variable. For instance, checking for uninitialized pointers there will be one CTL formula for each pointer variable. Many checks and many variables lead to many short model checking queries, but those add up to overall excessive runtimes.

Our initial approach using NuSMV did not scale well across many CTL formulas as it appears that each CTL formula gets checked from scratch. Moreover, NuSMV symbolically encodes models as binary decision diagrams (BDDs) [5]. This encoding computation in itself appears to take some significant time. In the end, we implemented our own explicit-state model checker that was optimized for re-using sub-formulas already evaluated by a different CTL formula on the same model and also lazily evaluating operations such as negations when needed. As a result we get around 60% of cache hits and have seen cases where we were 500x faster than NuSMV. But most importantly, and somewhat surprisingly, we were able to reduce run-time especially for larger models where NuSMV originally performed relatively poorly.

Positive Surprises. On the upside, using our own model checker and our abstraction refinement approach works very well in terms of performance. In the end, what we used to believe is the core bottleneck only consumes around 10-20% of the overall runtime, often being faster than the actual parsing and preprocessing, which as mentioned earlier is the baseline for measuring the overhead. This makes us particularly hopeful that we can add more sophisticated techniques further down the road without too much of a runtime impact.

Also, the notion of model checking and the value it might deliver through a rigorous analysis is something that is highly welcome in many organizations. Depending on their level of expertise these days there is some understanding of these techniques within companies and there is a desire to apply them, if the overall overhead both in runtime and required expertise is manageable.

Having mentioned key integration and analysis challenges, there are a range of engineering tasks that are typically not addressed in a more academic environment. This includes: extensive regression testing, nightly automated testing (we analyze tens of millions of lines of code every night), build and release management with different development branches and quality gates, support for different hardware platforms and OS versions, support for a range of embedded compiler dialects and ensuring a management-friendly reporting mechanism. Many of these tasks are not directly related to the core algorithms, but to common engineering tasks and best practices.

Overall the development effort and investment in getting a software tool to almost fool-proof industrial level is immense. It is fair to say that the amount of time and effort from a prototype, to an alpha version to the first commercial release is exponential. For instance, we spend around five times more resources on productization than the inital whole research and prototyping phase.

5. Business Value

Sections 3 and 4 explained our underlying technology and some of the engineering challenges we faced that are outside a typical academic setting. In this section we go one step further and report on the experiences from a business perspective as a vendor of code analysis tools.

One lesson we quickly learned embarking on a commercial journey is that the technology and even the product only plays a limited role in the overall success. As already highlighted in Section 4 there can be engineering hurdles that can easily prevent the use of any underlying technology. Moreover, there are basic factors such as market positioning, competition, licensing model, sales channels and support capabilities that all contribute to the business outcome. In the following we address some of those aspects that are likely of specific interest from a technology transfer point of view and that might be worthwhile considering for similar projects.

Software Bug Reporting. The value of a product and in turn technology is often closely related to the value it provides to the business deploying that product. The value of detecting a software bug is hard to quantify. There are often mundane bugs that do not exhibit fatal crashes, but for instance manifest as a higher memory consumption or performance degradation that might not be noticeable to a service or product. On the other hand there can be so called showstoppers that are detrimental to business. Often it is not easy to exactly pinpoint the severity of an issue.

However, knowing the existence of various potential issues in a product is of value to a software developing organization. It assists risk management by reducing the number of previously unknown quality and security issues in a system. While not every bug might be fixed (and often is not), the overall numbers and metrics are of interest especially to team leaders and management.

For this reason bug reporting is often deemed important. This includes trending information as well as facilities to drill down into certain areas of code as well as classes of bugs. From a business perspective this high level overview and quality control is often critical for making decisions around where to spend time and effort, for defining release gating criteria or for meeting compliance according to industry standards.

Value of Reduced Development Time. Another core aspect of using automated analysis techniques is to cut down on development and test cycles. In larger organizations there are separate teams for development and testing. Both can profit from automated tools:

Value for Test Engineers. Unlike traditional software testing, the use of static analysis tools requires little manual work as there are no test cases to set up or interpret. Moreover, static analysis is complementary to traditional testing, this means that bugs such as crashes or memory leaks are a different focus than incorrect functional behavior. Therefore, these types of tools can provide a good incentive to the test engineer and the quality assurance group. The value is mostly quantified in time saved and effort reduced, which depends on the specific organization.

Value for Developers. As an alternative or as an addition, providing automated software analysis tool to the developers themselves opens up new capabilities for detecting bugs during the development rather than in a later test cycle. This is valuable, since even if a bug was detected by the test team a day later and reported back to the developer, it still creates a significant overhead and distraction from the developer's ongoing work. Bringing this forward not only cuts down on costs, but to a certain extent can help in job satisfaction or at least reduces interruptions.

Value of Precision. In all of the above cases the value is closely correlated to the precision of a tool, meaning the number of false positives and false negatives. While a certain number of false positives is acceptable, because it is still relatively quick to investigate a few warnings that might be spurious, it is only acceptable if the majority of issues are real. This means that the better the underlying technology is in being precise or automatically removing false positives, the higher is the acceptance of a tool and its value. Moreover, in our experience it is at the same time important to provide the end user with some mechanism to suppress issues in future analysis runs once they are classified as false or as not serious enough to consider fixing.

Cost of Integration. A new product not only brings value, but naturally comes with costs. License fees are often only a small fraction of the overall costs to an organization which includes: Costs of procurement, costs of training or induction required to staff members, costs of integration with existing tools and processes or, most significantly, cost of changes in workflow and infrastructures. Some additional questions around costs that might arise are about who will be responsible for ongoing maintenance and configuration: the build manager, the quality assurance manager or each individual developer?

In our experience it appears to be important to provide open interfaces to integrate with existing tools and processes. Moreover, it is highly important that a software analysis tool minimizes the need to modify core critical infrastructure such as the software development build system.

Lastly, we would like to highlight some of the complexities not necessarily in the application of the tools, but more on the soft side, including communication and selling processes. Even with a very specific software analysis tool there can be various levels of company hierarchy involved in making a buying decision. This includes the developer, but also the test engineers, the quality assurance or security team manager, the product manager or even top-level management.

While in smaller organizations these buying decisions might be made by an individual developer, in our experience this is unusual. Typically, decisions are made by higher levels in the management hierarchy who take advise from other technical staff in the organization. However, this also means that a product is viewed as much more than just a technology, it is the solution to a specific need. The problem might be poor product quality, missed release deadlines, low confidence in a code base or even lack of skilled staff. Software bugs are a manifestation of those problems. Software tools provide assistance to quality assurance, quality tracking, high-quality release management and alike, where software bugs are only one component in a larger picture that gets tracked in dashboards and reports. From that perspective, it is important to understand that difference and to provide the necessary tools that fit into the enterprise workflow, that deliver reports, and that provide some level of management view into the results.

6. Lessons

Our experiences are to a degree similar to those who have been involved in moving academic tools to commercial products. Prime examples are Coverity [10], Astrée [3] and its precursor Polyspace, as well as Fortify [6]. The Coverity team has highlighted some of their challenges regarding false positives and build system integration [1]. These are similar to ours. Moreover, Gary McGraw described his technology transfer experience from a venture capital injection point of view [26] emphasizing the chasm between sophisticated research technology and the realities of business needs.

Interestingly, software analysis plays only a limited role in the overall software development lifecycle and general product development process. Bugs are of course a concern if they lead to critical failures and therefore a loss in business. However, a bug that never or very rarely manifests might often not be of primary concern. Nonetheless, being able to track the overall software quality and observe trending is generally considered to be important and worthwhile. This leads to the interesting constellation where the people making purchasing decisions are typically different from the software developers and motivated by quite different reasons. To produce a solution that makes all sides happy is not straightforward.

In our experience there are some key reasons to use formal methods based tools: There is a notable benefit when designing such tools by the intrinsic rigor that comes with the field, i. e., limitations and corner cases are typically considered and the overall internal architecture can be better designed. Moreover, the analysis can often be deeper and produce better results than comparable technology based purely on heuristics. And finally, also from the customer side it is well understood that verification-based tools are built on solid frameworks, which in turn can create a positive impression.

While creating a product required a large investment both in engineering and business development, we were quite pleased to notice, and to a certain degree surprised, how well our formal analysis techniques performed on large industrial applications. This encourages us to believe that continuous development in core analysis technologies such as SMT solvers will play a much larger role in industrial software assurance.

Of course, not many universities and research organizations have the desire, need or capacity to lift this often costly technology transfer. But even so, we see some level of acceptance of more academic style tools into commercial environments. In our experience there is a willingness to accept tools with rough edges if the licensing conditions are unproblematic, if tools are well documented with a certain user base, and the tool can deliver a degree of reliability that is *good enough* for most cases. Combined with delivering something unique a software has a good potential to get at least some uptake in industrial applications.

Looking forward, we see growing challenges that are in our opinion not yet well addressed in the formal analysis community. These are mainly related to the huge source code bases in industry that continue to grow rapidly. For instance, the requirement to handle 30+ millions of lines of code will soon be a reality. Moreover, these large systems are under constant development with short product development cycles. Being able to handle massive data in an efficient manner that finds just the right software issues is one of the core open issues that might require a more inter-disciplinary approach combining for instance data analytics, machine learning and other big data approaches with formal methods.

Acknowledgments. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program. The author also likes to thank the anonymous reviewers for their comments on the earlier versions of this worked that helped to shape this article.

References

[1] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D., 2010. A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM 53, 66–75.

[2] Biallas, S., Olesen, M., Cassez, F., Huuck, R., 2013. PtrTracker: Pragmatic pointer analysis, in: IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 69–73.

[3] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Min, A., Monniaux, D., Rival, X., 2002. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, in: The Essence of Computation. Springer-Verlag. Volume 2566 of *Lecture Notes in Computer Science*, pp. 85–108.

[4] Brauer, J., Huuck, R., Schlich, B., 2009. Interprocedural pointer analysis in Goanna. Electronic Notes in Theoretical Computer Science 254, 65 – 83. Proceedings of the 4th International Workshop on Systems Software Verification (SSV 2009).
[5] Bryant, R., 1986. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers C-35, 677–691.
[6] Chess, B., West, J., 2007. Secure programming with static analysis. Addison-Wesley Professional.
[7] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A., 2002. NuSMV 2: An opensource tool for symbolic model checking, in: Computer Aided Verification, Springer-Verlag, pp. 359–364.
[8] Clarke, E., Emerson, E., 1982. Design and synthesis of synchronization skeletons using branching time temporal logic, in: Logics of Programs. Springer-Verlag. Volume 131 of *Lecture Notes in Computer Science*, pp. 52–71.
[9] Cousot, P., Cousot, R., 1979. Systematic design of program analysis frameworks, in: Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, New York, NY, San Antonio, Texas. pp. 269–282.
[10] Coverity, 2006. Prevent for C and C++. http://www.coverity.com.
[11] Dams, D., Namjoshi, K., 2004. Orion: High-Precision Methods for Static Error Analysis of C and C++ Programs. Bell Labs Tech. Mem. ITD-04-45263Z. Lucent Technologies.
[12] De Moura, L., Bjørner, N., 2011. Satisfiability modulo theories: introduction and applications. Communications of the ACM 54.
[13] Deutsch, A., 1995. Semantic models and abstract interpretation techniques for inductive data structures and pointers, in: Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, ACM, New York, NY, USA. pp. 226–229.
[14] D'Silva, V., Kroening, D., Weissenbacher, G., 2008. A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 27, 1165–1178.
[15] Engler, D., Chelf, B., Chou, A., Hallem, S., 2000. Checking system rules using system-specific, programmer-written compiler extensions, in: Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4, USENIX Association, Berkeley, CA, USA. pp. 1–1.
[16] Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M., Rauch, F., 2007a. Goanna: A static model checker, in: Formal Methods: Applications and Technology. Springer-Verlag. Volume 4346 of *Lecture Notes in Computer Science*, pp. 297–300.
[17] Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M., Rauch, F., 2007b. Model checking software at compile time, in: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, IEEE Computer Society, Washington, DC, USA. pp. 45–56.
[18] Fehnker, A., Huuck, R., Seefried, S., Tapp, M., 2010. Fade to grey: Tuning static program analysis. Electronic Notes in Theoretical Computer Science. 266, 17–32.
[19] Heizmann, M., Hoenicke, J., Podelski, A., 2013. Software model checking for people who love automata, in: Sharygina, N., Veith, H. (Eds.), Computer Aided Verification. Springer-Verlag. Volume 8044 of *Lecture Notes in Computer Science*, pp. 36–52.
[20] Hidders, J., Paredaens, J., 2009. Xpath/xquery, in: Liu, L., Özsu, M.T. (Eds.), Encyclopedia of Database Systems. Springer US, pp. 3659–3665.
[21] Holzmann, G.J., 2002. Uno: Static source code checking for user-defined properties, in: In 6th World Conference on Integrated Design and Process Technology, IDPT 02.
[22] Jetley, R.P., Jones, P.L., Anderson, P., 2008. Static analysis of medical device software using Codesonar, in: Proceedings of the 2008 workshop on Static analysis, ACM, New York, NY, USA. pp. 22–29.
[23] Junker, M., Huuck, R., Fehnker, A., Knapp, A., 2012. SMT-based false positive elimination in static program analysis., in: Aoki, T., Taguchi, K. (Eds.), ICFEM. Springer-Verlag. volume 7635 of *Lecture Notes in Computer Science*, pp. 316–331.
[24] Klocwork, 2009. Klocwork Insight. http://www.klocwork.com/.
[25] Lattner, C., Adve, V., 2004. Llvm: A compilation framework for lifelong program analysis & transformation, in: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, IEEE Computer Society, Washington, DC, USA.
[26] McGraw, G., 2011. Technology transfer: A software security marketplace case study. IEEE Software 28, 9–11.
[27] MIRA Ltd, 2004. MISRA-C:2004 Guidelines for the use of the C language in Critical Systems. URL: `www.misra.org.uk`.
[28] Queille, J., Sifakis, J., 1982. Specification and verification of concurrent systems in cesar, in: International Symposium on Programming. Springer-Verlag. Volume 137 of *Lecture Notes in Computer Science*, pp. 337–351.
[29] Schmidt, D.A., 1998. Data flow analysis is model checking of abstract interpretations, in: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA. pp. 38–48.
[30] Schmidt, D.A., Steffen, B., 1998. Program analysis as model checking of abstract interpretations, in: Proceedings of Static Analysis: 5th International Symposium (SAS) 1998, Springer-Verlag, pp. 351–380.
[31] Seacord, R.C., 2008. The CERT C Secure Coding Standard. 1st ed., Addison-Wesley Professional.