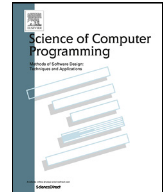
Science of Computer Programming 103 (2015) 3–12

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico



Technology transfer: Formal analysis, engineering, and business value

Ralf Huuck

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

ARTICLE INFO

Article history:
Received 17 June 2013
Received in revised form 13 November 2014
Accepted 13 November 2014
Available online 24 November 2014

Keywords:
Static analysis
Model checking
SMT solving
Industrial application
Experience report

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.

© 2014 Elsevier B.V. All rights reserved.

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* [16, 17] and the technology transfer of moving the tool from a research prototype to a fully fledged commercial product that

E-mail address: ralf.huuck@nicta.com.au.

http://dx.doi.org/10.1016/j.scico.2014.11.003
0167-6423/© 2014 Elsevier B.V. All rights reserved.

is used by large organizations around the globe. In particular, we report on bringing verification techniques such as model checking [8,28], abstract interpretation [9] and SMT solving [12] to professional software engineers.

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 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 [10,11,24]. Another option would be to keep soundness, but limit the accepted C/C++ constructs and usages [3,13]. 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

[1] http://redlizards.com.

Download English Version:

https://daneshyari.com/en/article/6875329

Download Persian Version:

https://daneshyari.com/article/6875329

Daneshyari.com