Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 217 (2008) 5–21

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Comparative Study of Industrial Static Analysis Tools

Pär Emanuelsson[1],[2]

Ericsson AB
Datalinjen 4
SE-583 30 Linköping, Sweden

Ulf Nilsson[1],[3]

Dept. of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Abstract

Tools based on static analysis can be used to find defects in programs. Tools that do shallow analyses based on pattern matching have existed since the 1980's and although they can analyze large programs they have the drawback of producing a massive amount of warnings that have to be manually analyzed to see if they are real defects or not. Recent technology advances has brought forward tools that do deeper analyses that discover more defects and produce a limited amount of false warnings. These tools can still handle large industrial applications with millions lines of code. This article surveys the underlying supporting technology of three state-of-the-art static analysis tools. The survey relies on information in research articles and manuals, and includes the types of defects checked for (such as memory management, arithmetics, security vulnerabilities), soundness, value and aliasing analyses, incrementality and IDE integration. This survey is complemented by practical experiences from evaluations at the Ericsson telecom company.

Keywords: Static analysis, dataflow analysis, defects, security vulnerabilities.

1 Introduction

Almost all software contain *defects*. Some defects are found easily while others are never found, typically because they emerge seldom or not at all. Some defects that emerge relatively often even go unnoticed simply because they are not perceived as errors or are not sufficiently severe. Software defects may give rise to several types of *errors*, ranging from logical/functional ones (the program sometimes computes

[1] Thanks to Dejan Baca, Per Flodin, Fredrik Hansson, Per Karlsson, Leif Linderstam, and Johan Ringström from Ericsson for providing tool evaluation information.

[2] Email: par.emanuelsson@ericsson.com

[3] Email: ulfni@ida.liu.se

doi:10.1016/j.entcs.2008.06.039

incorrect values) to runtime errors (the program typically crashes), or resource leaks (performance of the program degrades possibly until the program freezes or crashes). Programs may also contain subtle security vulnerabilities that can be exploited by malicious attackers to gain control over computers.

Fixing defects that suddenly emerge can be extremely costly in particular if found at the end of the development cycle, or even worse, after deployment. Many simple defects in programs can be found by modern compilers, but the predominating method for finding defects is *testing*. Testing has the potential of finding most types of defects, however, testing is costly and no amount of testing will find all defects. Testing is also problematic because it can be applied only to executable code, i.e. rather late in the development process. Alternatives to testing, such as *dataflow analysis* and *formal verification*, have been known since the 1970s but have not gained widespread acceptance outside academia—that is, until recently; lately several commercial tools for detecting runtime error conditions at compile time have emerged. The tools build on *static analysis* [27] and can be used to find runtime errors as well as resource leaks and even some security vulnerabilities statically, i.e. without executing the code. This paper is a survey and comparison of three market leading static analysis tools in 2006/07: PolySpace Verifier, Coverity Prevent and Klocwork K7. The list is by no means exhaustive, and the list of competitors is steadily increasing, but the three tools represent state-of-the-art in the field at the moment.

The main objective of this study is (1) to identify significant static analysis functionality provided by the tools, but not addressed in a normal compiler, and (2) to survey the underlying supporting technology. The goal is not to provide a ranking of the tools; nor is it to provide a comprehensive survey of all functionality provided by the tools. Providing such a ranking is problematic for at least two reasons: Static analysis is generally only part of the functionality provided by the tool; for instance, Klocwork K7 supports both refactoring and software metrics which are not supported by the two other tools. Even if restricting attention only to static analysis functionality the tools provide largely non-overlapping functionality. Secondly, even when the tools seemingly provide the same functionality (e.g. detection of dereferencing of null pointers) the underlying technology is often not comparable; each tool typically finds defects which are not found by any of the other tools.

Studying the internals of commercial and proprietary tools is not without problems; in particular, it is virually impossible to get full information about technical solutions. However, *some* technical information is publicly available in manuals and white papers; some of the tools also originate from academic tools which have been extensively described in research journals and conference proceedings. While technical solutions may have changed somewhat since then, we believe that such information is still largely valid. We have also consulted representatives from all three providers with the purpose to validate our descriptions of the tools. Still it must be pointed out that the descriptions of suggested technical solutions is subject to a certain amount of guessing in some respects.

This technological survey is then complemented by a summary and some examples of tool evaluations at Ericsson.

The rest of the report is organized as follows: In Section 2 we define what we mean by the term static analysis and survey some elementary concepts and preconditions; in particular, the trade off between precision and analysis time. In Section 3 we summarize the basic functionality provided by the three tools—Coverity Prevent, Klocwork K7 and PolySpace Verifier/Desktop—focusing in particular on the support for the C and C++ programming languages. The section also surveys several industrial evaluations of the tools over time at Ericsson, in particular involving the products from Coverity and Klocwork. Section 4 contains conclusions.

2 Static analysis

Languages such as C and, to a lesser extent, C++ are designed primarily with efficiency and portability in mind[4], and therefore provide little support to avoid or to deal with runtime errors. For instance, there is no checking in C that read or write access to an array is within bounds, that dereferencing of a pointer variable is possible (that the variable is not null) or that type casting is well-defined. Such checks must therefore be enforced by the programmer. Alternatively we must make sure that the checks are not needed, i.e. guarantee that the error conditions will never occur in practice.

By the term *static analysis* we mean *automatic* methods to reason about runtime properties of program code without actually executing it. Properties that we consider include those which lead to premature termination or ill-defined results of the program, but precludes for instance purely syntactic properties such as syntax errors or simple type errors.[5] Nor does static analysis address errors involving the functional correctness of the software. Hence, static analysis can be used to check that the program execution is not prematurely aborted due to unexpected runtime events, but it does not guarantee that the program computes the correct result. While static analysis *can* be used to check for e.g. deadlock, timeliness or non-termination there are other, more specialized, techniques for checking such properties; although relying on similar principles. Static analysis should be contrasted with *dynamic analysis* which concerns analysis of programs based on their execution, and includes e.g. testing, performance monitoring, fault isolation and debugging.

Static analysis does not in general guarantee the absence of runtime errors, and while it can reduce the need for testing or even detect errors that in practice cannot be found by testing, it is not meant to replace testing.

The following is a non-exhaustive list of runtime problems that typically cannot be detected by traditional compilers and may be difficult to find by testing, but which can be found by static analysis:

[4] Or so it is often claimed; in fact, even in ANSI/ISO Standard C there are many language constructs which are not semantically well-defined and which may lead to different behavior in different compilers.

[5] The borderline is not clear; some checks done by compilers, such as type checking in a statically typed language, are closer to runtime properties than syntactic ones. In fact, in a sufficiently rich type system some type checking must be done dynamically.

- Improper resource management: Resource leaks of various kinds, e.g. dynamically allocated memory which is not freed, files, sockets etc. which are not properly deallocated when no longer used;
- Illegal operations: Division by zero, calling arithmetic functions with illegal values (e.g. non-positive values to logarithm), over- or underflow in arithmetic expressions, addressing arrays out of bounds, dereferencing of null pointers, freeing already deallocated memory;
- Dead code and data: Code and data that cannot be reached or is not used. This may be only bad coding style, but may also signal logical errors or misspellings in the code;
- Incomplete code: This includes the use of uninitialized variables, functions with unspecified return values (due to e.g. missing return statements) and incomplete branching statements (e.g. missing cases in `switch` statements or missing `else` branches in conditional statements).

Other problems checked for by static analysis include non-termination, uncaught exceptions, race conditions etc.

In addition to finding errors, static analysis can also be used to produce more efficient code; in particular for "safe" languages like Java, where efficiency was not the primary objective. Many runtime tests carried out in Java programs can in practice be avoided given certain information about the runtime behavior. For instance, tests that array indices are not out-of-bounds can be omitted if we know that the value of the indices are limited to values in-bounds. Static analysis can provide such information.

Static analysis can also be used for type inference in untyped or weakly typed languages or type checking in languages with non-static type systems [21]. Finally static analysis can be used for debugging purposes (see e.g. [1]), for automatic test case generation (see e.g. [19]), for impact analysis (see e.g. [26]), intrusion detection (see e.g. [29]) and for software metrics (see e.g. [30]). However, in this paper we focus our attention on the use of static analysis for finding defects and software vulnerabilities which typically would not show up until the code is executed.

Most interesting properties checked by static analyses are *undecidable*, meaning that it is impossible, even in theory, to determine whether an arbitrary program exhibits the property or not. As a consequence static analyses are inherently *imprecise*—they typically infer that a property (e.g. a runtime error) *may* hold. This implies that

(i) if a program has a specific property, the analysis will usually only be able to infer that the program *may* have the property. In some special cases the analysis may also be able to infer that the program *does* have the property.

(ii) if the program does not have the property, there is a chance that (a) our analysis is actually able to infer this (i.e. the program *does not* have the property), but it may also happen that (b) the analysis infers that the program *may* have the property.

If the property checked for is a defect then we refer to case (ii)(b) as a *false positive*. Hence, if the analysis reports that a program may divide by zero we cannot tell in general whether it is a real problem (item (i)) or if it is a false positive (item (ii)(b)). The precision of the analysis determines how often false positives are reported. The more imprecise the analysis is, the more likely it is to generate false positives.

Unfortunately precision usually depends on analysis time. The more precise the analysis is, the more resource consuming it is, and the longer it takes. Hence, precision must be traded for time of analysis. This is a very subtle trade-off—if the analysis is fast it is likely to report many false positives in which case the alarms cannot be trusted. On the other hand a very precise analysis is unlikely to terminate in reasonable time for large programs.

One way to avoid false positives is to filter the result of the analysis, removing potential errors which are unlikely (assuming some measure of likelihood). However, this may result in the removal of positives which are indeed defects. This is known as a *false negative*—an actual problem which is not reported. False negatives may occur for at least two other reasons. The first case is if the analysis is too optimistic, making unjustified assumptions about the effects of certain operations. For instance, not taking into account that `malloc` *may* return null. The other case which may result in false negatives is if the analysis is incomplete; not taking account of all possible execution paths in the program.

There are a number of well-established techniques that can be used to trade-off precision and analysis time. A *flow-sensitive* analysis takes account of the control flow graph of the program while a *flow-insensitive* analysis does not. A flow-sensitive analysis is usually more precise—it may infer that `x` and `y` may be aliased (only) after line 10, while a flow-insensitive analysis only infers that `x` and `y` may be aliased (anywhere within their scope). On the other hand, a flow-sensitive analysis is usually more time consuming.

A *path-sensitive* analysis considers only valid paths through the program. It takes account of values of variables and boolean expressions in conditionals and loops to prune execution branches which are not possible. A path-insensitive analysis takes into account all execution paths—even infeasible ones. Path-sensitivity usually implies higher precision but is usually more time consuming.

A *context-sensitive* analysis takes the context—e.g. global variables and actual parameters of a function call—into account when analyzing a function. This is also known as *inter-procedural* analysis in contrast to *intra-procedural* analysis which analyses a function without any assumptions about the context. Intra-procedural analyses are much faster but suffer from greater imprecision than inter-procedural analyses.

Path- and context-sensitivity rely on the ability to track possible values of program variables; for instance, if we do not know the values of the variables in the boolean expression of a conditional, then we do not know whether to take the `then`-branch or the `else`-branch. Such *value analysis* can be more or less sophisticated; it is common to restrict attention to intervals (e.g. `0 < x < 10`), but some approaches rely on more general relations between several variables (e.g. `x > y+z`). Another important issue is aliasing (see e.g. [14,28]); when using pointers or arrays the value of a variable can be modified by modifying the value of another variable. Without a careful value and aliasing analyses we will typically have large numbers of false positives, or one has do ungrounded, optimistic assumptions about the values of variables.

The undecidability of runtime properties implies that it is impossible to have an analysis which always finds all defects and produces no false positives. A framework for static analysis is said to be *sound* (or *conservative* or *safe*) if all defects checked for are reported, i.e. there are no false negatives but there may be false positives. [6] Traditionally, most frameworks for static analysis have aimed for soundness while trying to avoid excessive reporting of false positives (e.g. the products from PolySpace). However, most commercial systems today (e.g. Coverity Prevent and Klocwork K7) are not sound (i.e. they will not find all actual defects) and also typically produce false positives.

It is sometimes claimed that static analysis can be applied to incomplete code (individual files and/or procedures). While there is some truth to this, the quality of such an analysis may be arbitrarily bad. For instance, if the analysis does not know how a procedure or subprogram in existing code is called from outside it must, to be sound, assume that the procedure is called in an arbitrary way, thus analyzing executions that probably cannot occur when the missing code is added. This is likely to lead to false positives. Similarly incomplete code may contain a call to a procedure which is not available, either because it is not yet written, or it is a proprietary library function. Such incomplete code *can* be analyzed but is also likely to lead to a large number of false positives and/or false negatives depending on if the analysis makes pessimistic or optimistic assumptions about the missing code.

On the positive side, it is often not necessary to provide complete code for missing functions or function calls. It is often sufficient to provide a stub or a top-level function that mimics the *effects* of the properties checked for.

The tools studied in this report adopt different approaches to deal with incomplete code and incremental analysis when only some code has been modified (as discussed in the next section).

3 A comparison of the tools

Shallow static analysis tools based on pattern matching such as FlexeLint [17] have existed since the late 1980s. Lately several sophisticated industrial-strength static analysis tools have emerged. In this report we study tools from three of the main

[6] Soundness can be used in two completely different senses depending on if the focus is on the reporting of defects or on properties of executions. In the former (less common) sense soundness would mean that all positives are indeed defects, i.e. there are no false positives. However, the more common sense, and the one used here, is that soundness refers to the assumptions made about the possible executions. Even if there is only a small likelihood that a variable takes on a certain value (e.g. `x=0`) we do not exclude that possibility. Hence if the analysis infers that `X` *may* be zero in an expression `1/x`, there is a possibility that there will be a runtime error; otherwise not. This is why a sound analysis may actually result in false positives, but no false negatives.

providers—PolySpace, Coverity and Klocwork. There are several other static analysis tools around, including PREfix/PREfast from Microsoft [3], Astree [7], which are not as widely available. A tool which has existed for some years but not until recently has become commercially available is CodeSonar from Grammatech, founded by Tim Teitelbaum and Tom Reps, which is similar in style and ambition level to Coverity Prevent and Klocwork K7, see [18]. Even if we focus here on tools intended for global and "deep" (=semantic) analysis of code, more lightweight tools like FlexeLint may still be useful in more interactive use and for local analysis.

There are also dynamic tools that aim for discovering some of the kinds of defects as the static analysis tools do. For example Insure++ [22] and Rational Purify [24] detect memory corruption errors.

A rough summary of major features of the three systems studied here can be found in Table 1. Such a table is by necessity incomplete and simplistic and in the following sub-section we elaborate on the most important differences and similarities. A more thorough exposition of the tools can be found in the full version of the paper, see [16].

3.1 Functionality provided

While all three tools have much functionality in common, there are noticeable differences; in particular when comparing PolySpace Verifier [15,23] against Coverity Prevent [10,11] and Klocwork K7 [20]. The primary aim of all three tools obviously is to find real defects, but in doing so any tool will also produce some false positives (i.e. false alarms). While Coverity and Klocwork are prepared to sacrifice finding all bugs in favor of reducing the number of false positives, PolySpace is not; as a consequence the former two will in general produce relatively few false positives but will also typically have some false negatives (defects which are not reported). It is almost impossible to quantify the rate of false negatives/positives; Coverity claims that approximately 20 to 30 per cent of the defects reported are false positives. Klocwork K7 seems to produce a higher rate of false positives, but stays in approximately the same league. However, the rate of false positives obviously depends on the quality of the code. The rate of false negatives is even more difficult to estimate, since it depends even more on the quality of the code. (Obviously there will be *no* false negatives if the code is already free of defects.) According to Coverity the rate of defect reports is typically around 1 defect per 1-2 KLoC.

PolySpace, on the other hand, does in general mark a great deal of code in orange color which means that it *may* contain a defect, as opposed to code that is green (no defects), red (definite defect) or grey (dead code). If orange code is considered a potential defect then PolySpace Verifier produces a high rate of false positives. However, this is a somewhat unfair comparison; while Coverity and Klocwork do not even give the developer the opportunity to inspect all potential defects, PolySpace provides that opportunity and provides instead a methodology in which the developer can systematically inspect orange code and classify it either as correct or faulty. In other words, Coverity and Klocwork are likely to "find some bugs", but provide no guarantees—the rest of the code may contain defects which are not even reported by the tool. PolySpace on the other hand can provide guarantees—if all code is green (or grey) it is *known* not to contain any bugs (wrt the properties checked for, that is). On the other hand it may be hard to eliminate all orange code.

All three tools rely at least partly on inter-procedural analyses, but the ambition level varies significantly. PolySpace uses the most advanced technical solution where relationships between variables are approximated by convex polyhedra [8] and all approximations are sound—that is, no execution sequences are forgotten, but some impossible execution paths may be analyzed due to the approximations made. Coverity Prevent and Klocwork K7 account only of interval ranges of variables in combination with "simple" relationships between variables in a local context with the main purpose to prune some infeasible execution paths, but do not do as well as PolySpace. Global variables and nontrivial aliasing are not accounted for or treated only in a restricted way. As a consequence neither Coverity nor Klocwork take all possible behaviors into account which is one source of false negatives. It is somewhat unclear how Coverity Prevent and Klocwork K7 compare with each other, but impression is that the former does a more accurate analysis.

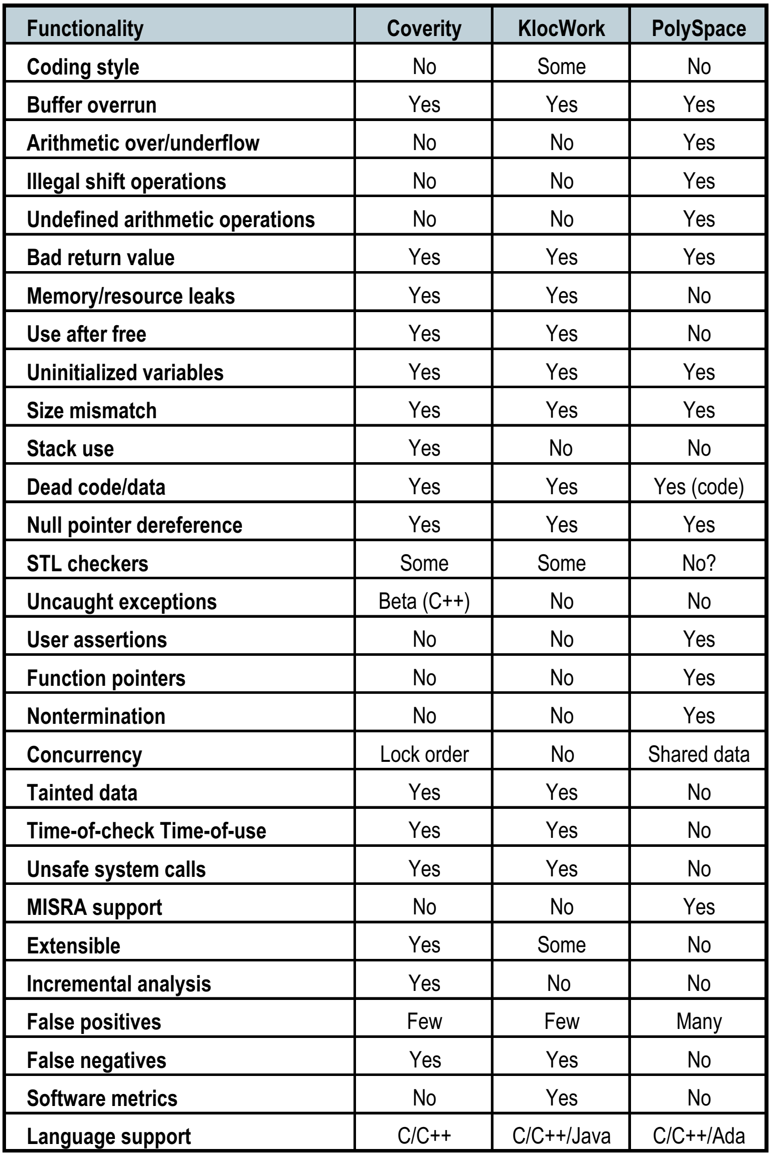
Another consequence of the restricted tracking of arithmetic values of variables in Coverity Prevent and Klocwork K7 is that the products are not suitable for detecting arithmetic defects, such as over- and underflows or illegal operations like division by zero. The products did not even provide arithmetic checkers at the time of the study. PolySpace on the other hand does provide several arithmetic checkers, setting it apart from the others.

While PolySpace is the only tool that provides arithmetic checkers, it is also the only one among the three which does not provide any checkers for resource leaks; in particular there is no support for discovering defects in dynamic management (allocation and deallocation) of memory. As a consequence there are also no checkers e.g. for "use-after-free". This lack can perhaps be explained by PolySpace's focus on the embedded systems market, involving safety or life critical applications where no dynamic allocation of memory is possible or allowed.

While PolySpace appears to be aiming primarily for the embedded systems market, Klocwork and Coverity have targeted in particular networked systems and applications as witnessed, for instance, by a range of security checkers. Klocwork and Coverity address essentially the same sort of security issues ranging from simple checks that critical system calls are not used inappropriately to more sophisticated analyses involving buffer overruns (which is also supported by PolySpace) and the potential use of so-called tainted (untrusted) data. The focus on networked application also explains the support for analyzing resource leaks since dynamic management of resources such as sockets, streams and memory is an integral part of most networked applications.

Coverity supports incremental analysis of a whole system, where only parts have been changed since last analysis. Results of an analysis are saved and reused in subsequent analyses. An automatic impact analysis is done to detect and, if necessary, re-analyze other parts of the code affected indirectly by the change. Such

Table 1
Summary of features of Coverity Prevent, Klocwork K7 and PolySpace Verifier



Functionality	Coverity	KlocWork	PolySpace
Coding style	No	Some	No
Buffer overrun	Yes	Yes	Yes
Arithmetic over/underflow	No	No	Yes
Illegal shift operations	No	No	Yes
Undefined arithmetic operations	No	No	Yes
Bad return value	Yes	Yes	Yes
Memory/resource leaks	Yes	Yes	No
Use after free	Yes	Yes	No
Uninitialized variables	Yes	Yes	Yes
Size mismatch	Yes	Yes	Yes
Stack use	Yes	No	No
Dead code/data	Yes	Yes	Yes (code)
Null pointer dereference	Yes	Yes	Yes
STL checkers	Some	Some	No?
Uncaught exceptions	Beta (C++)	No	No
User assertions	No	No	Yes
Function pointers	No	No	Yes
Nontermination	No	No	Yes
Concurrency	Lock order	No	Shared data
Tainted data	Yes	Yes	No
Time-of-check Time-of-use	Yes	Yes	No
Unsafe system calls	Yes	Yes	No
MISRA support	No	No	Yes
Extensible	Yes	Some	No
Incremental analysis	Yes	No	No
False positives	Few	Few	Many
False negatives	Yes	Yes	No
Software metrics	No	Yes	No
Language support	C/C++	C/C++/Java	C/C++/Ada

an incremental analysis may take significantly less time than analyzing the whole system from scratch. With the other tools analysis of the whole system has to be redone. All of the tools provide the possibility to analyze a single file. However such an analysis will be much more shallow than analyzing a whole system where complete paths of execution can be analyzed.

Both Klocwork and Coverity provide means for writing user defined checkers and integrating them with the analysis tools, see e.g. [9,4]. However, the APIs are nontrivial and writing new, non-trivial checkers is both cumbersome and error prone. There are no explicit guidelines for writing correct checkers and no documented support for manipulation of abstract values (e.g. interval constraints). There is also no support for reusing the results of other checkers. Termination of the checker is another issue which may be problematic for users not familiar with the mathematical foundations of static analysis, see e.g. [6,27].

All three tools support analysis of the C programming language and C++. At the initial time of this study only Klocwork supported analysis of Java but Coverity was announcing a new version of Prevent with support for Java. Only PolySpace supported analysis of Ada. Klocwork was the only provider which claimed to handle mixed language applications (C/C++/Java).

The downside of PolySpace's sophisticated mechanisms for tracking variable values is that the tool cannot deal automatically with very large code bases without manual partitioning of the code. While Coverity Prevent and Klocwork K7 are able to analyze millions of lines of code off-the-shelf and overnight, PolySpace seems to reach the complexity barrier already at around 50 KLoC with the default settings. On the other hand PolySpace advocates analyzing code in a modular fashion. Analysis time is typically not linear in the number of lines of code—analyzing 10 modules of 100 KLoC is typically orders of magnitude faster than analyzing a single program consisting of 1,000 KLoC. However this typically involves human intervention and well-defined interfaces (which may be beneficial for other quality reasons...)

On the more exotic side Coverity provides a checker for *stack use*. It is unclear how useful this is since there is no uniform way of allocating stack memory in different compilers. Klocwork is claimed to provide similar functionality but in a separate tool. PolySpace set themselves aside from the others by providing checkers for non termination, both of functions and loops. Again it is unclear how useful such checkers are considering the great amount of research done on *dedicated* algorithms for proving termination of programs (see e.g. [13,2]). Coverity has a checker for uncaught exceptions in C++ which was still a beta release. PolySpace provides a useful feature in their support for writing general *assertions* in the code. Such assertions are useful both for writing stubs and may also be used for proving partial correctness also of functional properties; see [25].

None of the tools provide very sophisticated support for dealing with concurrency. Klocwork currently provides no support at all. Coverity is able to detect some cases of mismatched locks but does not take concurrency into account during analysis of concurrent threads. The only tool which provides more substantial support is PolySpace which is able to detect shared data and whether that data is protected or not.

Both Coverity and Klocwork have developed lightweight versions of their tools aimed for frequent analysis during development. These have been integrated with Eclipse IDEs. However the defect databases for Coverity and Klocwork have not been integrated into Eclipse IDEs or TPTP. PolySpace has integrated with the Rhapsody UML tool to provide a UML static analysis tool. It analyzes generated code and links back references to the UML model to point out where defects have been detected. Besides that PolySpace has its general C++ level advantages with a sound analysis (no false negatives) and presumably problems with analyzing large code bases (larger than 50-100 KLoC)—a restriction which should be more severe in the UML situation compared to hand-coded C++.

3.2 Experiences at Ericsson

A number of independent evaluations of static analysis tools were performed by development groups at Ericsson. Coverity was evaluated by several groups. Klocwork has also been subject to evaluations but not quite as many. There was an attempt to use PolySpace for one of the smallest applications, but the evaluation was not successful; the tool has either presented no results within reasonable time (a couple of days' execution) or the results were too weak to be of use (too much orange code to analyze). We do not know if this was due to the tool itself or to the actual configuration of the evaluations. It would have been valuable to compare results from PolySpace, which is sound, to those of Klocwork and Coverity. Perhaps that would give some hint on the false negative rate in Klocwork and Coverity.

Some general experiences from use of Coverity and Klocwork were:

- The tools are easy to install and get going. The development environment is easy to adapt and no incompatible changes in tools or processes are needed.
- The tools are able to find bugs that would hardly be found otherwise.
- It is possible to analyze even large applications with several million lines of code and the time it takes is comparable to build time.
- Even for large applications the false positive rate is manageable.
- Several users had expected the tools to find more defects and defects that were more severe. On the other hand, several users were surprised that the tools found bugs even in applications that had been tested for a long time. There might be a difference in what users find reasonable to expect from these tools. There might also be large differences in what users classify as a false positive, a bug or a severe bug.
- It is acceptable to use tools with a high false positive rate (such as FlexeLint) if the tool is introduced in the beginning of development and then used continuously.
- It is unacceptable to use tools with a high false positive rate if the product is large and the tool is introduced late in the development process.
- Many of the defects found could not cause a crash in the system as it was defined and used at the moment. However if the system would be only slightly changed or the usage was changed the defect could cause a serious crash. Therefore these problems should be fixed anyway.
- Even if the tools look for the same categories of defects, for instance memory leaks, addressing out of array bounds etc, the defects found in a given category by one tool can be quite different from those found by another tool.
- Handling of third party libraries can make a big difference to analysis results.

Declarations for commercial libraries that come with the analysis tool can make the analysis of own code more precise. If source for the library is available defects in the library can be uncovered, which may be as important to the quality of the whole application as the own code.

- There are several aspects of the tools that are important when making a tool selection that has not been a part of the comparison in this paper; such as pricing, ease of use, integration in IDEs, other functionality, interactiveness etc.

Below follows some more specific results from some of the evaluations. We do not publish exact numbers of code sizes and found bugs etc for confidentiality reasons since some of the applications are commercial products in use.

Evaluation 1 (Coverity and FlexeLint): The chosen application had been thoroughly tested, both with manually designed tests and systematic tests that were generated from descriptions. FlexeLint was applied and produced roughly 1,200,000 defect reports. The defects could be reduced to about 1,000 with a great deal of analysis and following filtering work. These then had to be manually analyzed. Coverity was applied to the same piece of code and found about 40 defects; there were very few false positives and some real bugs. The users appreciated the low false positive rate. The opinion was that the defects would hardly have been found by regular testing.

The users had expected Coverity to find more defects. It was believed that there should be more bugs to be found by static analysis techniques. It was not known if this was the price paid for the low false positive rate or if the analyzed application actually contained only a few defects. The users also expected Coverity to find more severe defects. Many of the findings were not really defects, but code that simply should be removed, such as declarations of variables that were never used. Other defects highlighted situations that could not really occur since the code was used in a restricted way not known to the analysis tool.

Evaluation 2 (Coverity): A large application was analyzed with Coverity. Part of the code had been previously analyzed with FlexeLint. The application had been extensively tested.

Coverity was perceived both as easy to install and use, and no modifications to existing development environment was needed. The error reports from the analysis were classified as follows

- 55 per cent were no real defects but perceived only as poor style,
- 2 per cent were false positives,
- 38 per cent were considered real bugs, and 1 per cent were considered severe.

The users appreciated that a fair number of defects were found although the code had already been thoroughly tested.

Evaluation 3 (Coverity and Klocwork): An old version of an application that was known to have some memory leaks was analyzed using Coverity and Klocwork.

In total Klocwork reported 32 defects including 10 false positives and Coverity reported 16 defects including 1 false positive. Only three defects were common to both tools! Hence Klocwork found more defects, but also had a larger false positive rate. Although the tools looked for similar defects the ones actually found were largely specific to each tool. This suggests that each of the tools fail in finding many defects.

Looking at only the memory leaks the results were similar. Klocwork reported 12 defects of which 8 were false, totalling 4 real defects and Coverity reported 7 defects all of which were true defects. None of the tools found any of the known memory leaks.

Evaluation 4 (Coverity and Klocwork): Old versions of two C++ products were analyzed with Coverity and Klocwork. Trouble reports for defects that had been detected by testing were available. One purpose was to compare how many faults each of the tools found. Another purpose was to estimate how many of the faults discovered in testing were found by the static analysis tools.

Coverity found significantly more faults and also had significantly less false positives than Klocwork. One of the major reasons for this was the handling of third party libraries. Coverity analyzed the existing source code for the libraries and found many faults in third party code! Klocwork did not analyze this code and hence did not find any of these faults. Besides that the analysis of the libraries that Coverity did resulted in fewer false positives in the application code since it could be derived that certain scenarios could not occur.

The time of analyses was about the same as build time for both tools—i.e. is good enough for overnight batch runs but not for daily, interactive use during development.

Both tools lacked integration with CM tool Clearcase, the source code had to be copied into the repository of the analysis tools. There was no way to do inspection of analysis results from an IDE, but the reviews had to be done in the GUI of the analysis tools.

Coverity was preferred by the C++ developers. It had incremental analysis that would save time and it could easily analyze and report on single components.

Although the main part of the evaluation was on old code some studies were done on programs during the development. The development code had more warnings and most of them were real faults; most of these were believed to have been found during function test. It had been anticipated that more faults would be found in low level components, but these components proved to be stable and only a few defects were discovered. More faults were however found in high level components with more frequent changes.

Evaluation 5 (Coverity, Klocwork and CodePro): A Java product with known bugs was analyzed. A beta version of Coverity Prevent with Java analysis capabilities was used. None of the known bugs were found by the tools. Coverity found more real faults and had far less false positives than Klocwork. For Coverity one third of the warnings were real bugs.

Klocwork generated many warnings; 7 times the number of warnings of Coverity. The missing analysis of the third party library seemed to be the major reason. However, Klocwork does a ranking of the potential defects and when only the four most severe levels of warnings were considered the results were much better—there were few false positives.

CodePro Analytix (developed and marketed by Instantiations) is a tool aimed for analysis during development. It is integrated into the Eclipse IDE and the results of an analysis cannot be persistently saved, but only exist during the development session with the IDE. The analysis is not as deep as that of Coverity or Clockwork, but is faster and can easily be done interactively during development. The tool generates a great deal of false positives, but these can be kept at a tolerable level by choosing an appropriate set of analysis rules. No detailed analysis was done of the number of faults and if they were real faults or not.

In this evaluation there was a large difference in the number of warnings generated, Coverity 92 warnings, Klocwork 658 warnings (in the top four severities 19), CodePro 8,000 warnings (with all rules activated).

4 Conclusions

Static analysis tools for detection of runtime defects and security vulnerabilities can roughly be categorized as follows

- **String and pattern matching approaches**: Tools in this category rely mainly on syntactic pattern matching techniques; the analysis is typically path- and context-insensitive. Analyses are therefore shallow, taking little account of semantic information except user annotations, if present. Tools typically generate large volumes of false positives as well as false negatives. Tools (often derivatives of the lint program) have been around for many years, e.g. FlexeLint, PC-Lint and Splint. Since the analysis is shallow it is possible to analyze very large programs, but due to the high rate of false positives an overwhelming amount of post-processing may be needed. These tools are in our opinion more useful for providing almost immediate feedback in interactive use and in combination with user annotations.
- **Unsound dataflow analyses**: This category of tools which have emerged recently rely on semantic information; not just syntactic pattern matching. Tools are typically path- and context-sensitive but the precision is limited so in practice the tools have to analyze also many impossible paths or make more-or-less justified guesses what paths are (im-)possible. This implies that analyses are unsound. Aliasing analysis is usually only partly implemented, and tracking of possible variable values is limited; global variables are sometimes not tracked at all. A main objective of the tools, represented e.g. by Coverity Prevent and Klocwork K7, is to reduce the number of false positives and to allow for analysis of very large code bases. The low rate of false positives (typically 20–30 per cent

in Coverity Prevent) is achieved by a combination of a unsound analysis and filtering of the error reports. The downside is the presence of false negatives. It is impossible to quantify the rate since it depends very much on the quality of the code, but in several evaluations Coverity and Klocwork find largely disjoint sets of defects. This category of tools provide no guarantees—the error reports may or may not be real defects (it has to be checked by the user), and code which is not complained upon may still be defective. However, the tools will typically find *some* bugs which are hard to find by other techniques.

- **Sound dataflow analyses**: Tools in this category are typically path- and context-sensitive. However, imprecision may lead to analysis of some infeasible paths. They typically have sophisticated mechanisms to track aliasing and relationships between variables including global ones. The main difficulty is to avoid excessive generation of false positives by being as precise as possible while analysis time scales. The only commercial system that we are aware of which has taken this route is PolySpace Verifier/Desktop. The great advantage of a sound analysis is that it gives some guarantees: if the tool does not complain about some piece of code (the code is green in PolySpace jargon) then that piece of code must be free of the defects checked for.

There is a forth category of tools which we have not discussed here—namely tools based on *model checking* techniques [5]. Model checking, much like static analysis, facilitates traversal and analysis of all reachable states of a system (e.g. a piece of software), but in addition to allowing for checking of runtime properties, model checking facilitates checking of functional properties (e.g. safety properties) and also so-called temporal properties (liveness, fairness and real-time properties). There are commercial tools for model checking hardware systems, but because of efficiency issues there are not yet serious commercial competitors for software model checking.

It is clear that the efficiency and quality of static analysis tools have reached a maturity level were static analysis is not only becoming a viable complement to software testing but is in fact a required step in the quality assurance of certain types of applications. There are many examples where static analysis has discovered serious defects and vulnerabilities that would have been very hard to find using ordinary testing; the most striking example is perhaps the Scan Project [12] which is a collaboration between Stanford and Coverity that started in March, 2006 and has reported on more than 7,000 defects in a large number of open-source projects (e.g. Apache, Firebird, FreeBSD/Linux, Samba) during the first 18 months.

However, there is still substantial room for improvement. Sound static analysis approaches, such as that of PolySpace, still cannot deal well with very large code bases without manual intervention and they produce a large number of false positives even with very advanced approximation techniques to avoid loss of precision. Unsound tools, on the other hand, such as those from Coverity and Klocwork do scale well, albeit not to the level of interactive use. The number of false positives is surprisingly low and clearly at an acceptable level. The price to be paid is that they are not sound, and hence, provide no guarantees: they may (and most likely will) find some bugs, possibly serious ones. But the absence of error reports from such a tool only means that the *tool* was unable to find any potential defects. As witnessed in the evaluations different unsound tools tend to find largely disjoint defects and are also known not to find known defects. Hence, analyzed code is likely to contain dormant bugs which can only be found by a sound analysis.

Most of the evaluations of the tools have been carried out on more or less mature code. We believe that to fully ripe the benefits of the tools they should not be used only at the end of the development process (after testing and/or after using e.g. FlexeLint), but should probably be used throughout the development process. However, the requirements on the tools are quite different at an early stage compared to at acceptance testing. Some vendors "solve" the problem by providing different tools, such as PolySpace Desktop and PolySpace Verifier. However, we rather advocate giving the user means of fine-tuning the behavior of the analysis engine. A user of the tools today has very limited control over precision and the rate of false positives and false negatives—there are typically a few levels of precision available, but the user is basically in the hands of the tools. It would be desirable for the user to have better control over precision of the analyses. There should for example be a mechanism to fine-tune the effort spent on deriving value ranges of variables and the effort spent on aliasing analysis. For some users and in certain situations it would be acceptable to spend five times more analysis time in order to detect more defects. Before an important release it could be desirable to spend much more time than on the day to day analysis runs. In code under development one can possibly live with some false negatives and non-optimal precision as long as the tool "finds some bugs". As the code develops one can improve the precision and decrease the rate of false positives and negatives; in particular in an incremental tool such as Coverity Prevent. Similarly it would be desirable to have some mechanism to control the aggressiveness of filtering of error reports.

References

[1] Ball, T. and S. Rajamani, *The SLAM Project: Debugging System Software via Static Analysis*, ACM SIGPLAN Notices **37** (2002), pp. 1–3.

[2] Ben-Amram, A. M. and C. S. Lee, *Program Termination Analysis In Polynomial Time*, ACM Trans. Program. Lang. Syst. **29** (2007).

[3] Bush, W., J. Pincus and D. Sielaff, *A Static Analyzer For Finding Dynamic Programming Errors*, Software, Practice and Experience **30** (2000), pp. 775–802.

[4] Chelf, B., D. Engler and S. Hallem, *How to Write System-specific, Static Checkers in Metal*, in: *PASTE '02: Proc. 2002 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering* (2002), pp. 51–60.

[5] Clarke, E., O. Grumberg and D. Peled, "Model Checking," MIT Press, Cambridge, MA, USA, 1999.

[6] Cousot, P. and R. Cousot, *Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs by Construction Or Approximation of Fixpoints*, in: *Conf. Record of the Fourth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (1977), pp. 238–252.

[7] Cousot, P., R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival, *The ASTRÉE Analyser*, in: M. Sagiv, editor, *Proceedings of the European Symposium on Programming (ESOP'05)*, Lecture Notes in Computer Science **3444** (2005), pp. 21–30.

[8] Cousot, P. and N. Halbwachs, *Automatic Discovery of Linear Restraints Among Variables of a Program*, in: *Conf. Record of the Fifth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (1978), pp. 84–97.

[9] Coverity Inc., "Coverity Extend$^{\mathrm{TM}}$ User's Manual (2.4)," (2006).

[10] Coverity Inc., *Coverity Prevent^{TM}: Static Source Code Analysis for C and C++* (2006), product information.

[11] Coverity Inc., "Coverity Prevent$^{\mathrm{TM}}$ User's Manual 2.4," (2006).

[12] Coverity Inc., *The Scan Ladder* (2007), URL: http://scan.coverity.com.

[13] Dershowitz, N. and Z. Manna, *Proving Termination With Multiset Orderings*, Commun. ACM **22** (1979), pp. 465–476.

[14] Deutsch, A., *Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting*, in: *Proc. PLDI* (1994), pp. 230–241.

[15] Deutsch, A., *Static Verification of Dynamic Properties*, White paper, PolySpace Technologies Inc (2003).

[16] Emanuelsson, P. and U.Nilsson, *A Comparative Study of Industrial Static Analysis Tools (Extended Version)*, Technical Reports in Computer and Information Science no 2008:3, Linköping University Electronic Press (2008), URL: http://www.ep.liu.se/ea/trcis/2008/003/.

[17] Gimpel Software, "PC-lint/FlexeLint," (1999), URL: http://www.gimpel.com/lintinfo.htm.

[18] GrammaTech Inc., *Overview of GrammaTech Static Analysis Technology* (2007), white paper.

[19] King, J., *Symbolic Execution and Program Testing*, Comm. ACM **19** (1976), pp. 385–394.

[20] Klocwork Inc., *Detected Defects and Supported Metrics* (2005), K7 product documentation.

[21] Palsberg, J. and M. Schwartzbach, *Object-Oriented Type Inference*, in: *Conf Proc Object-Oriented Programming Systems, Languages, And Applications (OOPSLA '91)* (1991), pp. 146–161.

[22] Parasoft Inc., *Automating C/C++ Runtime Error Detection With Parasoft Insure++* (2006), white paper.

[23] PolySpace Technologies, "PolySpace for C Documentation," (2004).

[24] Rational Software, *Purify: Fast Detection of Memory Leaks and Access Errors*, White paper (1999).

[25] Rosenblum, D. S., *A Practical Approach to Programming With Assertions*, IEEE Trans. Softw. Eng. **21** (1995), pp. 19–31.

[26] Ryder, B. and F. Tip, *Change Impact Analysis For Object-Oriented Programs*, in: *Proc. of 2001 ACM SIGPLAN-SIGSOFT workshop on Program Analysis For Software Tools And Engineering (PASTE '01)* (2001), pp. 46–53.

[27] Schwartzbach, M., *Lecture Notes on Static Analysis* (2006), BICS, Univ. Aarhus, URL: http://www.brics.dk/~mis/static.pdf.

[28] Steensgaard, B., *Points-to Analysis in Almost Linear Time*, in: *ACM POPL*, 1996, pp. 32–41.

[29] Wagner, D. and D. Dean, *Intrusion Detection via Static Analysis*, in: *Proc. of 2001 IEEE Symp. on Security and Privacy (SP'01)* (2001), pp. 156–168.

[30] Wagner, T., V. Maverick, S. Graham and M. Harrison, *Accurate Static Estimators For Program Optimization*, in: *Proc. of ACM SIGPLAN 1994 Conf. on Programming Language Design And Implementation (PLDI '94)* (1994), pp. 85–96.