Goanna Static Analysis at the NIST Static Analysis Tool Exposition

Mark Bradley, Ansgar Fehnker, Ralf Huuck and Paul Steckler
Red Lizards Software
Email: `info@redlizards.com`
Url: `redlizards.com`

Abstract

In 2010 Red Lizard software participated for the first time in the Static Analysis Tool Exposition (SATE) organized by the National Institute of Standards and Technology (NIST) with the static analysis tool *Goanna*. The aim of SATE is to advance static analysis research and solutions that detect serious security and quality issues in source code. Goanna is a static analysis solution for the desktop and server, which find detects bugs in C/C++ source code by a combination of static analysis techniques with model checking technology. This report will give a brief introduction to source code analysis with Goanna, it describes how the submission to SATE was prepared, the results that were obtained, and some of the lessons that were learned in the process.

1 Introduction

Software development cycles are a major competitive aspect in many market segments including mobile phone handsets, games, and consumer electronics. The obvious goal is deliver software as fast as possible, as cheaply as possible at the highest possible quality. For these reasons automation and tool support play an increasing role. VDC estimates that around 50% of the software development costs result from testing and debugging.

A category tools that help to reduce these costs are source code analysis tools like *Goanna*. These tools use a combination of techniques to detect deficiencies of the source code in the programming phase. Integrating the use of these tools in the SDLC has the numerous benefits. First, it reduces the number of defects detected by testing, and thus the number of test cycles. It is estimated that finding bug by testing can be up to 80 times more costly than finding them in the programming phase. These tool can be used to ensure that code meets certain coding standards, which will help to make keep it maintainable. It help the programmer with debugging his own code more efficiently, i.e. coding itself becomes more efficiently, especially if these tools are available in the development environment. Another benefit is that these tool can find potential bugs that are difficult, if not impossible, to find through traditional means. And of course there is the benefit that these tools give the programmer automatic, and often instant feedback on his programming.

Goanna by Red Lizards Software is an integrated C/C++ source code analysis tools for mission-critical industries. It is the first solution in the market that combines the automated technologies of static analysis with model checking. There are two product lines: *Goanna Studio*, the IDE version, and *Goanna Central*, the command line version.

Goanna Studio is the desktop solution, integrated with either the Eclipse IDE both for Linux and Windows, or Microsoft Visual Studio, version 2005, 2008 and 2010. Red Lizard Software was a SimShip partner when Microsoft launched Visual Studio 2010, the only static analysis solution in the market to be jointly launched. Goanna Studio is a developer tool, fully integrated into the IDEs, and offers the full solution to be used while programming.

Goanna Central is the command line version that can be integrated with the nightly build system. It supports all common common C/C++ dialects such as ANSI/ISO C, the Microsoft dialects of C/C++, and GNU C/C++, and the most common build systems such as make, cmake, or scons.

NIST specified for SATE five C/C++ code bases as test bed for the participating static analysis tools. One code base was for Dovecot, an open source IMAP and POP3 server, and two different version for each Wireshark, a network protocol analyzer, and for Chrome, and internet browser. To analyze this code we used Goanna Central for Linux. The next section introduces the types of analyzes that Goanna performs, Section 4 describes which checks and which version of Goanna were used in SATE, Section 5 presents the overall results, while Section 6 selects and explain a few warnings from the tool report.

2 Goanna Static Analysis

Goanna checks for bugs, memory leaks and security vulnerabilities, is fully path-sensitive and inter-procedural. It uses a combination of techniques, from pattern matching, to data flow analysis, and model checking. In the following we describe the main techniques used by modern static analysis tools for detecting security vulnerabilities in source code.

2.1 Tree-Pattern Matching

Pattern matching is the simplest technology applied by virtually all static analysis tools. Pattern matching is when a tool tries to identify points of interest in the program's syntax, typically by defining queries on either the plain source code or on the *abstract syntax tree* (AST), i.e. the representation of the source code after parsing. For instance, it is well known that in C/C++ programs the library function `strcopy` is vulnerable to security exploitation if not used absolutely correctly. Hence, a simple keyword scan for `strcopy` can help with this. Queries can become more complicated, and a series of interdependent queries may be used for more advanced checks, for example to identify inconsistent use of semantic attributes. Our analysis tool Goanna uses such tree-pattern matching [BFK02] on the AST. However, pattern matching is fundamentally limiting, since it only searches for keywords and their context, but is unable to take control flow, data flow, or other semantic information into account.

2.2 Data Flow Analysis

The next step up in terms of technology is about understanding how certain constructs are related. Data flow analysis makes use of the structure of program, in particular its *control flow graph* (CFG). It is a standard compiler technique to examine the flow of information between variables and other elements of interest that can be syntactically identified. An example is checking for uninitialized variables. We can syntactically identify program locations that are variable declarations and uses of variables either as a reading operation (such as the right hand side of an assignment) or a write (such as the left hand side of an assignment). Data flow analysis enables us to examine if there is a read operation to a variable without a prior write. These and similar tools are used to analyze how certain elements in a program are related. A limitation of data flow analysis is that it assumes that control flows can merge, which is an over-approximation of the real program behavior. For some checks it can be necessary to distinguish individual paths in a program; or accept the consequence that over-approximation might lead to false warnings.

2.3 Model Checking

Model checking was developed in the early 1980s as a technique to check whether larger concurrent systems satisfy given temporal properties [CE82, QS82, Pnu77]. It is essentially a technique to determine whether all paths in a graph satisfy certain ordering of events along that path. Unlike other techniques that enumerate paths it does not put a limit of the number, the length or the branching of paths. Another advantage is that if it finds a violation, it will also return a counter example path. For their solution the original authors received the Turing Award in 2007.

Model checking was originally applied to the formal verification of protocols and hardware designs. In recent years, a strong push has been made towards software verification, and effective methods have been developed to overcome scalability challenges. Or tool Goanna is the first tool that managed to apply model checking on a grand scale to static program analysis. This means that we use it to analyze millions of lines of code for over one hundred different classes of checks. The advantage of this approach that is oftentimes faster than existing data flow and path enumeration approaches. Moreover, it also tends to be more powerful as it allows specifying complex relations between program constructs.

2.4 Abstract Data Tracking

To detect buffer overflows, division-by-zero errors and other defects it is helpful to know as much as possible about the values that can occur at certain program locations. A technique to approximate and track data values is commonly called abstract interpretation [Cou81]. Abstract interpretation estimates for every variable at every program location all the potential range of variables. For example, for an array access, all potential index values can be estimated. Different domains can be used for data tracking, with varying levels of precision. There is a trade-off between the precision and speed of the analysis. Goanna uses a variant of interval constraint analysis that is reasonably precise while fast, but can also use automated theorem proving techniques (SMT solving) to validate individual program paths.

2.5 Whole Program Analysis

While the aforementioned techniques help to detect various security issues and are often complimentary, one of the key factors to success is to scale to large code bases. Many potential issues require to understand the overall call structure of a program, and being able to track data and control flow across function boundaries. To overcome this challenges advanced tools introduce the ability to generate *function summaries* automatically. These summaries contain only the essential information of a function that is needed for particular security check. Instead of propagating information of the whole function, which can be prohibitively large, only the summary information is used. Generating summaries is an iterative process taking the mutual impact (from recursion etc.) into account.

3 The Goanna Static Analysis Tool

Goanna is an automated analysis tool that does not execute code as in traditional testing, but examines the code structure, the keywords, their inter-relation as well as the data and information flow across the whole source code base. These techniques can be fully automated and scale to millions of lines of code. As such Goanna is able to identify many classes of security issues automatically at software development time. Moreover, Goanna is the only tool that combines all the techniques mentioned in Section **??**. The Goanna tool is fully path-sensitive and performs whole-program analysis. For a detailed technical description of the underlying formal techniques see [FHJ$^+$07].

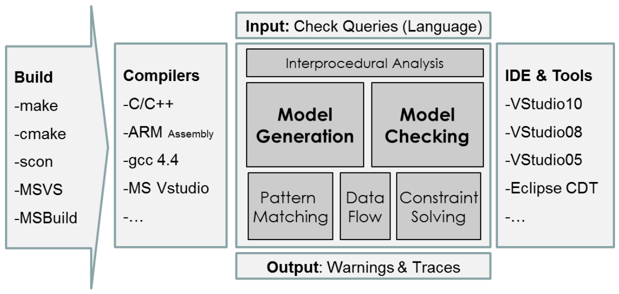

Figure 1: Goanna static analysis architecture

3.1 Supported Languages and Architectures

The Goanna tool is currently implemented to handle C/C++ source code. The architecture and the technology are in theory adaptable to other imperative programming languages such as $C\#$ or Java. However, the C/C++ programming language is still predominant in many mission-critical systems while at the same time it easily suffers from potential exploits.

The Goanna tool fully parses C/C++ source code including compiler specific dialects and extensions such as GNU, Visual C++ or GreenHills. It is available for Microsoft Windows as well as various Linux on either 32-bit or 64-bit machines. On top of that Goanna supports the cross-compilation to a number of embedded platforms. It is possible to integrate the tool in popular build environments, such as make or scons.

3.2 Analysis Framework

Figure 1 depicts how the different types of analysis are embedded in Goanna. The core of the tool uses model checking, while the other techniques are used for particular checks, and to assist in the generation of the models. A distinguishing feature of Goanna is that it architecturally modularizes the core analysis engine and the checks that are performed. Goanna has a specification language to define checks; this allow for the rapid development of new checks, separately from the core algorithms, and any future improvement of the engine will be effective for all checks.

In short, Goanna uses model checking for all path-sensitive check on how program information relates. It uses tree-pattern matching to identify certain locations and operations of interest and abstract data tracking by constraint solving for specialized checks, such as buffer overruns, shifting beyond bounds and overflow errors. Moreover, Goanna uses a number of heuristics and advanced features such as interprocedural whole-program analysis to achieve speed and scalability.

4 The Entry

NIST specified five code bases for SATE, but the participants were free to configure their tool to fit the code base. For the entry to SATE we used a development snapshot of the then current version of Goanna 2.0 (r7225), running on Linux Ubuntu Server 8.04. None of the code bases required special configuration above what would be required to configure a normal build. All of the code bases specified by NIST use a combination of `configure` and `make` files; all that was required is to configure them with Goanna, and in addition, since we are using interprocedural analysis, to specify a database file. The remaining configuration dealt with formatting the output to comply with the NIST specification.

For analysis of Chrome 5.0.375.54 and 5.0.375.70 we used the standard version of goanna. For Dovecot and Wireshark 1.2.0 and 1.2.9 we used a prototype version. It differed from the standard version in that it used an additional SMT solver to evaluate paths, and an off-the-shelf model checker, rather than our custom build checker. The latter has no influence on the results. It was used to for the prototype because it supported a secondary feature that was not supported at the time by the in house checker. This came at an expense of performance, since it has not been optimised for the use with Goanna. The in-house checker can be up to 2 orders of magnitude faster than the off-the-shelf solution.

For our submission we selected the 55 default checks of Goanna 2.0. These checks are enabled by default, because they detect the more serious issues that should be addressed by any programmer. We omitted the other checks which typically deal more with either stylistic requirements, such as the unused parameter check, or warn very conservatively, i.e. even if an actual bug is unlikely. The remainder of this section will give a quick overview of the classes of checks that we included.

Array bounds. These checks are concerned with correct array access. They will warn if they detect an out of bound array access. We included three checks of this class.

Arithmetic errors. These checks are concerned with arithmetic errors such as division by zero or out of bound shifts. We included ten checks of this class.

C++ copy control. These checks are concerned with the correct initialization, construction or destruction. This class only applies to C++. We included three checks from this class.

C++ usage. These checks are concerned with the correct use of C++ features. For example, they warn if a nonvirtual destructor is defined for an abstract class.

Potentially unexpected behavior. These checks are concerned with features and idiosyncracies of C/C++ that are often poorly understood. An example is an ambiguous use of an `else`. Two of these checks have been included.

Function pointer usage. These are concerned with using accidentally a function pointer incorrectly, for example in an arithmetic expression. We included two checks of this class.

Memory usage. These checks deal with the correct usage of stacks, arrays and pointers, like storing a stack address in a global variable. Ten of these checks have been included.

Pointer misuse. These checks deal more specifically with the correct use of pointers, like testing for NULL pointer. We included five checks of this class.

Redundant code. These checks deal with code that may be redundant, like dead code or trivial conditions. We included six checks of this class.

Semantic attributes. These checks are concerned with the correct use of the GNU C language extension with semantic attributes like `const` or `pure`. We included four checks from this class.

Unspecified behavior. These checks deal with code for which the C standard does not define behavior. This includes cases where the execution order is undefined, or cases where initialization is undefined. We included five checks of this class.

As mentioned before, we selected the default checks, which report serious issues, and assume that the programmer is programming defensively. They try not to warn in cases that are common and accepted programming practice. And example would be redundant `return` statements. While the warning would be technically correct, redundant `return` statements are often included to deal with compiler warnings on missing `return` statements.

5 The Results

The size of the code bases in SATE differed greatly. Dovecot is 360 kLoC in size, Wireshark 1.2.0 and 1.2.9 are both 1.7 MLoC, and Chrome 5.0.375.54 and 5.0.375.70 are 1.5MLoC and 1.7MLoC, respectively. The number of warnings Goanna issued range from 180 for Dovecot, 534 and 532 for the Wireshark code bases, to 1079 and 1173 for the Chrome code bases, respectively. But the result do not only differ in the number of warnings found, but also in the type of warnings. Table 1 list the top ten checks by the number of warnings for Dovecot.

Most of the Dovecot warnings relate to potential NULL pointer dereferences and to trivial conditions. A class of warning that only occurred for the Dovecot code base are the warnings on the correct use of the semantic attributes `const` and `pure`. These attributes are a GNU language extension, that can be used in compiler optimization, however the gnu compiler does not check for proper use of these attributes. This might explain the fairly high number of warnings of this type; the programmer is expected to use these attributes correctly, without being held to a correct use the compiler.

#	**Check**	
1	Dereference of possible NULL pointer	59
2	Comparison never holds	31
3	Comparison always holds	21
4	Uninitialized variable on some paths	15
5	Global variable access by `const` function	12
6	Dereference of possible NULL pointer by function	11
7	Call of function w/o `pure` by function with `pure` attribute	10
8	Unused variable on all paths	7
9	Uninitialized struct field	4
10	Store stack in a global	2

Table 1: The 10 checks with the most warnings for Dovecot, ranked by the number of warnings.

#	**Check**	**1.2.0**	**1.2.9**
1	Comparison never holds	281	282
2	Unused pointer value	54	55
3	NULL check after dereference	54	50
4	Uninitialized variable on some paths	35	35
5	Dereference of possible NULL pointer	35	34
6	Comparison always holds	33	35
7	Uninitialized struct field	10	10
8	Dereference of possible NULL pointer by function	8	8
9	Variable used in divisor before comparison with 0	6	6
10	Parameter checked before deref only on some paths	4	4

Table 2: The 10 checks with the most warnings for Wireshark 1.2.0 and 1.2.9, ranked by the number of warnings.

The Wireshark code bases does not use this language extension, and this type of warnings is therefore absent. The majority of warnings concern potential NULL pointer, uninitialized variables and trivial conditions. One check with remarkably many warnings is the check for unused pointer values. This check will warn if a pointer is assigned a value that is not NULL, and then not used along any path. While the warnings can be insignificant, they do quite often uncover serious issues, if, for example, by a copy an paste error, the wrong pointer was used subsequently.

The number of warnings changed slightly from version 1.2.0 to 1.2.9, but not significantly. A warning that disappeared completely in revision 1.2.9 was a warning about the potential dereference of a definite NULL pointer. There were only two warning of this particular check for Wireshark 1.2.0, both caused by the incorrect used of a macro `#define MATCH ((class == info->tclass) && (tag == info->tag))`. It was used when `info` was possibly NULL. In Wireshark 1.2.9 this macro was replaced by `#define MATCH (info && (class == info->tclass) && (tag == info->tag))`, i.e. it included a NULL check before the rest of the expression got evaluated. This effectively addressed the warning.

Table 3 lists the most common warning for Chrome. Also for this code bases most warning were concerned with trivial conditions, uninitialized , and potential null pointers. For Chrome there are significant changes between version 5.0.375.54 and 5.0.375.70, but this reflects that the later version also grew 11% in size. The next section will discuss a few warning that have been evaluated by NIST in detail.

6 Warnings Explained

NIST selected as a part of SATE for each tool 30 warnings from the set of Dovecot warnings for evaluation. In addition NIST selected CVEs from Wireshark, that is known bugs the list of Common Vulnerabilities and Exposures maintained by MITRE. In this section we will discuss a few of these warnings and defects, since they illustrate nicely the type of analyzes Goanna performs to detect potential defects.

The following warnings from the Dovecot code base deals with the correct use of semantic attributes, a checks that requires pattern matching only.

```
unichar.c:193: warning: Goanna[SEM-const-call]
Non-const function 'uint16_find' is called in
const function 'uni_ucs4_to_titlecase'
```

```
    unichar_t uni_ucs4_to_titlecase(unichar_t
        chr)
    { [...]
193   if (!uint16_find(titlecase16_keys,
          N_ELEMENTS(titlecase16_keys), chr, &
              idx))
            return chr;
          else [...]
```

The detected issue results from a wrong use of GNU semantic attribute `const`. These allow the use to define attributes of functions, which can then be used by the compiler to optimize code. In the above example function `uni_ucs4_to_titlecase` has the attribute `const`. The documentation on the GNU language extension says that "a function that calls a non-const function usually must not be const". This requirement is violated in the above example, since function `uint16_find` does not have the attribute `const`. To find this type of violation it is sufficient to check if all functions that are a

#	Check	x.54	x.70
1	Dereference of possible NULL pointer	400	424
2	Comparison always holds	166	193
3	NULL check after dereference	110	112
4	Comparison never holds	92	102
5	Dereference of possible NULL pointer by function	54	63
6	Uninitialized variable on some paths	49	55
7	Paramter checked before derefence only on some paths	42	43
8	Unused variable on all paths	29	23
9	Switch case unreachable	21	21
10	Uninitialized variable on all paths	14	25

Table 3: The 10 checks with the most warnings for Wireshark 1.2.0 and 1.2.9, ranked by the number of warnings.

called in a function with attribute `const`, have this attribute themselves. This can be achieved by combining two patterns on the AST, one to find function calls in `const` functions, and one to check the attribute of the called functions.

The following example, also from Dovecot, was found with a combination of summaries and abstract data tracking.

```
director-connection.c:655: warning: Goanna[RED-
cmp-never] Comparison never holds
```

```
655 if (str_array_length(args) != 2 ||
        director_args_parse_ip_port(...) < 0) {
          i_error( );
          return FALSE;
    }
```

The above code fragment is part of an error handler; the error routine `i_error` will be called if the output of `director_args_parse_ip_port` is negative. However, the output range of this function is in the interval $[0, 1]$ and will thus never be negative. Closer inspection showed that this code was refactored to return 0 if an error is detected rather than -1. Everywhere else in the code, except for a few exceptions, the output of this function was treated like a Boolean. The mistake presumably entered since manual refactoring did not change all occurring test in the error handler to use a Boolean condition. Abstract data tracking was used to determine the output range of `director_args_parse_ip_port`. This range was then part of the whole-program summary and be used to detect unsatisfiable conditions in other functions.

6.1 Denial of Service

The following example, from the Wireshark code base, highlights a potential denial of service exploit. Goanna required its model checking capabilities to detect this security issue. The surrounding code is:

```
packet-smb.c:8211: warning: Goanna[PTR-param-
unchk-some] Parameter 'nti' is not checked
against NULL before it is dereferenced on some
paths, but on other paths it is
```

```
     case NT_TRANS_IOCTL: [...]
8211   dissect_smb2_ioctl_data(ioctl_tvb, pinfo,
           tree, top_tree, nti->ioctl_function,
           TRUE);
       [...]
     case NT_TRANS_SSD:
       if(nti){switch(nti->fid_type){ [...]
```

The detected issue resides in a longer switch statement. In one case `nti` is not checked to be not null before it is dereferenced, in the other case it is. This points to an inconsistency, which can lead to a null-pointer dereference. This particular example was reported as CVE-2010-2283 in the MITRE bug database. Finding this bug requires to compare behavior along two paths. In this example, the inconsistency happens within a few lines. In general inconsistent paths may contain conditional jumps and loops, such that an explicit path enumeration becomes infeasible. Model checking provides algorithms to checks this exhaustively and efficiently.

Unfortunately, this warning was not included in the Goanna report for Wireshark. In Goanna it is possible to set a timeout, after which the tool stop with the analysis of a file, and proceeds with the following file. For the SATE participation the timeout was set to 120 seconds, which in hindsight seems to be rather low. In this particular case only half of the file was analyzed, the function in line 8211 however was not. Increasing the timeout to a more reasonable five minutes would have revealed this bug.

6.2 Potential Program Crash

The final warning we like to discuss is also taken from the Dovecot code base. Goanna used model checking and interprocedural whole-program analysis to detect this defect.

```
director.c 180: warning: Goanna[PTR-null-assign
-fun-pos] Dereference of 'preferred_host' which
may be NULL
```

```
     *preferred_host =
         director_get_preferred_right_host(dir);
                   [...]
     if (cur_host != preferred_host)
180    (void)director_connect_host(dir,
           preferred_host);
     else {[...]}
```

The detected issues is based on the fact that the function `director_get_preferred_right_host` may return a NULL pointer and assign it to `*preferred_host`. This is later used as parameter of `director_connect_host`. However, inter-procedural analysis shows that there exists a path in `director_connect_host` where this parameter will be dereferenced, without a prior check for being potentially NULL. The dereferencing of this NULL pointer can lead to an exception/crash that enables an attacker to potentially enter the system.

7 Summary and Future Directions

This report described the entry and results of the static analysis tool Goanna by Red Lizard Software for the Static Analysis Tool Exposition, organized by NIST. Goanna is a novel type of tool that combines static analysis techniques such as pattern matching, data flow analysis and abstract data tracking with model checking to obtain a fast, scalable and precise solution to detect potential software defects. For SATE we selected the default checks of Goanna, and applied them to the five predetermined code bases.

The results show that Goanna is a competitive solution to find serious software defects in real life code. At the time of writing the final results of the NIST evaluation is unknown, and it would be premature for us to make a final comment. Intermediate results that we shared among participants, however, confirm that Goanna is at least on par with the other leading tools in this area when it comes to the fraction of serious and quality issues detected, versus the fraction of insignificant and false warnings.

The intermediate feedback also gave us valuable feedback on how to further improve the tool. The feedback was in particular useful, since the manual evaluation of NIST went throught the effort to describe the potential issue in detail. Cause for false positives were omitted corner cases, unused semantic information, incomplete semantic models, non-trivial invariants, and custom asserts. The first two causes can be directly addressed by refining the existing checks. Given that Goanna defines the checks separately from the analysis engine, this is an easy improvement. This type of issue that were revealed by the NIST evaluation have been addressed in the meanwhile.

Addressing an incomplete semantic model requires to improve the basic semantic models are already used by the tool; all that is required to refine these models. Non trivial invariants is a more fundamental problem of static analysis. For example, detecting one of the CVEs selected by NIST, CVE-2010-2286, would require to prove the absence of an appropriate loop invariant, for a loop that was realized by gotos that span approximately 2000 lines of code. This type of analysis is arguably outside of the scope of static analysis tools. Even more powerful tools, such as automatic theorem provers would have a hard time detecting such issues, besides the problem that these solution will currently not scale to problems of this size. The final problem of custom asserts can be addressed by giving the user a way to redefine or refine checks, or give by giving them the means to add annotations or pragmas. However, from among users there exists some reluctancy to these solutions, since they require to change the code to accommodate the analysis. Solving this problem is mostly a usability issue.

Red Lizard Software participated in 2010 for the first time in SATE. The challenge for the tools was to deal with code bases of different sizes and of different type. The evaluation by NIST focused mostly on the quality of the warnings, rather than on the speed of a solution or its ease of use. It helped to shed a light on the strength and weaknesses of the tool, and confirm that Goanna is a competitive solution for C/C++ analysis.

References

[BFK02] Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper, *Structural properties of xpath fragments*, ICDT '03: Proceedings of the 9th International Conference on Database Theory (London, UK), Springer, 2002, pp. 79–95.

[CE82] Edmund M. Clarke and E. Allen Emerson, *Design and synthesis of synchronization skeletons for branching time temporal logic*, Logics of Programs Workshop, LNCS, vol. 131, Springer, 1982, pp. 52–71.

[Cou81] P. Cousot, *Semantic foundations of program analysis*, Program Flow Analysis: Theory and

Applications, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981, pp. 303–342.

[FHJ$^+$07] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch, *Model checking software at compile time*, Proc. TASE 2007, IEEE Computer Society, 2007.

[Pnu77] Amir Pnueli, *The temporal logic of programs*, Proceedings of the 18th Annual Symposium on Foundations of Computer Science (Washington, DC, USA), IEEE Computer Society, 1977, pp. 46–57.

[QS82] Jean-Pierre Queille and Joseph Sifakis, *Specification and verification of concurrent systems in CESAR*, Proc. Intl. Symposium on Programming, Turin, April 6–8, 1982, Springer, 1982, pp. 337–350.