Appears in: Proceedings of the 1st IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering, June 6–8, 2007, Shanghai, China.

Model Checking Software at Compile Time

Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch
National ICT Australia Ltd.
University of New South Wales
Locked Bag 6016, Sydney NSW 1466, Australia
firstname.lastname@nicta.com.au

Abstract

Software has been under scrutiny by the verification community from various angles in the recent past. There are two major algorithmic approaches to ensure the correctness of and to eliminate bugs from such systems: software model checking and static analysis. Those approaches are typically complementary. In this paper we use a model checking approach to solve static analysis problems. This not only avoids the scalability and abstraction issues typically associated with model checking, it allows for specifying new properties in a concise and elegant way, scales well to large code bases, and the built-in optimizations of modern model checkers enable scalability also in terms of numbers of properties to be checked. In particular, we present Goanna, *the first C/C++ static source code analyzer using the off-the-shelf model checker NuSMV, and we demonstrate Goanna's suitability for developer machines by evaluating its run-time performance, memory consumption and scalability using the source code of OpenSSL as a test bed.*

1 Introduction

The application of formal verification techniques to software is hard. While full functional correctness can be shown by proof-based methods such as interactive theorem-proving, the effort is high—i.e., there is substantial expert manpower needed over an extended period of time. This is not always practical. Drivers, for instance, have typically short development times as they need to be supplied in time with the hardware release. Also, ongoing code changes have to be taken care of, creating the demand for automated analysis tools working at compilation time.

Model checking [8, 25] and static analysis [21, 23] are automated techniques promising to ensure (limited) correctness or to find bugs in software. Software model checking typically operates on the semantic level of a program. The common approach is to find a finite state abstraction and to model check this abstraction. If the abstraction is too coarse it will be further refined. Finding the right level of abstraction is challenging and the subject of much research. Without user interaction, software model checking approaches are often not mature enough yet to cope with real life code efficiently [26, 22].

Static analysis, on the other hand, works on the syntactic level of the program. As such, any finite program and its control flow graph results in a finite state system and is, therefore, suitable for algorithmic analysis. While static analysis is known to scale well to large code bases, it is limited by the number of properties to be checked and the definition of new properties is often cumbersome [13]. In contrast, model checking allows for a convenient and often concise specification of program properties and optimizations in the checker technology makes it less affected by the number of properties checked within the same model.

In this work we present an approach that combines the best of both worlds by using the off-the-shelf model checker NuSMV [6] and its specification language to define and check static analysis type properties on large C/C++ programs. This allows for a concise and flexible specification of properties and an analysis scalable both in the size of the code base as well as the number of properties analyzed.

Contribution. We demonstrate that model checking is a practical and scalable solution to solve static analysis problems. We demonstrate the practicality by presenting a method of encoding C/C++ program analysis as model checking problems for the NuSMV model checker. We implement the proposed encoding in our prototype tool *Goanna* and present its architecture. Moreover, we demonstrate that Goanna is competitive with respect to run-time performance, memory consumption and scalability.

The ability to directly use all the optimizations built into modern model checkers, automatically obtain a counterexample trace in case of a property violation, and add more semantic-based software model checking techniques in the

future makes the proposed approach a viable alternative to existing technology.

The remainder of this paper is organized as follows: In Section 2 we discuss a number of related approaches in this area. In Section 3 we give a presentation of our approach and illustrate this by an extended example in Section 4. We give a detailed runtime evaluation of our approach by checking the source code of OpenSSL and present a preliminary evaluation on precision in Section 5. In Section 6, we discuss current limitations of our tool, ideas for future work and our conclusions.

2 Related Work

The basic ideas of solving static analysis problems by model checking were first developed by Steffen and Schmidt [28, 27]. Their main focus is on developing a safe approximation of the program's behaviour and, therefore, checking for a safe subset of CTL, i.e., *qualified safety* properties. The drawbacks of this approach are that safe approximations of real C/C++ programs including pointer arithmetic are either hard to compute or too coarse, leading to unnecessary over-approximations.

We have a stronger focus on the effectiveness of the analysis and abandon in some cases soundness as defined by Steffen. This means, we treat programs purely as a set of syntactic objects on the program's CFG and allow to check any CTL property on that level. While our analysis is sound on this syntactic level it is not necessarily a sound abstraction of the program's semantics. However, this approach has been followed by others (e.g.,[12, 11]) and proves to be well-suited for checking real-life systems.

A similar approach to ours can be found in the Uno tool [17] and its later development into Orion [11]. The analysis is also done by model checking on a syntactic level. However, the authors do not use an off-the-shelf model checker, but implement model checking techniques. Orion is currently more limited to checking for three properties: uninitialised variables, nil-pointer dereferences and out-of-bounds array indexing. The tool currently has a strong focus on achieving a good signal to noise ratio by incorporating symbolic solver techniques. Goanna focuses on a wider range of properties, with future plans to include user-defined rules and embedded assembly. It will be interesting to compare future versions of both tools.

Related to our philosophy is, e.g., the work in the static analysis community done by Engler et al. [12]. The authors use meta-level compilation (MC) which allows system implementors to build their own application-specific compiler extensions based on the *Metal* language. Those extensions are used as specifications for searching the abstract syntax tree, control flow and data flow graph. The approach has been further developed into a commercial product [10].

There are other commercial static analysis tools, e.g. [14, 19, 15, 20] which, however, mostly do not support specification languages such as Metal or CTL. This limits their applicability for system development.

A semantic model checking approach to software verification is realised in SLAM [1] and its successor SDV, a tool used to verify device drivers. SLAM is a suite of tools for counterexample-guided abstraction refinement. SLAM starts with a coarse Boolean program abstraction that is subsequently refined given predicates discovered from counterexamples in the abstraction, until an abstraction is found that satisfies the property. Other tools that implement counterexample-guided abstraction refinement are Blast [16] and Magic [4].

A tool for bounded model checking of ANSI-C code was presented in [7]. This tool, called CMBC, can be used to verify safety properties, and also to verify an ANSI-C model of a circuit against a specification in a hardware description language such as Verilog. The tool unrolls the program and checks with a SAT-solver if there exists an error trace up to the given depth. CMBC is particularly useful for debugging since it can find all errors up to a certain depth quickly.

The Eau Claire tool [5] makes use of automatic theorem proving. It is an extended static checker for C, based on the earlier ideas in [18]. The tool translates C code into a set of guarded commands which will then be transformed into verification conditions. These verification conditions are checked automatically by the Simplify theorem prover.

Based on abstract interpretation [9] are the PolySpace [24] and Astrée [3] static analyzers. They aim at proving the absence of run-time errors in programs written in the C/C++ programming languages. Astrée analyzes structured C programs, without dynamic memory allocation or recursion. Abstract interpretation is particularly well suited for array bound checking and alike, as it provides a semantic framework to capture domains and the operations on them, but suffers from high computational costs resulting in much longer analysis times.

3 Syntactic Software Model Checking

In this section we present details of how to encode static analysis properties by model checking in a practical way. The goal is to determine syntactic properties of C/C++ programs ranging from uninitialized variables to null pointer dereferences. Given a program P and a property ϕ the task of checking whether P satisfies ϕ, i.e., $P \models \phi$, is reduced to checking $P_s \models \phi_s$ where P_s is a finite syntactic representation of P and ϕ_s a syntactic encoding of ϕ.

Although we use model checking for our analysis, the type of properties we are addressing are similar to those in static analysis. For instance, we check whether a variable v

is initialized, but not, e.g., whether v holds the correct computation result by the end of the program execution. The latter often requires more information about the program's semantics.

As model checking is the analysis of a labeled graph (a Kripke structure over atomic propositions) with respect to some formula, typically in temporal logic, we have to develop:

1. A temporal logic formula expressing the desired property.

2. A graph representing the C/C++ program, labeled with propositions relevant to evaluate the property.

3. A translation to a model checker.

In the following we describe our approach to a path-sensitive, intra-procedural analysis in the order of the different steps above. Throughout the remainder of this paper we will use absence of uninitialized variables as an example property, because it is simple in comparison to other properties that we check, but serves well to demonstrate our approach.

3.1 Temporal Logic Properties over Programs.

Under the assumption that we can identify atomic propositions $decl_x$, $used_x$ and $assigned_x$, representing program locations where a variable x is declared, used, or assigned a value respectively, we can specify that this variable is always initialized before it is used as follows in CTL:

$$\text{AG } decl_x \Rightarrow (\text{A } \neg used_x \text{ W } assigned_x) \qquad (1)$$

This means we require that on all program paths if a variable x is declared it must not be used until it has a value assigned or it will not be used at all. We use the *weak until* operator W here to include the second possibility. The latter can also point to unused variables, which is checked separately.

In the same style we can express other properties on correct pointer handling, variable usage or memory allocation and deallocation. Moreover, it allows specifying application specific properties to handle general programming guidelines, API-specific rules or even hardware/software interface rules for drivers.

In the remainder of this section we describe how to map programs to transition systems labeled with atomic propositions such as the ones above and how to derive the labels themselves from a program.

3.2 Model Construction

To construct a model from the program, we require a formal notion of an abstract syntax tree (AST) and a labeled graph. We define a labeled graph/tree and an attributed tree as follows.

Definition *A* labeled graph (L, E, μ_L) *over the alphabet* Σ_L *is a finite and directed graph, where* L *is a set of nodes,* $E \subseteq L \times L$ *is an edge relation between the nodes, and* $\mu_L : L \to \Sigma_L$ *is a node labeling function.*

A labeled tree *is a labeled graph* $T = (L, E, \mu_L)$ *if it has a single root node* $root(T)$ *for which we have the following: For each node* $l \in L$ *there exists exactly one path from the root to the node, i.e. exactly one sequence* $l_0, \ldots, l_n$*, such that* $l_0 = root(T)$*,* $l_n = l$*, and* $(l_{i-1}, l_i) \in E$*, for* $i = 1, \ldots, n$*.*

An attributed tree (L, E, μ_L, μ_E) *over the alphabets* Σ_L *and* Σ_E *is a labeled tree where there is an additional labeling function* $\mu_A : L \to \Sigma_A$*, assigning* attributes *to nodes.*

Given two nodes l_1 and l_2 of a labeled tree (L, E, μ_L), we call l_1 the *parent* of l_2 and l_2 the *child* if $(l_1, l_2) \in E$. If there exists a (non-empty) path from l_1 to l_2, l_1 is called *ancestor* of l_2, and l_2 is a *descendant* of l_1.

An AST can be seen as an attributed tree where the nodes are labeled with program statements and (sub)expressions while the attributes describe the role of a node's branch. E.g., the construct in Figure 3(a) shows part of an AST for an if-then branching. The attributes describe whether the subtree is the if-branch, the condition, or the next instruction of its parent node. The labels then describe the kind of statement or expression of the condition or the statement following the if-then.

From the AST we can construct in a straightforward manner the control flow graph (CFG). Note that a CFG does typically not contain all the information available in the AST, only the control structure down to the level of statements, but not the structure of expressions, types, and constant values.

A CFG is a graph, typically with a single root node. Later we add labels making it a labeled graph. We denote the labeled CFG of an AST T by CFG_T. The labels represent whether certain atomic propositions hold in a node. E.g., if a particular variable is assigned a value, if it is used on the right-hand side of an assignment, or if it is dereferenced and so on.

In our framework, these labels are associated with tree patterns on the AST. We define the syntax of a query language to match tree patterns as follows:

$$\begin{aligned} P &::= \epsilon \mid \emptyset \mid \sigma_E \mid \sigma_L \mid \downarrow \mid \downarrow^* \mid P/P \mid P \cup P \mid P[Q] \\ Q &::= P \mid label = \sigma_L \mid attr = \sigma_A \mid Q \wedge Q \mid Q \vee Q \end{aligned}$$

where $\sigma_L \in \Sigma_L$ and $\sigma_A \in \Sigma_A$.

Given an (attributed) tree T and a node l, a pattern defines a set of nodes in the subtree rooted in l. The pattern ϵ defines the node l itself, $\emptyset$ the empty set, and pattern σ_A and σ_L children labeled σ_A or σ_L respectively. The patterns $\downarrow$ and $\downarrow^*$ stand for the children and descendants of l, $/$ and $\cup$ for concatenation and union. Finally, pattern $P[Q]$ filters all nodes satisfying Q.

This tree query language is the downward, recursive fragment of the language defined in [2]. We refer the reader to this paper for formal semantics and a discussion on expressiveness. The only difference is that we allow for two types of labels, which however, does not add expressivity.

Given an atomic proposition p, we associate a tree pattern P with it. We label every node l that matches P, in the AST T with respect to the root node of T, with p. In the case that l is not in CFG_T, we label its closest ancestor in T which is in CFG_T.

Example 1 Take as an example an atomic proposition $decl_x$, used to label declarations of variable x. This proposition is associated with pattern $\downarrow^*$ $Decl[Var{:}x]$, i.e. it matches all nodes (descendants of the root node) in the AST labeled $Decl$, that have a child labeled *Var:x*. Those nodes will then be labeled with $decl_x$ in the CFG.

3.3 Translation to NuSMV

In order to check our generated model automatically with respect to the defined properties, we developed a translation to the NuSMV model checker. In this section we sketch how to translate a labeled CFG and a CTL formula into a simple NuSMV model.

For a given C/C++ function f we translate the corresponding labeled CFG (L_f, E_f, μ_f) into a *simple NuSMV model*, $NuSMV_f = (var_f, \Delta_f, Def_f, CTL_f)$, where

- var_f is *one* enumerated type variable in NuSMV over the set of types $l \in L_f$. Enumerated type variables are implemented efficiently in NuSMV and guarantee a much smaller state space than, e.g., using one boolean variable for each control location.
- $\Delta_f = \{(l, Succ(l)) | l \in L_f, Succ(l) = \{l' | (l, l') \in E_f\}\}$. This means, the CFG transition relation is translated into a transition relation, where the target for each transition is the set of locations we possibly can branch to. This is according to NuSMV's syntax and does not change the original CFG transition relation.
- $Def_f = \{define(p) = \{l | \mu_f(l) = p, l \in L_f\} | p \in \Sigma_f\}$. Where every $define(p)$ is a `DEFINE` declaration of p in NuSMV. A define declaration is a space efficient way to declare, e.g., that a propositional variable p holds exactly in a particular set of locations. In our case, that a label is evaluated to true iff it has been a label in the original labeled CFG.
- CTL_f is the set of CTL properties in NuSMV syntax.

In the subsequent Section 4 we will give an example of the actual NuSMV code which is a syntactic expression of the above model.

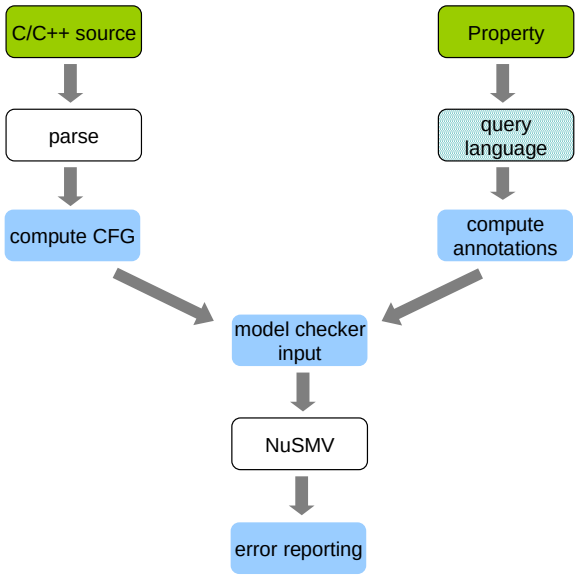

Figure 1. Model checking approach for statically analysing C/C++ code.

3.4 Architecture

The architecture of our approach is outlined in Figure 1. Given a C/C++ program, the only interaction needed from the user is to

1. provide a CTL specification, and
2. define the atomic proposition of the specification in terms of queries as described in Section 3.2.

The translation of a program into the CFG, the pattern matching, the subsequent labeling, the translation to NuSMV, as well as the error reporting, are all fully automatic. This reduces the burden on the user to a minimum and for generic pre-defined properties to zero.

4 Example

This section presents an example to illustrate the proposed approach of combining syntactic checking with

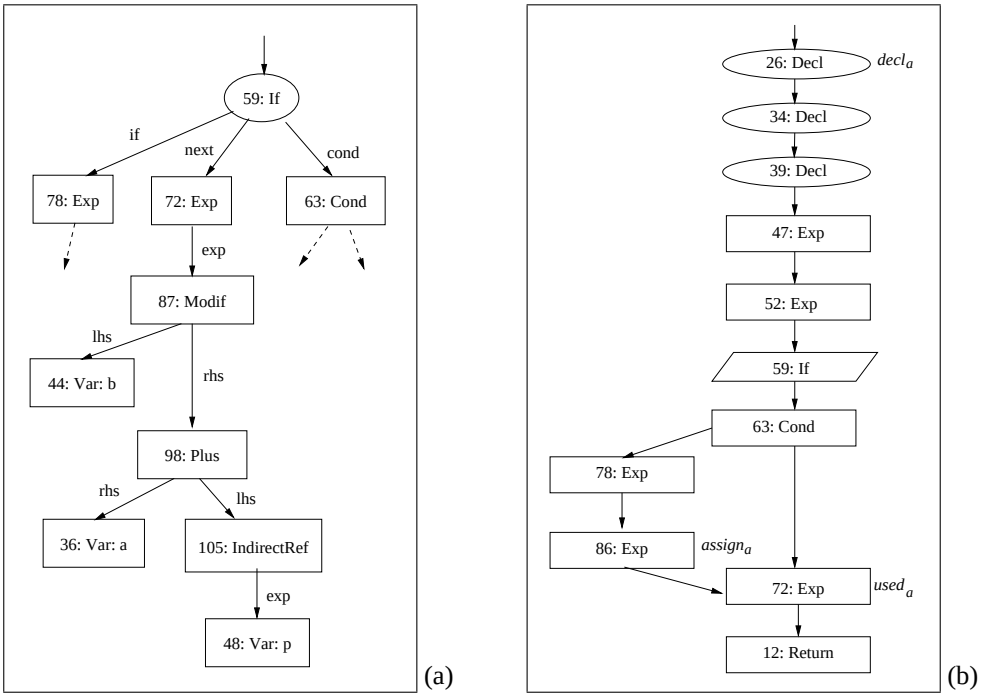

Figure 3. Fragments of (a) the attributed abstract syntax tree (AST), and (b) the annotated control flow graph (CFG).

```
1    void f(int x) {
2      int a, b;
3      int *p;
4
5      p = (int *)
           malloc(sizeof(int));
6      *p = 42;
7      if(x == 0) {
8        free(p);
9        a = *p;
10     }
11     b = a + *p;
12   }
```

Figure 2. Source code example

model checking. Consider the contrived code fragment in Figure 2, which is obviously flawed. Not only does it contain a potential access of an uninitialized variable (`a` in line 11), but also dereferences a pointer that has already been freed (`p` in line 9), and assigns a value to a variable that is never used afterwards (`b` in line 11).

We will illustrate the labeling of the AST and CFG, the generation of NuSMV code that implements the checks and the ease of adding properties to Goanna. Again, we demonstrate it in the context of uninitialized variables for the sake of simplicity.

4.1 Annotation of CFG

The program analysis builds on an annotated control flow graph. Generating the control flow graph—without annotations—for the code fragment is straightforward. A fragment of the AST is depicted in Figure 3 (a) and the resulting CFG in Figure 3 (b). Each node is labeled with an ID and a label. The IDs refer to identifiers in the intermediate format generated by the parser and are used for technical reasons only. The attributes of nodes in the AST are used to label the edge with the parent node in Figure 3 (a). Only nodes on certain levels in the AST will be used to build the CFG, in this example nodes 59, 63, 72 and 78.

The atomic propositions that are used as labels on the CFG are generated, as mentioned before, in two steps. In the case of uninitialized variable analysis, the procedure introduces three labels for each variable. E.g., the declaration of variable `a` in line 2 of the code fragment, for example, leads to the introduction of the labels: $decl_a$, $used_a$ and $assigned_a$. Each of these labels will be an atomic proposition in the model checking framework.

Next, we describe the associated tree patterns for each proposition. Proposition $decl_a$, used to label declarations of variable a, is associated with pattern $\downarrow^* Decl[$*Var:a*$]$. Only node 26 in the CFG matches this pattern, corresponding to the declaration in line 2 of the source code, and will be labelled $decl_a$.

Proposition $used_a$ will label nodes that use the variable a. These are nodes which are either (i) labelled *Plus*, *Postinc* or *Preinc*, and have a child labelled `Var:a`, or (ii) nodes labeled *Modif* with a righthand-side child *Var:a*. The corresponding pattern is $\downarrow^*$ (*Plus* $\cup$ *Postinc* $\cup$ *Preinc* $\cup$ $\ldots$)[*Var:a*] $\bigcup \downarrow^*$ *Modif*[*Var:a*$[attr = "rhs"]$]. Node 98 in Figure 3 (a) matches this pattern. However, this node is not part of the CFG. Hence, we backtrack to the nearest ancestor which is part of the CFG, in this case node 72, and label it $used_a$. Node 72 corresponds to line 11 in the code fragment.

Finally, for proposition $assigned_a$ we look for nodes that modify a variable which have labels such as *Modif*, *Postinc* or *Preinc* and a left-hand-side successor *Var:a*. This corresponds to the pattern $\downarrow^*$ (*Modif* $\cup$ *Postinc* $\cup$ *Preinc*)[*Var:a*$[attr = "lhs"]$]. Node 86 will be labeled $assigned_a$, because it is the nearest ancestor of a node that matches this pattern. The described process results in the labelled CFG depicted in Figure 3 (b).

4.2 Model Checking with NuSMV

Parts of the NuSMV code resulting from the translation of the labelled CFG are shown in Figure 4. As described in Section 3.3 we introduce one enumerated type variable, i.e., `location`, ranging over the control locations (i.e. the nodes in the CFG), describe the transition relation as a set of transitions from locations to a set of locations, and use `DEFINE` declarations to associate labels to certain locations. Note that, for clarity of presentation, we use the weak until operand `W` in the CTL specification, which does not exist in NuSMV syntax, but can be equally expressed through other existing operators.

Using NuSMV for checking property (1) (described in Section 3.1) on the annotated CFG now reveals a violation of the NuSMV specification from Figure 4. Goanna automatically examines the violation reported by NuSMV and concludes that there is an incorrect use of variable `a`—in fact, there is a path in the program

```
MODULE main

VAR location : {loc26, loc34,..., loc12}

next(location) :=
 case
    location = loc26 : {loc34};
    location = loc34 : {loc39};
    ...
    location = loc63 : {loc78, loc72};
    ...
 esac

DEFINE
    decl_a := location in {loc26};
    used_a := location in {loc86};
    assign_a := location in {loc72};

SPEC AG decl_a => (A ~used_a W assign_a)
```

Figure 4. NuSMV code (fragment)

on which `a` is used, but not assigned any value beforehand. The prototype tool Goanna will warn the user with: "`Warning: Variable 'a' might be uninitialized`" and automatically produces a counterexample with the sequence of line numbers 2, 3, 5, 6, 7, 11. Note that this analysis points to a *potential* bug. There are other properties that, if violated, point to definite bugs.

4.3 Property Implementation in Goanna

In order to implement such a property in our Goanna tool, we only need to identify the nodes of interest (as described in Section 4.1), describe their relationships in CTL formulas and print these formulas in NuSMV syntax to NuSMV's input file (as described in Subsection 4.2). An implementation of our example property is shown in pseudo code in Figure 5. In the code, we first find the set of all variables that are declared in a function, because these are the ones that need to be checked for proper initialization. For each of these variables, we then print the CTL specifications to the NuSMV input file (the function `NuSMV()` simply prints text to NuSMV's input file). In the next step, for every variable in the set of declared variables, we search for the set of nodes in the AST where the variable is assigned a value and where it is used, respectively. For this search we use the function `FindAST()` that identifies nodes in the AST according to specific patterns. Finally, from these three sets, we get the corresponding sets of locations of the respective nodes in the CFG by using the function `CFG_locations()`. That function translates

```
# Find all variables declared in a function:
Set_varDecl = FindAST(type=var, label=decl);
# Print specification for each declared variable:
foreach $var in Set_varDecl
{
  NuSMV( "SPEC AG decl_$var ->
          (A !used_$var W assign_$var);" );
}
NuSMV( "DEFINE" );
foreach $var in Set_varDecl {
  # Identify nodes that use or assign $var:
  Set_varAssigned = FindAST(var=$var, attr=lhs,
      label=Modif|Postinc|Preinc|...);
  Set_varUsed = FindAST(var=$var, attr=rhs,
      label=Modif)
    + FindAST(var=$var, label=Postinc|Preinc);
  # Output locations of propositions:
  NuSMV( "decl_$var := location in {"
    + CFG_locations( Set_varDecl ) + "};" );
  NuSMV("assigned_$var := location in {"
    + CFG_locations( Set_varAssigned ) + "};" );
  NuSMV("used_$var := location in {"
    + CFG_locations( Set_varUsed ) + "};" );
}
```

Figure 5. Pseudo code to create a NuSMV code fragment for the "uninitialized variables" property

AST nodes to CFG nodes and, if required, backtracks to the nearest ancestor node that has a corresponding node in the CFG. Then we print the locations to NuSMV's input file.

Note that the code does *not* have to describe *how* the check is implemented. In Goanna, we only describe the checks that need to be done—the implementation of the checks is left to NuSMV.

This might seem like heavy machinery to check for uninitialized variables. However, during analysis this property is checked for all variables at once. Furthermore, having this framework in place enables us to define other syntactic properties easily—e.g., properties for inappropriate use of dereferenced pointers, or unused variables. Applying Goanna to the source code example shown in Figure 2 will warn that the value assigned to variable b in line 11 is never used, that variable b is never used at all (on a right-hand side), that variable a might be used uninitialized, and that pointer p is dereferenced after being freed.

5 Evaluation

5.1 Implementation

Our implementation is written in OCaml and we use NuSMV 2.3.1 as the back-end model checker. The current implementation is a prototype working on intra-procedural analysis and is not yet optimized towards performance. However, it provides the reader with a first impression with respect to speed and scalability.

At the current stage, we have 15 different checks implemented. These checks cover the correct usage of malloc/free operations, use and initialization of variables, potential null-pointer dereferences, memory leaks and dead code. The CTL property is typically one to two lines in the program and the description query for each atomic proposition is around five lines when covering a lot of exceptions. This is still rather short compared to standard static analysis frameworks or meta compilation [12].

5.2 Evaluation Principles

We evaluate our approach regarding run-time performance, memory usage, and scalability. The run-time of the tool should be reasonably low and integrate well into the development process, e.g. it should be within the same order of magnitude as the compile time. The runtime should scale well with increasing program size and number of properties. Also, the memory usage must be within the resource limits of a typical developer machine or build server.

To evaluate Goanna's speed and memory usage, we apply it to a larger real-world open-source project of realistic size: The *OpenSSL*[1] toolkit implementing the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols (260 kLoC). We build OpenSSL for a Debian Linux 3.1 environment with gcc 3.3.5. Our hardware platform for the experiments is a DELL PowerEdge SC1425 server, with an Intel Xeon processor running at 3.4 GHz, 2 MiB L2 cache and 1.5 GiB DDR-2 400 MHz ECC memory.

5.3 Run-time Performance and Memory Usage

First, we look at some overall performance numbers. We measure the wall-clock compile time and compare it with Goanna's total analysis time. Furthermore, we measure the maximum internal memory consumption of our tool as well as NuSMV's memory consumption during the analysis.

The results for one property and for all 15 properties are shown in Table 1. It shows that the overall analysis time is well within the same order of magnitude as the compile

[1] http://www.openssl.org/

Test set	Compile	Goanna		NuSMV		Total (Goanna + NuSMV)	
	[min:sec]	[min:sec]	[MB]	[min:sec]	[MB]	[min:sec]	[MB]
OpenSSL (1 property)	3:07	2:58	23.5	3:07	11.8	6:05	35.3
OpenSSL (15 properties)	3:07	6:19	35.4	5:54	17.9	12:13	53.3

Table 1. Compile time, run-time and maximum memory usage for Goanna and NuSMV separately, and for the whole tool chain in total.

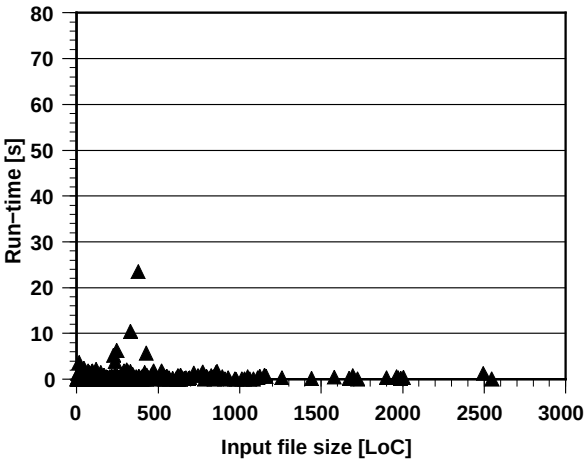

Figure 6. Run-times of NuSMV with respect to size of input source files.

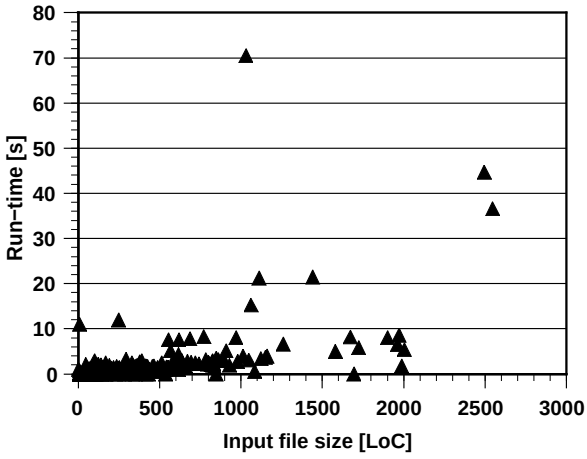

Figure 7. Run-times of the whole Goanna tool chain with respect to size of input source files.

time. In fact, it is twice as long as the compilation for one property and four times as long for 15 properties.

Moreover, for the analysis with all 15 properties, out of 602 source files, only 3.6% took longer than 2 seconds to analyze and 99.2% of all files were analyzed in less than 5 seconds. The time spent in NuSMV is mostly negligible with 98.7% of all files being analyzed in less then 2 seconds.

The overall distribution of the runtime with respect to the file size is shown in Figure 6 for NuSMV and in Figure 7 for the overall analysis time. Note that the complexity of the analysis—and hence its runtime—does not perfectly correlate with the file size, but the file size is easily understandable and typically a measure of interest to the developer. In fact, the complexity of our current implementation is mostly dependent on the number of variables and the size of the CFG.

The memory consumption for one as well as for 15 properties has been considerably low with 35.3 and 53.3 MB, respectively. This fits well into the standard memory of a state-of-the-art machine, making this approach well suited to be integrated into the standard build process on a developer's desktop machine.

Discussion. There are a couple of pathological cases where NuSMV takes disproportionally long and some where Goanna, i.e., the tree matching, takes very long. There are two sometimes interrelated reasons for this: First of all, Goanna's tree matching is impacted by the number of variables in a program. The current implementation runs all matching operations for all properties and all variables in separate runs, creating a rather large overhead. For programs with few variables, the impact is not significant, however, when analyzing hundreds of variables it is considerable. An example of this effect can be seen in Figure 7, where the outliner with 70 seconds run-time is caused by a source file that has a large number of variables. Consequently, we have plans to optimize the tree matching in the future.

Secondly, NuSMV is impacted by the number of variables and the complexity of the control structure. Moreover, the BDD encoding plays a major role. As is typical in BDD-based model checking, run-times are sometimes hard to predict and fluctuate wildly when changing the variable order. An explicit state model checker might be more suit-

able for the analysis and we will explore the option in the future.

5.4 Scalability and Precision

One of the encouraging results of our case study is that performance scales nicely when adding more properties. In fact, going from one property to 15 properties only doubles the analysis time. With an optimized tree matching, we expect to further improve this ratio.

There are two main reasons for this: Some of the labels created for certain properties can be reused. E.g., the label for a variable x being declared ($decl_x$) might be part of several properties and as such can be reused. Right now, we only do this to a limited extend in Goanna.

The other reason is that NuSMV scales well when adding more labels. Since the underlying control structure for one property and 15 properties is the same—only the number of labels increases—it is not required to run NuSMV more often for a larger number of properties. Again, for reasons of the exact BDD encoding, it is sometimes difficult to precisely predict how large an increase in run-time can be expected by adding a certain number of properties/labels.

The precision of our analysis is very much dependent on the exact property encoding. Some of our properties come in two flavors: A strict version and a relaxed version. The strict version tends to be an under-approximation of the program's behavior and the relaxed one an over-approximation. In the case of uninitialized variables, the strict version flags a violation if a variable is uninitialized on *all* paths and the relaxed version reports a violation if the variable is uninitialized on *some* path. The difference is in the path quantifier of the CTL formula (for all/exists a path).

The strict versions typically create zero false alarms, while the relaxed versions are comparable to other static analysis tools which do not do any additional path pruning or similar techniques to remove impossible paths.

6 Future Work and Conclusions

Summary. In this work we presented an approach to use model checking for solving static analysis problems. Moreover, we implemented Goanna, the first tool that uses an off-the-shelf model checker as a static analysis engine. This brings the two worlds of static analysis and model checking one step closer together. We believe that in the future this will enable the integration of more semantic-based software model checking techniques into static analysis while retaining one uniform framework.

The approach has been shown to be scalable to real-life software projects. We evaluated our prototype tool with respect to OpenSSL and showed that most files are analyzed in the same order of magnitude as the compilation itself. This enables the integration of model checking into the standard build process, increasing the overall software quality.

Current limitations and future work. The Goanna tool is still at a prototype stage. While it is already fast enough for practical use, it is far from optimal and there is still much room for significant performance improvements. One particular aspect is our current implementation of the tree matching algorithm which traverses the tree for each and every subpattern, creating a large unnecessary overhead.

Another area that hasn't been addressed so far is path pruning to achieve a higher precision for relaxed properties. We are planning to take advantage of the model checking approach by including more semantic-based techniques—such as predicate abstraction—to rule out infeasible path combinations.

Finally, while in principle our approach extends to classic inter-procedural analysis, we still have to develop heuristics to deal with the combinatorial blow up in order to keep the analysis to a similar speed as it is for intra-procedural analysis. We are in the process of creating a two-pass analysis by making use of summaries which can be generated from the intra-procedural stage.

Acknowledgements We thank Bernard Blackham, Jörg Brauer, and Sean Seefried for many fruitful discussions and comments.

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

References

[1] T. Ball and S. K. Rajamani. The SLAM Toolkit. In *Intl. Conf. on Computer Aided Verification (CAV '01)*, pages 260–264, London, UK, 2001. Springer-Verlag.

[2] M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of xpath fragments. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, pages 79–95, London, UK, 2002. Springer-Verlag.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné;, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. pages 85–108, 2002.

[4] S. Chaki, E. M. Clarke, A. Groce, and O. Strichman. Predicate Abstraction with Minimum Predicates. In *CHARME*, pages 19–34, 2003.

[5] B. Chess. Improving computer security using extended static checking. In *SP '02: 2002 IEEE Symposium on Security and Privacy*, page 160, Washington, DC, USA, 2002. IEEE Computer Society.

[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Intl. Conf. on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*. Springer, 2002.

[7] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[8] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons for branching time temporal logic. In D. Kozen, editor, *Logics of Programs Workshop, IBM Watson Research Center, Yorktown Heights, New York, May 1981*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1982.

[9] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[10] Coverity. Prevent for C and C++. http://www.coverity.com.

[11] D. Dams and K. S. Namjoshi. Orion: High-precision methods for static error analysis of C and C++ programs. Bell Labs Technical Memorandum ITD-04-45263Z, Lucent Technologies, 2004.

[12] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. Symposium on Operating Systems Design and Implementation, San Diego, CA*, Oct. 2000.

[13] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, 2004.

[14] Gimpel Software. Flexelint for C/C++ . http://www.gimpel.com/html/flex.htm.

[15] GrammaTech. CodeSurfer . http://www.grammatech.com/products/codesurfer/.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.

[17] G. Holzmann. Static source code checking for user-defined properties. Pasadena, CA, USA, June 2002.

[18] K. Rustan M. Leino, James B. Saxe and R. Stata. Checking Java programs via guarded commands. Technical Report 1999-002, Palo Alto, CA, 1999.

[19] Klocwork. K7 . http://www.klocwork.com/products/klocworkk7.asp.

[20] Microsoft. Prefast . http://www.microsoft.com/whdc/devtools/tools/PREfast.mspx.

[21] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[22] J. Mühlberg and G. Lüttgen. BLASTing Linux code. In *Proc. of the 11th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 06)*, volume 4346 of *LNCS*, To appear.

[23] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[24] PolySpace. PolySpace for C++ . http://www.polyspace.com/cpp.htm.

[25] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proc. Intl. Symposium on Programming, Turin, April 6–8, 1982*, pages 337–350. Springer-Verlag, 1982.

[26] B. Schlich and S. Kowalewski. Model checking C source code for embedded systems. In *Proc. of the IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation*. NASA/CP-2005-212788, Sept 2005.

[27] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, New York, NY, USA, 1998. ACM Press.

[28] D. A. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In *Intl. Symposium on Static Analysis (SAS '98)*, pages 351–380, London, UK, 1998. Springer-Verlag.