Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking

Hendrik Post, Carsten Sinz
University of Karlsruhe
ITI VerAE Group
Karlsruhe, Germany
{post, sinz}@ira.uka.de

Alexander Kaiser
University of Tübingen
Symbolic Computation Group
Tübingen, Germany
akaiser@rkaiser.de

Thomas Gorges
Robert Bosch GmbH
Chassis Systems Control
Leonberg, Germany
Thomas.Gorges@de.bosch.com

Abstract

Fully automatic source code analysis tools based on abstract interpretation have become an integral part of the embedded software development process in many companies. And although these tools are of great help in identifying residual errors, they still possess a major drawback: analyzing industrial code comes at the cost of many spurious errors that must be investigated manually. The need for efficient development cycles prohibits extensive manual reviews, however. To overcome this problem, the combination of different software verification techniques has been suggested in the literature. Following this direction, we present a novel approach combining abstract interpretation and source code bounded model checking, where the model checker is used to reduce the number of false error reports. We apply our methodology to source code from the automotive industry written in C, and show that the number of spurious errors emitted by an abstract interpretation product can be reduced considerably.

1 Introduction

When verification technology is to be used in an industrial software project, the decision which paradigm to choose is often based on a tradeoff between level of automation, acceptable runtimes for analysis, and required precision. Robert Bosch GmbH, the world's largest supplier of automotive components, decided to employ the commercial abstract interpretation tool Polyspace [14] as part of their standard software development process for driver assistance systems on embedded devices. Other currently employed quality measures include traditional unit and system tests as well as code reviews. And although Polyspace is able to produce impressive results, it suffers from a high false positive rate. One reason for this is the complexity of the software that is used nowadays on embedded devices, and it must be expected that it will even increase in the future.

Previous projects like Orion [9] have already indicated a solution to the problem of increasing complexity in analysis and verification tasks. They propose to combine different analysis techniques: in a first step, a global light-weight data-flow analysis component is used to generate potential error candidates or warnings. Then, in a second step, high precision solvers check the feasibility of these potentially spurious warnings. The first step may help the second one by restricting the extensive original source code to fractions that are relevant to reproduce the error.

In this work we adapt the original Orion idea by replacing the light-weight dataflow technique by the abstract interpretation tool Polyspace. Instead of the solvers CVC and Simplify [10] which are used in Orion, we employ the source code bounded model checker CBMC [5] to reduce the number of false positives. We have also conducted limited experiments with another model checker, SATABS [6], which uses counterexample-guided abstraction refinement (CEGAR). We use the bounded model checker in two ways: first, we feed it with the error reports provided by Polyspace. This step, which we call *Phase A*, tries to obtain more information about the errors in order to classify them as real or spurious. Note that Phase A can be automated as it requires few, trivial user interactions. If Phase A cannot successfully handle an error report, we append a second, manually guided step, called *Phase B*. In this second step we add further information like invariants, input constraints and non-formal requirements before performing another run of CBMC to further improve the warning classification.

Although Phase B involves manual guidance, two advantages compared to common warning inspection arise: The required guidance (in our experiments, at least) is limited to providing very simple invariants—e.g. input constraints

of integer variables like `i > 0`—which is easier than a rigourous inspection of complex C code involving possible arithmetic overflows. Additonally, the confidence in the analysis is increased, as a formal proof on the basis of the former information might be created.

As a side-effect, Phase B gives information about the origins of the residual class of warnings. This class can be refined into a set of warnings that could not be discharged or proven because of limitations of the technique and as set of warnings that could not be checked because of missing information, i.e. these warnings could not be refined by any possible verification technique.

Note that our analysis, because it starts with the results emitted by Polyspace, is not a comparison between abstract interpretation and bounded model checking, but rather deals with possible refinements of the first by the latter.

In our case study, which is based on 77 warnings produced by a run of Polyspace, we could reduce the number of warnings by more than 23% in Phase A (cf. Section 3.1) by either identifying them as real errors[1] or rejecting them. Manually guided post-processing, Phase B (cf. Section 3.2), could then reduce the number of warnings by 70% of the original number. It is important to note that in all cases where an error could occur—or is inevitable—a concrete counter-example trace is obtained by using CBMC. This trace significantly helps to understand the problem.

The problem of proving any non-trivial property about a computer program is well known to be undecidable. Common program analysis techniques therefore deal with approximations that may be sufficient for some programs and a limited set of properties. A common shortcoming of all these approaches is that they either report property violations that may not occur (*false positives*), or they do not report violations that may in fact occur (*false negatives*). Due to the undecidability of the verification problem, it is impossible to entirely avoid both false positives and false negatives for all possible input programs. Nevertheless, many different techniques have been tuned and applied successfully to even industrial sized software systems. The bandwidth ranges from light-weight data-flow techniques [11] up to the semi-automatic application of first-order dynamic logic provers [12]. In general, there is always a tradeoff between precision, level of automation, and required resources, which vary among different techniques and application areas.

For the analysis of safety critical software systems, false negatives must be avoided with highest priority. False positives, on the other hand, may be acceptable, but cause a significant development overhead due to the manual inspection required for every—possibly spurious—violation.

[1] Note, that in a concrete execution on embedded hardware these errors are handled in some implementation specific way, i.e. not necessarily the termination of the program.

We now give a short definition of the basic techniques that we use in our approach. For a more detailed presentation we refer the reader to the literature [3, 7].

Abstract Interpretation. Static analysis by abstract interpretation is a technique that tries to prove the absence of runtime errors by analyzing the source code of a program. Among the errors that are typically checked are buffer overflows, invalid pointer accesses, array bounds checks, and arithmetic under- and overflows. Abstract interpretation has been successfully applied to many imperative programming languages. It uses an overapproximation on the set of possible program execution traces, i.e., it considers more program execution paths than the program can actually perform. Traces are iteratively extended until the program terminates, reaches an error state, or arrives at a fixed limit of execution steps. Overapproximation uses techniques such as abstract variable domains and use of widening and narrowing operators [7]. In the presence of loops, recursion, or any other re-entrant code, the approximated program semantics is obtained by fix-point computation. It is well known that these abstractions typically introduce false positives (see, e.g., [16]).

Bounded Model Checking (BMC). Bounded model checking (BMC) is a technique that was introduced by Biere *et al.* [3] to check properties of hardware designs, but has later been extended to also allow verification of C programs [5]. Bounded model checking generates program execution traces with bit-precision on the data level. It cannot handle unlimited recursion and restricts loop executions to a fixed bound (by unwinding loops up to this bound). If the bound is high enough to capture the system semantics, BMC is sound and complete. If the bound is too low a warning about the possible unsoundness is usually provided. Values of fixed-size variables are not approximated, but handled on the bit-level instead. Variable-size (possibly infinite) data structures must be approximated, though. This means, that BMC implements a precise program semantics up to the point where the loop, recursion or data structure size bound is reached. By using a precise semantics on fixed-size variables, effects like overflows can be handled accurately. An implementation of BMC for C programs is CBMC [5]. We have used CBMC for the experiments reported in this paper. Internally, CBMC generates program traces by modelling the effects of each program statement as a propositional logic formula. Each formula encodes a single-step transition relation. These formulas are then put together for all program steps on a trace. A further formula is added that excludes entering error states. The complete formula is then checked by a propositional logic satisfiability checker. The execution bound imposed by BMC removes the need for fix-point computations, which dif-

```
1: void foo(unsigned int var) {
2:       var = var & 0x02;
3:       assert(var==0x02 || var==0);
4: }
```

Figure 1. Polyspace cannot prove that the assertion in line 3 is true. As CBMC models data operations on the bit level, it is able to discharge the spurious warning.

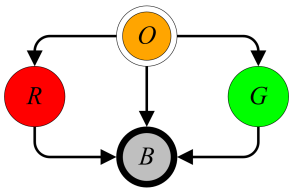

Figure 2. Polyspace provides four claim-states. When viewed as a refinement process, all claims start at O. Further iterations may then refine the model and determine which claim may be shifted to one of the states R, G or B. Note that R and G claims may still be proven unreachable (B).

ferentiates bounded model checkers from general symbolic model checking algorithms, in which fix-points are computed. General model checking approaches are also in use for software verification, and may either work on abstract models which resemble abstract domains, or they follow a dynamic refinement process where modelling precision is increased as necessary (CEGAR). Bounded model checking has already been applied successfully for medium to large scale software projects, e.g. for checking the correctness of Linux kernel modules [15]. Model checking by abstraction refinement has also been applied successfully [1, 4].

To illustrate the typical cases for refinements by bounded model checking, a minimal C program in Figure 1 is given. Polyspace is not able to show whether the assertion holds or not, whereas CBMC deduces that the claim must always hold.

2 Abstract Interpretation with Polyspace

Our work deals with the analysis whether *runtime errors* may occur in embedded C programs. Runtime errors do commonly include: division-by-zero, variable domain over- and underflows, as well as array index operations on invalid elements. Polyspace presents a *warning* for each potential line that may lead to a runtime error when executed. Such a warning includes the line in the source code as well as a specification which kind of runtime error may occur. The word warning indicates that not all program executions will fail at the reported location. If no program execution whatsoever may ever produce a failure at this point in the program, the warning is called *spurious*. The expression *false positive* is used synonymously.

A *claim* represents a (location-bound) program property that must be true under all program executions. A claim that is false on at least one execution trace implies that Polyspace emits a warning (due to the soundness of Polyspace). Claims represent proof obligations for program verification. The process of software analysis maps claims to a discrete set of claim-states, depending on whether the claim holds or not. This set typically includes at least the states TRUE, FALSE, and UNKNOWN, where the last value indicates that it could not be determined whether the claim holds or not. Polyspace provides an enriched set of claim-states, where each state is indicated by a color:

R)ed: The claim will always fail, i.e. the claim-state is FALSE under the assumption that the code is reachable.

O)range: The generated overapproximation allows execution traces where the claim evaluates to FALSE. It is not determined, however, whether the claim holds or not in the original program. Commonly the claim-state would also be labelled FALSE.

G)reen: The claim evaluates to TRUE for every program execution.

B)lack: Program executions do not reach this location, i.e. the location is either *dead code* or its execution is masked by a previous runtime error.

Note that G and R claim-states can be promoted to B, if re-evaluation of another orange claim causes the line to become non-reachable. Possible transitions between claim-states are indicated in Figure 2.

Table 1 presents a quantitative summary of the results that we obtained running Polyspace on Bosch's driver assistance software. The part of Bosch's software we examined consists of nine components, which were analyzed independently by Polyspace. The analysis run reported 468 error candidates (claim-state O or R), out of which three could be confirmed as real errors by Polyspace. For the other 465 cases only a warning was produced, which indicates that the system could not determine whether it is a real or spurious error. The errors reported by Polyspace fall into four categories: array index out of bound, division by zero, under- or overflow, and other, not further categorized errors, e.g. shift overflows.

Although the number of reported errors is quite low (a total of three), a lot of warnings are produced. Out of the

Table 1. Polyspace reports 468 possible errors in the analyzed system (3 errors, 465 warnings; R and O columns). The three R-labelled claims will always cause a runtime error on execution. O-colored lines of code indicate that an error may occur at that line. G-claims tell that no error can occur at this location, while B-claims show locations that are not reachable at all. We have selected 77 out of these error candidates for a further analysis with our proposed methodology.

Software	Array Index Operation				Division by Zero				Under-/Overflow				Other			
Module	R	O	G	B	R	O	G	B	R	O	G	B	R	O	G	B
1	0	6	192	81	0	0	8	24	2	10	875	1005	0	27	3214	2551
2	0	5	577	3	0	0	17	1	0	31	1012	28	0	99	5148	157
3	0	0	14	0	0	0	3	0	0	13	151	30	0	21	1522	236
4	0	0	13	0	0	0	0	0	0	4	96	0	0	0	360	3
5	0	3	467	16	0	1	31	3	0	38	1097	94	0	58	4511	490
6	0	17	135	26	0	0	6	2	0	12	686	184	0	19	3385	1054
7	0	1	32	0	0	0	2	0	0	0	54	0	0	11	611	5
8	0	0	65	0	0	2	12	0	0	0	130	14	1	10	352	91
9	0	6	40	6	0	0	4	0	0	11	419	18	0	60	3595	177
Sum	0	38	1535	132	0	3	83	30	2	119	4520	1373	1	305	22698	4764
Selected	-	38	-	-	-	3	-	-	2	34	-	-	0	0	-	-
%Selected	-	100%	-	-	-	100%	-	-	100%	29%	-	-	0%	0%	-	-
Total Sel.	77/468 (16.5%)															

proclaimed 465 warnings we selected 75 as well as two reported overflow errors for our further analysis. Our criteria for selecting these instances were as follows: we selected all warnings in the first two categories (42), as these checks are also built into CBMC. Among the under- and overflow errors we selected those, where CBMC did not have problems to parse the C input files[2]. We also included under- and overflows that were detected by Polyspace as real errors, first, to confirm them with another tool and, second, to get further localization information and error traces by CBMC. We did not include any warnings of category "others" into our analysis so far, due to time restrictions.

Our goal in the next sections is to reduce the number of these 75 warnings reported by Polyspace by using another more precise analysis engine, namely CBMC. Without this further step, all of the warnings would need to be analyzed manually. We also want to confirm the two overflow errors reported by Polyspace, and obtain more information about the origin of these errors.

3 Integrating BMC

Our approach to reduce spurious warnings is accomplished by doing an iterative filtering. Each warning emitted by Polyspace has an associated claim-state that indicates the so-far obtained knowledge about it. Starting with Polyspace-generated claim-states, a two-phase refinement is performed that aims at individually moving each warning towards a state that yields a higher level of knowledge.

[2]As CBMC is still in an early stage of development, it is not able to handle all C constructs properly. Assembler and certain C++ passages also prevent straightward application of CBMC and SATABS.

For example, the bounded model checker CBMC may prove that a Polyspace-reported possible zero division error is spurious. Thereby, a warning O is upgraded to belong to the set of safe claims G.

Filtering works by applying two refinement phases on the Polyspace report. Phase A, and the alternatively to be used Phase A', encompass an almost automatic application of source code model checking techniques. In Phase A, the bounded source code model checker CBMC is used as a filtering engine. Phase A' differs in so far, as CBMC is replaced by SATABS. The result of this analysis may be a proof that a warning is spurious, or a concrete program trace that shows under which conditions the error occurs. In the former case the warning is removed, in the latter case a manual investigation is needed. However, as the counterexample trace delivers additional information, the manual inspection can be performed more efficiently. CBMC may also be used as an interactive tool in this phase to recheck the error after having made modifications to the code or having provided further assumptions about code usage. We call this manual, but tool-assisted part of the filtering Phase B.

3.1 Phase A: Automatic Filtering through CBMC

Phase A iterates over all warnings of the input set.

As tools like CBMC or SATABS are typically used for modular program analysis, they need an additional parameter: the function to be used as an entry point. The analysis then covers this entry function as well as all functions that are directly or indirectly called from it and for which source

code is provided. We call the set of functions included in the analysis the *code context*. We start with a minimal code context, i.e. the function where the line producing the warning is located in. Afterwards, the context is extended iteratively until the warning is either discharged or CBMC does not terminate in a fixed amount of time. The idea of stepwise context extension is illustrated in Figure 3.

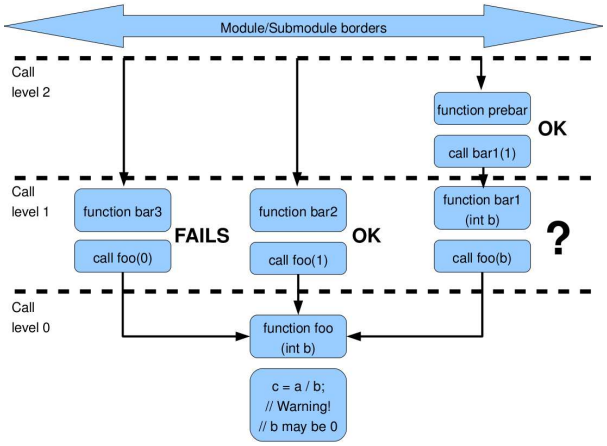

Figure 3. Example of Iterative Context Extension for CBMC.

Function `foo` contains a line of code that has lead to a division-by-zero warning. When trying to discharge the warning, CBMC will start using `foo` as the program entry point. CBMC will discover that the claim may be violated at local scope, as b might be zero. So the scope is extended by adding—one after one—functions `bar1` to `bar3` (on level 1) to the context. CBMC is run with each of these functions as an entry point. `bar3` implies that the claim must be flagged as an inevitable error, and this will terminate the analysis process. Assuming that function `bar3` was not present in the example, CBMC would prove that on the call-chain `bar2->foo` the claim holds. It would then move on to `bar1`, checking the call-chain `bar1->foo`. This call-chain (and the whole claim) would be flagged green after having made another extension step adding function `prebar` (on level 2) to the context. If CBMC fails to terminate within a fixed amount of time, respectively if the bound cannot be set high enough to recieve a soundness guarantee, the claim remains flagged O. Otherwise, if CBMC provides a counter-example trace, the claim will be flagged O_c instead of O, indicating that a counterexample has been provided for the claim of this warning. Our experiments have been performed manually, though an automatic implementation of Phase A could be accomplished with moderate effort.

Phase A': Automatic Filtering through SATABS In Phase A' CBMC is replaced by the model checker SATABS[6]. SATABS features counterexample-guided abstraction refinement, and is therefore expected to provide a broader coverage due to its powerful automatic abstraction technique.

3.2 Phase B: Manually Supported Analysis of Residual Warnings

Phases A and A' can be performed automatically and lead to a substantial reduction of warnings. The class of residual warnings is treated in this phase, Phase B. This study encompasses three possible treatments to further reduce the number of unclassified warnings.

1. *Library functions.* The software we analyzed consists of library functions and application-specific code. For library functions there is the additional requirement that they have to be total, i.e. work for all possible input values. So, in case CBMC provides a counterexample for a warning located in a library function, this warning can be directly classified as an error without checking actual calls to this function from application-specific code. (By using function signature annotations for library functions, this step could also be automated.)

2. *Underspecified input channels.* Cross-module interaction is performed using shared memory directives. Accessing shared memory must be performed using unique `get` and `set` functions. Input and output ranges for cross-module interactions are currently not formally specified and provide a source for spurious errors. Manual code review can provide reasonable constraints about the ranges of input variables, however. Using `assume` semantics, this information can be passed to the model checker which then is able to prove that the system under test is correct under the given assumptions.

3. *Problem decomposition.* The state explosion problem as well as infinite loops pose a natural obstacle to the application of CBMC. Indeed, several warnings arose in a setting, where variable and constant initialization is performed in a separate initialization function that is called once at the beginning of the module's lifecycle. Two problems are induced by this setting: first, the function where the warning was located, was itself embedded in an infinite loop. Secondly, any experiment had to include the initialization function as well as the function under test. The resulting large chunk of infinitely often executed code could be decomposed by

manually performed invariant generation for the infinite loop. All classical loop-invariant proof obligations could then be provided in greatly improved runtime.

We will now present three examples for each of the above categories. Note that the above classification does not imply full coverage. However, the class of remaining errors that could not be analyzed semi-automatically is rather small. All code samples have been simplified, function names have been renamed for non-disclosure reasons.

Library functions. The following function `add()` may lead to an overflow in line L1, e.g., for `o1 = -10000` and `o2 = 40000`.

```
short add(short o1,long o2){
  if(o2>=(32767-(long)o1))    // (40000>=42767)
    return 32767;
  else{
    if(o2<=(-32768-(long)o1)) // (40000<=-22768)
      return -32768;
    else
L1:   return o1+(short)o2; // (short)40000 fails
  }
}
```

As `add` is in fact a library function that bears the requirement to accept all inputs, the overflow error is confirmed without further analysis using iterative context extension.

Adding input constraints. The following function `c_getval()` reads a `short` value from a shared memory location (the `ext[]` array). L2 may yield an overflow due to the implicit cast to the return type `short` and is therefore flagged O by Polyspace, SATABS and CBMC.

```
extern unsigned int ext[10];

_Bool __precondition(){
  return (ext[0] & 0xfff << 3)+(ext[4] & 0xf)
     < 32768-5;
}

short c_getval(){
L2: return (ext[0] & 0xfff << 3)
      +(ext[4] & 0xf);
}
```

Developers indicated that the overflow is spurious because the value encoded in `ext[]` is limited to a small interval. A sufficient restriction to avoid the overflow is encoded in the function `__precondition()`. The proof obligation that `__precondition()` always evaluates to `true` in the given source code could be established by inspecting all write accesses to `ext[]`[3]. CBMC could then prove that—assuming `__precondition()` evaluates to `true`—L2 is safe.

[3]Neither CBMC, SATABS or Polyspace could prove this global invariant. However, Polyspace simplifies this task by reporting all possible write accesses to `ext[]`.

Compositional verification. The following example illustrates the treatment of infinite loops by decomposition. Note that `m_init` is invoked once upon the start of the module's life-cycle. Afterwards, `m_main` is executed periodically.

```
// L3: array index within bounds [0..9]
short ready,i,x; long v[10];

_Bool m_invariant(){
  return ready? (x<10 && i<=x && i>=0) : 1;
}

void m_init(){ ready = 0; }

void m_main(){
  if(!ready) x=10,i=0;
  for(;i!=x; i++)
    if(m_getval()+5)
L3:   v[i]=add(-i,c_getval()*2L);
    else{
      ready=0; break;
    }
}
```

The spurious warning stems from line L3. The claim is that $v[i]$ obeys the array index borders. If `m_invariant()` holds globally, L3 is safe: The key observation is that `x` and `i` are always initialized when execution reaches L3. Initialization is performed upon the first execution of `m_main()`. The invariant expresses that when entering `m_main()`, initialization will be performed immediately or it has been performed in a preceding execution of `m_main()`. The invariant has been proven to hold after `m_init()`. If it holds prior to a single execution of `m_main()`, it will also hold afterwards. The latter proof obligations were proved using CBMC.

Additional Claim States. Our former classification into G, O, R, and B claims (cf. Section 2) is not adequate to describe the results of Phases A, A' and B. The classification is therefore extended. Claims can now be in the following states:

C)yan: Warnings that have been proven correct by a compositional verification task described in 3.2 with some restrictions on the component-input.

O_C/O: Warnings with/without a (possibly spurious) counterexample that could not be rejected in Phase A, and the compositional verification task could not be completed (because of complexity or modelling issues, e.g., type punning).

M)agenta: Warnings with counterexamples that appear in a library function, or that have been proven a true error using compositional verification.

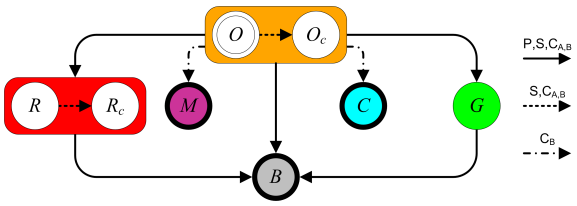

Figure 4. Extended transition graph of claim-states. In contrast to Polyspace's claim-states (cf. Figure 2), the states R_C, O_C, C, and M have been added. Arrows indicate that transitions may be triggered by Polyspace, SATABS or CBMC in all phases. Dotted arrows express that SATABS and CBMC may provide counter example traces. Transitions between O/O_C and M/C stem from the semi-automatic application of SATABS and CBMC in Phase B.

Additional claim-states and transitions are depicted in Figure 4. O-claims can be transformed to one of the classes R, M, B, C or G.

4 Result Summary

Table 2 shows the filtering results after Phases A, A' and B. It is worth noting that SATABS is not able to provide more information than CBMC. We found only one case where SATABS provided a counter-example where CBMC did not. Vice versa, CBMC was able to sort out 18 warnings by classifying them as G or B — SATABS identified only 3.

Phase B substantially refined the results from Phase A. Less than one third of the warnings remained in state O after Phase B. These examples include 12 warnings that involve type punning. Nine other residual problem cases are at least decorated with a concrete counter-example. Only 3 cases could not be refined due to performance issues.

Phase A can be performed as a fully automatic refinement step. A reduction of 23% can be considered to be a substantial improvement over the current situation.

Phase B involved manual application of bounded model checking techniques. In three weeks, a university student was able to prove 12 new errors (M). Moreover, the analysis yielded information beyond the previously available interface specification for 13 C-colored claims. 53% of the warnings are now classified to be safe.

Finally, only 23 (30%) of the warnings remain subject to further code reviews. More than half of these were caused by the not yet implemented support of type punning in CBMC. 8 of these are now combined with a concrete counter-example trace. 3 cases could not be completed because of performace issues.

The results indicate that CBMC is better suited to discharge warnings from abstraction based tools. Note that this result is limited to the current domain of data-intensive low-level system code. Moreover, the claim types, e.g. under- and overflow, may naturally pose an obstacle for automatic predicate refinement used by SATABS.

Runtimes. All experiments have been performed on a computer equipped with an Intel Pentium 4 processor running at 3 GHz with 4 GB of main memory. For all experiments a seven hour timeout has been used. Three tables (Tables 3, 4 and 5) summarize details about the runtimes and memory consumption for every claim.

Table 4 presents runtimes from Phase A where claims lead to the same result using SATABS and CBMC. In cases where the verification of a claim failed, SATABS and CBMC have similar runtimes. CBMC is able to analyze seven cases that SATABS cannot. Vice versa SATABS provides information about one claim that was not analyzable by CBMC. These claims significantly contribute to the on average higher runtime of CBMC shown in Table 3.

Cases where the warning could be discharged show that CBMC is clearly superior to SATABS in Phase A. CBMC discharges 18 warnings where SATABS only covers three warnings. The rather insignificant subset of the three common cases were analyzed much faster by CBMC (Table 4).

It is notable that the propositional logic (CNF) formulas generated by CBMC that could be successfully handled had up to seven million variables and 23 million clauses. The satisfiable instances were solved much faster than the unsatisfiable ones.

Table 5 indicates that in Phase B, where additional information—invariants or input constraints—was provided, CBMC terminated much faster.

5 Related Work

This work covers two basic verification techniques: abstract interpretation and bounded model checking. The former has already been successfully applied to low-level software systems as shown by other authors: In [13], the abstract interpretation tool Astrée is applied and tuned for aerospace software. The authors stress that a tight customer and tool developer cooperation may significantly improve the process of manually removing, respectively investigating, warnings. On a non-modular control/command program all spurious warnings could be removed. Our work deals with modular programs and data-sensitive properties, therefore it is doubtful whether all warnings could have been avoided with a tool like Astrée.

Table 2. Results after the refinement performed in Phases A, A', and B. The initial claim-states provided by Polyspace undergo transitions as shown in Figure 4.

	Start	Phase A/A'			Phase B
	Polyspace	SATABS	CBMC	Combined	CBMC
B	0	1	1	1 (≈1%)	1 (≈1%)
G	0	2	17	17 (22%)	27 (35%)
C	-	-	-	-	13 (17%)
O	75 (97%)	43	22	21 (27%)	15 (19%)
O_c	-	30	36	37 (47%)	8 (10%)
M	-	-	-	-	12 (16%)
R	2 (≈3%)	0	0	0	0
R_c	-	1	1	1 (≈1%)	1 (≈1%)

Rival [16] conducted an analysis of the origins of false alarms using the abstract interpretation analyzer Astrée. The time-consuming process of reviewing error reports is improved by a proposed framework for semi-automatic investigation. In accordance with our approach, Rival proposes to use sound analysis techniques to filter false alarms. Instead of using model checking, he refers to different static analysis techniques based on backward analysis, trace partitioning and slicing. In three case studies five false alarms could be discharged semi-automatically. It is unclear how his work relates to the large set of warnings that we observed in our case study. In contrast to Phase A, their approach involves manually choosing adequate execution patterns and constraints on input channels. Therefore, their work resembles more our Phase B. The runtimes reported in their study are significantly lower than in our case study, however.

Bounded source code model checking as implemented by CBMC [5] has been applied to Linux Device Drivers [15]. Though CBMC could expose many hard-to-find errors, it could not provide a full coverage of the complete code and all paths. Unbounded loops, callbacks, and the general problem of modularity remain an obstacle.

Examples for software model checking case studies on large, low-level code bases are given in [1] and [4]. These case studies can best be compared with our application of SATABS. While SATABS has been outperformed by CBMC in our experiments, the two studies report that abstraction/refinement-based model checking may yield good results on very large software. We believe that this contradiction is due to the differences of the analyzed code and the monitored program properties: The Bosch code-base is less control/command oriented and, secondly, the properties being checked are not high-level API safety-properties (as in[1, 4]). Our observation is that unbounded model checking—even with abstraction refinement—does not work very well on checking data-sensitive properties in low-level systems code.

Several groups have already considered various combinations of technologies for software verification. The Orion project implements a combined data-flow analysis with the SMT-solver simplify [10]. In the original work, the authors explicitly suggest to evaluate other combinations as the one covered in their paper.

Beyer et al. [2] describe the theoretical convergence between program analysis and model checking. They present an algorithm with implementation that combines lattice- and tree-based analysis. Their experiments on a small code base indicates large improvements over both approaches taken on their own. In contrast to their work, we have covered a much larger codebase of industrial size. Moreover we did not need to create a combined analysis method by following an iterative approach. Schmidt [17] presents—to the best of our knowledge—one of the earliest works dealing with combinations of model checking, flow analysis and abstract interpretation. In contrast to our work, the author concentrates on integrating abstract interpretation, model checking and data flow analysis. His work focuses on yielding benefits from the differences between model checking and abstract interpretation.

Csallner and Smaragdakis [8] present a combination of testing and static analysis. Similar to the approach presented in this paper, the authors refine the results of a static analysis tool by creating test cases from the generated analysis output. These test cases may prove that warnings and potential errors are spurious. This technique can be combined with the BMC based filtering proposed in this work.

6 Discussion and Experiences

The idea to combine different verification techniques is appealing and comes at a low cost. We have shown that in our case CBMC was able to discharge more than 20% of Polyspace's warnings automatically. Moreover, by man-

Table 3. Summary of all runtimes of Phase A and of the Polyspace analysis (grouped by analysis result, timeout after 7h). VF indicates that a claim may be violated (failed) while VS stands for a successful verification outcome.

	More information		Errors / No additional information				
	VF	VS	C-semantic	Refinement	Segm. fault	Timeout	Total
SATABS							
$\sum t$ [h:m]	0:36	4:12	0:12	16:22	28:18	56:27	106:09
Timeout	0	0	0	0	0	8	8
$\#claims$	31	3	12	14	9	8	77
CBMC							
$\sum t$ [h:m]	9:15	19:57	0:01	-	6:23	42:06	77:43
Timeout	0	0	0	-	6	6	6
$\#claims$	37	18	12	-	4	6	77
Polyspace							
$\sum t$ [h:m]	-	-	-	-	-	-	28:08
$\#runs$	-	-	-	-	-	-	9

Table 4. Comparison of runtimes for cases in which SATABS and CBMC produced identical results (Phase A, grouped by analysis result, problems resulting from unsupported language features ignored).

	VF		VS		Out of memory (oom)		Total	
	CBMC	SATABS	CBMC	SATABS	CBMC	SATABS	CBMC	SATABS
$\sum t$ [h:m]	0:39	0:36	0:02	4:12	1:58	0:23	2:39	5:12
$\#claims$	30		3		1		34	

Table 5. Summary of CBMC runtimes sorted by phase and verification result.

Phase → Result	A → VF	B → VF	A → VF	B → VS	A → oom	B → VS	$\sum$A	$\sum$B
$\sum t$ [h:m]	0:16	0:06	7:45	6:17	56:07	16:02	64:08	22:26
$\#claims$	1		16		7		24	

ual inspection using CBMC as a companion tool, we could reduce the number of false warnings even more. Given a safety critical application and high code review costs, the method is perfectly feasible for a verification practitioner.

Authors of other case studies, e.g. [13], claim that 100% of the spurious warnings can be removed for control/command programs. Three preconditions seem to be required to achieve such a result:

- The code has to be non-modular or the code must be formally specified in a way such that knowledge about interface borders can be inferred.
- Tool applicants and tool developers must work closely together.
- The program under test must be control/command oriented.

Cases that support all three properties exist, as the authors of [13] have shown. Nevertheless, the violation of one or more of these properties may be very likely in industrial practice. Then, manual code reviews or manual application of verification tools cannot be avoided. Results from Phase B support the claim that bounded model checking may also improve the situation then. Specifying few properties may expose hidden errors as well as discharge additional warnings. Only 31% of the warnings persisted in our case study. Of these, 50% are due to type punning being not yet supported by CBMC. For 37% at least concrete counterexample traces could be retrieved by CBMC.

The authors would like to thank Arshad Golam from Robert Bosch GmbH for his substantial support providing Polyspace reports.

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proc. of the the 2006 EuroSys Conference*, pages 73–85, New York, NY, USA, 2006. ACM Press.

[2] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. of the 19th Intl. Conf. on Computer Aided Verification (CAV)*, pages 504–518. Springer, Berlin, 2007.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *5th Intl. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS), Proc.*, pages 193–207, London, UK, 1999. Springer.

[4] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proc. of the 9th ACM Conf. on Computer and Communications Security (CCS)*, pages 235–244, New York, NY, USA, 2002. ACM Press.

[5] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *10th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[6] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In N. Halbwachs and L. D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 11th Intl. Conf., Edinburgh, UK, April 4-8, 2005, Proc.*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the Fourth ACM Symp. on Principles of Programming Languages (POPL), Los Angeles, California, January, 1977*, pages 238–252, 1977.

[8] C. Csallner and Y. Smaragdakis. Check 'n' crash: combining static checking and testing. In *The 27th Intl. Conf. on Software Engineering (ICSE), Proc.*, pages 422–431, New York, NY, USA, 2005. ACM.

[9] D. Dams and K. S. Namjoshi. Orion: High-precision methods for static error analysis of C and C++ programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects (FMCO), 4th Intl. Symp., Amsterdam, The Netherlands, Nov. 1-4, 2005, Revised Lectures*, volume 4111 of *LNCS*, pages 138–160. Springer, 2005.

[10] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[11] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In M. L. Scott and L. L. Peterson, editors, *19th ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing, NY, USA, October 19-22, 2003, Proc.*, pages 237–252, New York, NY, USA, 2003. ACM Press.

[12] J. J. Hunt, E. Jenn, S. Leriche, P. Schmitt, I. Tonin, and C. Wonnemann. A case study of specification and verification using JML in an avionics application. In *The 4th Workshop on Java Technologies for Real-time and Embedded Systems (JTRES), Proc.*, pages 107–116. ACM Press, 2006.

[13] H. R. Nielson and G. Filé, editors. *Static Analysis (SAS), 14th Intl. Symp., Kongens Lyngby, Denmark, August 22-24, 2007, Proc.*, volume 4634 of *LNCS*. Springer, 2007.

[14] PolySpace Technologies. Polyspace Client / Server for C/C++, Version 4.1.1.6 , 2008, http://www.polyspace.com., 2008.

[15] H. Post and W. Küchlin. Integration of static analysis for Linux device driver verification. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods (IFM), 6th Intl. Conf. , Proc., Oxford, UK, July 2007*, volume 4591 of *LNCS*, pages 518–537. Springer, 2007.

[16] X. Rival. Understanding the origin of alarms in Astrée. In C. Hankin and I. Siveroni, editors, *Static Analysis, 12th Intl. Symp., SAS 2005, London, UK, September 7-9, 2005, Proc.*, volume 3672 of *LNCS*, pages 303–319. Springer, 2005.

[17] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proc. of the 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 38–48, New York, NY, USA, 1998. ACM.