Accurate Dead Code Detection in Embedded C Code by Arithmetic Constraint Solving

Felix Neubauer*, Karsten Scheibler*, Bernd Becker*, Ahmed Mahdi[†],
Martin Fränzle[†], Tino Teige[‡], Tom Bienmüller[‡], Detlef Fehrer[§]
*Chair of Computer Architecture, University of Freiburg, Germany
[†]Research Group Hybrid Systems, Carl von Ossietzky University of Oldenburg, Germany
[‡]BTC Embedded Systems AG, Oldenburg, Germany
[§]Sick AG, Waldkirch, Germany

***Abstract*—Unreachable code fragments in software, despite not having a negative impact on the functional behavior, can have a bad effect in other areas, such as code optimization or coverage-based code validation and certification. Especially the latter is important in industrial, safety critical environments, where detecting such *dead* code is a major goal to adjust the coverage of software tests.**

In the context of embedded systems we focus on C programs which are reactive, control-oriented, and floating-point dominated. Such programs are either automatically generated from Simulink-plus-Stateflow models or hand-written according to coding guidelines. While there are various techniques – e. g. abstract interpretation or Counterexample guided abstraction refinement (CEGAR) – to deal with individual issues of this domain, there is none which can cover all of them. The AVACS transfer project T1 aims at the combination of these techniques in order to provide an accurate and efficient dead code detection.

In this paper we present the ideas and goals of the project as well as the current status (including very promising experimental results) and future challenges.

***Index Terms*—formal specification, formal verification, SMT, floating-point, dead code detection, CEGAR, abstract interpretation**

I. Introduction

The occurence of dead (i. e. unreachable) code fragments in sizable software development projects is a regular phenomenon – it can be created either intentionally (e. g. defensive programming) or unintentionally (e. g. reusing of code, modification of program context) in hand-written programs or as a byproduct of automated code generation. Although dead code has no negative impact on the functional behavior of the software, it can have a bad effect e. g. on code optimization or coverage-based code validation and certification. Especially the latter is important in industrial, safety critical environments, where standards (e. g. ISO 26262) for high-integrity software demand, among other criteria, reaching a certain test coverage or the absence of dead code. Obviously the presence of dead code violates the latter demand but also affects the interpretation of coverage-based criteria negatively. Therefore it is a major goal to detect dead code from high-integrity software and thereby be able to adjust the test coverage.

The classical dead code detection in compilers is not suitable to meet these demands. Compilers have to be fast which does not allow complex dead code analysis at compile-time, but the analysis itself has to be sound, which means that only really unreachable code may be marked as dead. The consequence of this tradeoff is a vastly incomplete detection, where substantial amounts of dead code are not discovered. In our use cases we also need a sound method to securely detect dead code, but additionally it has to be highly accurate with respect to both the amount (enhancing the interpretation of the test coverage) and localization (approching the demand of dead code free programs) of the dead code detected.

In the context of embedded systems our focus lies on reactive, control-oriented, and floating-point dominated C programs, in particular those automatically generated from Simulink-plus-Stateflow models of embedded control applications or those hand-written according to coding guidelines. Dead code detection in this domain is quite difficult because there might be no clear distinction between control and data in the C code. Considering automatically generated code this lack of distinction is caused by encoding the major part of the model's control flow into the data part of the C code. Hand-written embedded control software on the other hand often has a central control loop which defers obligations to a set of dedicated handlers. Some of these handlers are generic, offering both a state and data dependent control flow. This construction also blurs the distinction between control and data. An additional difficulty is the dependency of the control flow on complex floating-point computations.

The development of a method to handle these challenges, namely:

(1) handling the lack of distinction between control and data
(2) dealing with the dominant impact of (often non-linear) floating-point computations on the reachability of code
(3) scaling to industrial-size programs

is the main goal of the AVACS[1] transfer project T1. This project is a transregional collaboration between the *University of Freiburg*[2], the *Carl von Ossietzky University of Oldenburg*[3] and the industrial partners *BTC Embedded Systems AG*[4] in

[1]http://www.avacs.org
[2]http://ira.informatik.uni-freiburg.de
[3]https://www.uni-oldenburg.de/hs
[4]https://www.btc-es.de

Oldenburg and *Sick AG*[5] in Waldkirch.

There are various techniques which are able to deal with individual issues of the list above. Abstract interpretation (AI) [1]–[4] can handle very large programs (3), but lacks exactness when the data part of the state space can be fractioned and reshuffled more or less arbitrarily by data operations. Counterexample guided abstraction refinement (CEGAR) [5]–[7] is often able to create concise abstractions of the interplay of control and data (1) but is currently limited to the analysis of either programs with confined arithmetic fragments (mostly linear) or of complete hybrid systems with tiny control skeletons. Regarding the second issue there was much progress lately in the field of non-linear arithmetic constraint solving based on conflict-driven clause learning over arithmetic theories (CDCL(T)) [8], [9]. It is now possible to solve extremely large arithmetic constraint systems including a complex Boolean structure and linear, polynomial and even transcendental arithmetic. But there is no support for loops in the control flow and the method does not scale well enough for complex programs with a full branching structure.

The transfer project aims at the combination of those techniques in order to handle all required challenges and to provide an accurate and efficient dead code detection. This paper presents the current status of the project, ideas and possible solutions to unresolved issues and further challenges.

Structure of the paper: After introducing the goals and challenges of the AVACS transfer project in the current section, we describe the model checking-based tool chain for structural code coverage analysis of BTC Embedded Systems in Section II. In Section III we speak about the integration of our SMT-solver iSAT3 as an alternative backend solver into this tool chain and the experimental results vs. the standard backend solver. Section IV gives an outlook about future challenges, before Section V concludes this paper.

II. Model Checking-Based Tool Chain for Structural Code Coverage Analysis

One of the most prominent test criteria for production code in industrial and in particular safety-critical domains relies on structural code coverage such as function, statement, branch, and condition coverage or MC/DC [10]. Such coverage is often measured on a percentage basis by means of program simulations stimulated by so-called test cases or vectors, i. e. the number of tested coverage goals relative to the number of all goals. Intuitively, the higher the code coverage the higher the test confidence and thus the lower the probability of undetected software bugs.

In the following, we describe by way of example a typical tool chain for automatic analysis of code coverage being based on model checking backend tools.

In Listing 1, a small excerpt of a C code under test is given. Though – for illustration reasons – the code fragment is rather simple, it however contains some frequent code constructs occuring in practice which need to be handled within the tool chain. The "main"-function sketches a typical reactive program with an unbounded feedback loop: first, all input data for the current iteration or step are received, then the actual function is executed, and finally the computed outputs are transmitted. Thereafter the program waits for the next loop iteration.

```c
#include <math.h>

typedef struct { double x, y; } point_t;

double distance(int length, point_t p[]) {
  double sum = 0;
  unsigned int i;
  for (i = 0; i < length-1; i++) {
    sum += sqrt((p[i].x - p[i+1].x) * (p[i].y - p[i+1].y)
         );
  }
  return sum;
}

unsigned char step(int length, point_t p_array[]) {
  double dist_value = distance(length, p_array);
  unsigned char dist_level_ok;
  if (dist_value <= 10.0) {
    dist_level_ok = 0;
  } else {
    dist_level_ok = 1;
  }
  return dist_level_ok;
}

point_t p_array[3]; // inputs
unsigned char dist_level_ok; // output

int main(void) {
  while (1) {
    // receive inputs: p_array
    dist_level_ok = step(3, p_array);
    // transmit outputs: dist_level_ok
    // wait for next iteration
  }
}
```

Listing 1. example_1_orig.c

In each step, the "step"-function takes an array of three two-dimensional points (modelled by struct type "point_t" consisting of two variables of scalar type "double") as input and returns, whether the level of the aggregated distance of consecutive points is okay or not.

Please note that the C code contains several floating-point computations and, moreover, relies on the function "sqrt" from the standard C library "math.h". This actually is representative as there is an increasing trend of using floating-point processors and production code in industrial applications.

With regard to the use of the "math.h" library, we remark that there are two major ways of dealing with such complex mathematical functions such as "sqrt" in a model checking context: first, the production code itself resolves the problem, namely by reducing such complex functions to basic operations, or, second, to keep these functions in the code and apply a model checker that has dedicated support for such complex functions. Concerning performance, the second option has a clear benefit over the first one as implementations of mathematical functions are frequently very complex and contain several hundreds lines of code. A model checker with native "math.h" support can exploit dedicated algorithms

[5]https://www.sick.com

which are generally much more efficient. In the following, we assume a model-checking backend tool that is able to handle the function “sqrt”.

In the first step of the tool chain for analyzing code coverage, we need to annotate the original program with additional constructs to automatically deal with code coverage metrics. In our example we concentrate on statement coverage and – for the sake of simplicity – we just aim at covering the statements in lines 18 and 20. For the purpose of observing the reachability of the corresponding lines, two new variables “LINE_X_REACHED”, “LINE_Y_REACHED” are introduced. Listing 2 shows the neccessary changes for one possible annotation method.

```
[...]
// new observer variables for coverage
unsigned char LINE_X_REACHED = 0;
unsigned char LINE_Y_REACHED = 0;
[...]
  if (dist_value <= 10.0) {
    LINE_X_REACHED = 1;
    dist_level_ok = 0;
  } else {
    LINE_Y_REACHED = 1;
    dist_level_ok = 1;
  }
[...]
```

Listing 2. example_2_instr.c (changes)

After this annotation with coverage goals, normally the C code is translated to an intermediate language called SMI (which is a C-like programming language). To keep this example simple, we skip this translation and explain the further steps on C code also.

The original code is now instrumented in such a way that automatic analysis w. r. t. code coverage by a model checker is rendered possible. In order to reach the input language of a model checker, e. g. a declarative, logical description as in a SAT or SMT solver, the instrumented code needs to be transformed into a syntactically simpler form. Two usual transformation steps are the following:

- function calls are resolved
- complex structural data types are flattened

Usually, a model checker is called separately per proof obligation, in our context per coverage goal. To further optimize the code in that respect, a cone-of-influence reduction is performed, i. e. all code fragments that do not directly or indirectly influence the state of the corresponding goal expression are removed. Listing 3 shows the result of the two transformation steps mentioned above as well as an example for the cone of the goal “variable LINE_X_REACHED becomes 1”.

Depending on the capabilities and requirements of the concrete model checker, further transformations might be applied, such as:

- loop unwinding (naive or optimized; see Listing 4 for the optimized variant)
- static single assignment form, cf. Listing 5

```
#include <math.h>

// new observer variables for coverage
unsigned char LINE_X_REACHED = 0;
double p_array_0_x, p_array_1_x, p_array_2_x; // inputs
double p_array_0_y, p_array_1_y, p_array_2_y; // inputs

int main(void) {
  while (1) {
    // receive inputs: p_array
    double dist_value;
    {
      double sum = 0;
      unsigned int i;
      for (i = 0; i < 2; i++) {
        switch (i) {
          case 0:
            sum += sqrt((p_array_0_x - p_array_1_x) * (
                p_array_0_y - p_array_1_y));
            break;
          case 1:
            sum += sqrt((p_array_1_x - p_array_2_x) * (
                p_array_1_y - p_array_2_y));
            break;
        }
      }
      dist_value = sum;
    }
    if (dist_value <= 10.0) { LINE_X_REACHED = 1; }
    // transmit outputs: dist_level
    // wait for next iteration
  }
}
```

Listing 3. example_345_funcs_types_cone.c

```
[...]
int main(void) {
  while (1) {
    // receive inputs: p_array
    double dist_value;
    {
      double sum = 0;
      sum += sqrt((p_array_0_x - p_array_1_x) * (
          p_array_0_y - p_array_1_y));
      sum += sqrt((p_array_1_x - p_array_2_x) * (
          p_array_1_y - p_array_2_y));
      dist_value = sum;
    }
    if (dist_value <= 10.0) { LINE_X_REACHED = 1; }
    // transmit outputs: dist_level
    // wait for next iteration
  }
}
```

Listing 4. example_6_opt.c (changes)

The resulting code in static single assignment form can then be translated in a rather simple way into a corresponding SMT formula $\Phi(k)$ with loop iteration depth k such that the following property holds:

Property 1: The goal “variable LINE_X_REACHED becomes 1” is covered within k steps (meaning that the statement in line 18 of the original code is reached) if and only if the SMT formula $\Phi(k)$ is satisfiable.

One very important feature of the above approach is that a satisfying assignment of $\Phi(k)$ corresponds to a test case of the original program that proves coverage of the corresponding goal. We remark that construction of such a test case is a layered approach starting from the satisfying assignment of the lowermost SMT layer up to the original C code layer.

```c
#include <math.h>

// new observer variables for coverage
unsigned char LINE_X_REACHED = 0;
// inputs
double p_array_0_x, p_array_1_x, p_array_2_x;
double p_array_0_y, p_array_1_y, p_array_2_y;

int main(void) {
  while (1) {
    // receive inputs: p_array
    double dist_value;
    double sum1;
    unsigned char dist_value_leq_10;
    sum1 = sqrt((p_array_0_x - p_array_1_x) * (
         p_array_0_y - p_array_1_y));
    dist_value = sum1 + sqrt((p_array_1_x - p_array_2_x)
         * (p_array_1_y - p_array_2_y));
    dist_value_leq_10 = (dist_value <= 10.0);
    LINE_X_REACHED = dist_value_leq_10 ? 1 :
         LINE_X_REACHED;
    // transmit outputs: dist_level
    // wait for next iteration
  }
}
```

Listing 5. example_7_ssa.c

If it turns out that the SMT formula $\Phi(k)$ is unsatisfiable then the corresponding coverage goal is unreachable within k iterations. This may be a hint for dead code.

BTC Embedded Systems solves the SMT formula $\Phi(k)$ with the BMC solver CBMC in the backend using k-induction to perform unreachability proofs. In the following section we will use the interval constraint solver iSAT3[6] with Craig interpolation as backend solver and compare it to CBMC. The iSAT algorithm was developed in AVACS project H1/2.

III. iSAT3 as backend SMT-Solver

While SAT solving aims at finding solutions for propositional formulas (or proving the absence thereof), SMT aims at solving boolean combinations of so-called theory atoms. Such atoms may for example contain linear arithmetic over the reals or constraints involving floating-point arithmetic.

In eager SMT the Boolean structure as well as all theory atoms are translated into one big propositional formula. A SAT solver is then called once in order to determine whether there is a solution or not. This approach is also called bit-blasting. While it is in general not applicable to all theories, it is up to now the most prominent approach for SMT solvers addressing floating-point arithmetic – e.g. Z3[7], Mathsat[8] and CBMC[9] rely on this technique.

When doing SMT lazily, the formula is split into a set of theory atoms and a Boolean skeleton which abstracts the truth-values of the theory atoms with Boolean literals. The Boolean skeleton is processed by a SAT-solver in order to search for a satisfying assignment. If such an assignment is found, a separate theory solver is used to check whether the theory atoms are consistent under the truth values determined by the SAT-solver. In case of an inconsistency, the theory solver identifies an infeasible subset of the theory atoms which is then encoded into a clause and added to the Boolean skeleton. Thus, in lazy SMT the SAT solver is called multiple times and operates together with a theory solver in order to refine the Boolean skeleton incrementally. Therefore, this scheme is also abbreviated as DPLL(T) or CDCL(T) – with T being the theory used within the atoms.

In contrast to CDCL(T), the iSAT algorithm [8], [11] does not have such a separation between the SAT and the theory part. Instead, *interval constraint propagation* (ICP, see e. g. [12]) is tightly integrated into the CDCL framework in order to reason about the theory atoms directly – yielding a unified lattice-based view in the meantime also known as abstract conflict-driven clause learning (ACDCL) [13].

In this paper we build on the third implementation of the iSAT algorithm which is called iSAT3 [14], [15]. The search process in iSAT is similar to CDCL: it consists of alternating propagation and decision phases interspersed with the resolution of conflicts – but additionally, the first two phases are extended with ICP and interval splits. Similar to CDCL-based SAT solvers which operate on a conjuctive normal form (CNF), iSAT operates on a CNF as well – but enriched with so-called primitive constraints (PCs) which are the result of a Tseitin-like transformation of the original theory atoms. This transformation ensures that each PC contains only one arithmetic operation – which helps to keep the ICP contractors for each PC-type simple.

Thus, ICP contractors are responsible for performing the arithmetic reasoning. Adding support for new operations essentially requires adding further ICP contractors. Recently, we adapted iSAT3 in order to support accurate reasoning for floating-point arithmetic [16]. Somewhat earlier, another ICP-based decision procedure for floating-point arithmetic was proposed in [17]. But in contrast to [17], our approach also contains support for bitwise integer operations – as C programs usually contain a mix of floating-point arithmetic, integer arithmetic and bitwise integer operations. This allows iSAT3 to be used as the first non-bit-blasting SMT solver for automatic dead-code detection in floating-point dominated embedded C programs. Thus, we did a prototypical integration of iSAT3 into the tool chain explained in Section II.

Supporting the floating-point domain also affects the completeness of the ICP-based decision procedure. While ICP reasoning is in general incomplete on real arithmetics, it gains completeness in the *finite* floating-point domain.

In the following we give a summary of the experiments performed in [16]. We relied on a set of benchmarks which originate from TargetLink[10]-generated production C code (compiled from Simulink-Stateflow models) containing floating-point arithmetic and integer arithmetic as well as casts between these two domains. Each single benchmark represents a goal defined by a structural code coverage metrics like MC/DC. The industrial-strength embedded test-vector generation toolBTC

[6]https://projects.informatik.uni-freiburg.de/projects/isat3

[7]https://github.com/Z3Prover/z3/

[8]http://mathsat.fbk.eu/

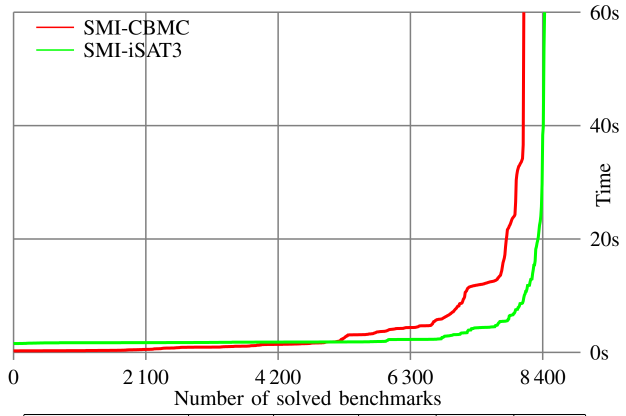
[9]http://www.cprover.org/cbmc/

[10]http://www.dspace.com/en/pub/home/products/sw/pcgs/targetli.cfm

Embedded*Tester*®preprocesses and translates the original C code (which is annotated with coverage goals) into the intermediate SMI language as explained in Section II.

In our measurements, we used the version of SMI-CBMC which is part of BTC Embedded*Tester*® 3.4 (July 2014). SMI-CBMC supports *incremental* BMC [18] and relies on CBMC 4.8[11] as its backend. But due to technical reasons the current version of SMI-CBMC does not support incremental solving in combination with k-induction.

Similar to SMI-CBMC, SMI-iSAT3 operates on SMI files as well. It is a prototypical implementation which runs in a Cygwin environment and uses a bunch of wrapper scripts in order to (1) translate the given SMI file into the iSAT3 input language, and to (2) validate every satisfying assignment found by iSAT3 by an SMI simulator. All experiments were performed on an Intel Core i7-2600 with 3.4 GHz and 8 GB RAM under Windows 7 (64 bit). We set a time limit of 60 seconds per benchmark. Setting a memory limit within the Cygwin environment did not work properly. Therefore, we omitted such a limit.



Solver	S+U	SAT	U51	U∞	TO
SMI-CBMC	8099	7424	44	631	679
SMI-iSAT3	8430	7427	172	831	348

Fig. 1. Comparison between SMI-CBMC and SMI-iSAT3 over a set of 8778 SMI benchmarks.

The benchmarks are encoded as BMC problems. For both solvers we allowed up to 51 BMC unwindings per benchmark. The results are shown in Figure 1. The diagram depicts the number of solved benchmarks together with the run time needed per benchmark. In the table, the columns S+U, SAT, U51 and U∞ show the number of solved instances (SAT: satisfiable; U51: unsatisfiable until depth 51; U∞: unsatisfiable for all depths proved with k-induction or Craig interpolation; S+U: SAT + U51 + U∞). The column TO shows the number of aborted benchmarks due to a timeout.

In order to detect unreachability (e. g. dead-code), CBMC uses k-induction, while iSAT3 uses Craig interpolation. Both solvers perform equally well when counting the number of solved satisfiable instances. The picture changes when looking at the number of solved U∞ instances – which are of special interest in the context of dead-code detection. Here, SMI-iSAT3 proves for 831 instances that the code fragment of interest is unreachable, which is an improvement of 30% compared to the 631 instances found by SMI-CBMC. Similarly, the number of solved U51 instances is higher in SMI-iSAT3: 172 compared to 44. Furthermore, when looking at the accumulated number of solved instances, SMI-iSAT3 solves 331 instances more than SMI-CBMC.

Due to the prototypical Cygwin-based integration, SMI-iSAT3 has some sort of initial runtime-offset (as revealed by the diagram in Figure 1). Despite this penalty SMI-iSAT3 outperforms SMI-CBMC in terms of runtime. E. g. when considering the instances which were solved within 3 seconds then SMI-CBMC solved 5305 such instances, while SMI-iSAT3 finished 6960.

IV. Future Challenges

While our current method is already able to cover one part of the goals of the transfer project, there are still open challenges to be solved.

A. *Counterexample Guided Abstraction Refinement*

CEGAR [5] is found to be useful in model checking while verifying large problems since its abstraction maps complex system models with infinite states to simpler ones (e. g. finite Kripke structures); thus it circumvents the state-space explosion problem. In our proposed approach, CEGAR starts from *a simple initial abstract model* representing the control flow of the original program which will be *continuously refined* based on model checking the abstraction and analysing the counterexamples generated by the model checker. In each refinement step, either *an erroneous counterexample* will be eliminated by attaching – to the abstract model – sufficient predicates obtained from the stepwise interpolants as proposed in lazy abstraction technique [19], or a real counterexample is found. This continuous refinement is feasible despite non-polynomial arithmetic as the floating-point numbers are from a large but finite domain.

Craig interpolation (CI) thus is our workhorse for abstraction refinement in the CEGAR loop. In conjunction with CEGAR, in software verification with lazy abstraction technique [19], stepwise interpolants are used to extract meaningful predicates from infeasible error paths, where the resultant interpolants are used to refine the abstract model. This ensures that the interpolants at the different control locations achieve the goal of providing a precision that eliminates the infeasible error path from further explorations.

CEGAR has been applied successfully in the context of programs [20], real time systems [21] and hybrid systems [22]. Moreover, CI-based CEGAR is standard in the domain of software model-checking, but current approaches are confined only to linear arithmetics or polynomials. However, what sets our method aside is that it applies Craig interpolation to learn concise reasons for a counterexample being spurious despite its rich, non-polynomial arithmetic domain as we

[11]http://www.cprover.org/svn/cbmc/releases/cbmc-4.8-incremental/

use iSAT as a backend solver where the latter handles the non-polynomiality by integrating CDCL(T) with ICP in one framework as aforementioned.

B. Handling of Interrupts

As the transfer project aims at covering analysis of handwritten embedded C code also, the ability to deal with the implicit transfer of control flow mediated by external interrupts is of importance, as interrupt handlers are a traditional means of ensuring reactivity to asynchronous events. The specific problem here is that in contrast to the control flow mediated by explicit control structures, interrupts can divert the control flow at any time. Analyzing such code by means of symbolic path enumeration in SMT-based bounded model checking would be prohibitively expensive due to the enormous branching width of the computation tree induced by unconstrained interrupts. To handle this problem, we are setting out to combine a number of techniques, the two most prominent of which are described in the following.

Semantics-preserving restriction of interrupt points: As interrupt handlers in embedded software tend to share only a small set of global variables with the main program, extensive assignment sinking or lifting is possible between statements in the main program and the interrupt routine. This implies that formally possible interrupt points can be eliminated from the analysis, as the execution sequences are confluent with those obtained from somewhat earlier or later interrupts. The analysis can thus w. l. o. g. confine control flow to interrupts happening at a limited subset of the factually possible interrupt points, e. g. just before the main program reads variables affected by the interrupt handler. This reduction reduces branching in the execution tree and helps SMT-based analysis.

Computation of summaries for interrupt routines: In order to accelerate analysis of the possible effect of interrupts on the main program's control flow (and thus, code reachability), we will take advantage of summaries for interrupt routines obtained using abstract interpretation and inter-procedural analysis. Abstract interpretation here is based on exploiting iSAT's floating-point and integer domains as abstract domains and iSAT's corresponding contractors as abstract transformers when analyzing straight-line code. Inter-procedural (or inter-interrupt-handler) analysis is then achieved by mutually using abstract interpretation on the individual interrupt handlers to compute safe approximations of the possible return values based on (over-approximate) knowledge of the possible entry values and on the main program to compute safe approximations of the possible entry values to interrupt handlers based on (over-approximate) knowledge of the possible return values.

The summaries thus obtained will later be exploited in the exact SMT-based analysis, where they can prune the search space tremendously before unfolding detailed statement sequences across interrupts.

C. Solver Improvements

Currently iSAT3 only supports floating-point reasoning for the basic arithmetic operations (Addition, Subtraction, Multiplication and Division). In the industrial context the support of more complex arithmetic operations such as *exp, sin, cos, sqrt* is mandatory and should be included in the current solver. Implementing these operations in iSAT3 might also have an advantage compared to bit-blasting solutions since in the latter context all those operations have to be translated into one big propositional formula while iSAT3 can handle them on the arithmetic level.

Alongside k-induction (CBMC) and Craig interpretation (iSAT3) there exist other techniques (e. g. IC3 [23], [24]) to prove the unreachability of code fragments. These may have some advantages over the currently used ones which may result in a larger number of successful unreachability proofs – or different instances which may complement and improve the results of the current techniques.

Symbolic Computation methods (such as Gröbner bases [25], virtual substitution [26], [27] or cylindrical algebraic decomposition (CAD) [28]) might also be beneficial in the context of floating-point reasoning. Especially for more complex arithmetic expressions these techniques might be able to (approximately) guide ICP during the search for a solution.

V. Conclusion

The AVACS transfer project T1 aims at a method for accurate dead code detection in embedded C code using arithmetic constraint solving. While in the target domain of reactive, control-oriented, and floating-point dominated embedded C programs several methods exist to deal with parts of the difficulties induced by this domain, there is none which can cover all of them. Thus a combination of these techniques as a solution seems obvious.

As a first step we integrated iSAT3 (with added support for floating-point arithmetic and Craig interpolation) into the tool chain of BTC Embedded Systems as an alternative backend solver. This allows us to perform code coverage analysis on automatically generated C code of industrial size including floating-point operations. The experimental results show, that iSAT3 is competitive in this environment – it is even able to prove more code fragments of interest as unreachable.

To tackle the remaining challenges of the transfer project there is work in progress concerning CEGAR and interrupt handling. Furthermore there are some ideas to improve the current method.

Acknowledgment

This work has been supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (DFG, SFB/TR 14 AVACS).

References

[1] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: ACM, 1977, pp. 238–252. [Online]. Available: http://doi.acm.org/10.1145/512950.512973

[2] ——, “Abstract interpretation and application to logic programs,” *The Journal of Logic Programming*, vol. 13, no. 2, pp. 103–179, 1992. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0743106692900307

[3] J. Julien Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival, “Static Analysis by Abstract Interpretation of Embedded Critical Software,” *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 1, pp. 1–8, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1921532.1921553

[4] P. Cousot, *Formal Verification by Abstract Interpretation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–7.

[5] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement,” in *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, ser. Lecture Notes in Computer Science, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Springer, 2000, pp. 154–169. [Online]. Available: http://dx.doi.org/10.1007/10722167_15

[6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided Abstraction Refinement for Symbolic Model Checking,” *J. ACM*, vol. 50, no. 5, pp. 752–794, Sep. 2003. [Online]. Available: http://doi.acm.org/10.1145/876638.876643

[7] R. Alur, T. Dang, and F. Ivancic, “Counter-Example Guided Predicate Abstraction for Hybrid Systems,” in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, H. Garavel and J. Hatcliff, Eds., vol. 2619. Springer, 2003.

[8] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert, “Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure,” *Journal on Satisfiability, Boolean Modeling, and Computation*, vol. 1, no. 3-4, pp. 209–236, 2007.

[9] J. G. Cleary, “Logical arithmetic,” *Future Computing Systems*, vol. 2, no. 2, pp. 125–149, 1987.

[10] H. Kelly J., V. Dan S., C. John J., and R. Leanna K., “A practical tutorial on modified condition/decision coverage,” Tech. Rep., 2001.

[11] C. Herde, “Efficient solving of large arithmetic constraint systems with complex boolean structure: proof engines for the analysis of hybrid discrete-continuous systems,” Ph.D. dissertation, 2011.

[12] F. Benhamou and L. Granvilliers, “Continuous and Interval Constraints,” in *Handbook of Constraint Programming*, ser. Foundations of Artificial Intelligence, 2006, pp. 571–603.

[13] M. Brain, V. D’Silva, L. Haller, A. Griggio, and D. Kroening, “An abstract interpretation of DPLL(T),” in *VMCAI 2013*.

[14] K. Scheibler, S. Kupferschmid, and B. Becker, “Recent improvements in the SMT solver iSAT,” in *MBMV*, 2013, pp. 231–241.

[15] K. Scheibler and B. Becker, “Using interval constraint propagation for pseudo-boolean constraint solving,” in *FMCAD 2014*.

[16] K. Scheibler, F. Neubauer, A. Mahdi, M. Fränzle, T. Teige, T. Bienmüller, D. Fehrer, and B. Becker, “Accurate ICP-based Floating-Point Reasoning,” in *Formal Methods in Computer-Aided Design, FMCAD 2016*. IEEE, 2016.

[17] M. Brain, V. D’Silva, A. Griggio, L. Haller, and D. Kroening, “Deciding floating-point logic with abstract conflict driven clause learning,” *Formal Methods in System Design*, vol. 45, no. 2, pp. 213–245, 2014.

[18] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller, “Successful use of incremental BMC in the automotive industry,” in *FMICS 2015*.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, “Lazy abstraction,” in *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, J. Launchbury and J. C. Mitchell, Eds. ACM, 2002, pp. 58–70. [Online]. Available: http://doi.acm.org/10.1145/503272.503279

[20] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, “The software model checker blast,” *STTT*, vol. 9, no. 5-6, pp. 505–525, 2007. [Online]. Available: http://dx.doi.org/10.1007/s10009-007-0044-z

[21] T. Nagaoka, K. Okano, and S. Kusumoto, “An abstraction refinement technique for timed automata based on counterexample-guided abstraction refinement loop,” *IEICE Transactions*, vol. 93-D, no. 5, pp. 994–1005, 2010. [Online]. Available: http://search.ieice.org/bin/summary.php?id=e93-d_5_994

[22] P. S. Duggirala and A. Tiwari, “Safety verification for linear systems,” in *Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada, September 29 - Oct. 4, 2013*. IEEE, 2013, pp. 7:1–7:10. [Online]. Available: http://dx.doi.org/10.1109/EMSOFT.2013.6658585

[23] A. R. Bradley, “Sat-based model checking without unrolling,” in *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, ser. Lecture Notes in Computer Science, R. Jhala and D. A. Schmidt, Eds., vol. 6538. Springer, 2011, pp. 70–87. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-18275-4_7

[24] M. Brain, V. D’Silva, A. Griggio, L. Haller, and D. Kroening, *Interpolation-Based Verification of Floating-Point Programs with Abstract CDCL*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 412–432. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38856-9_22

[25] T. Becker and V. Weispfenning, *Gröbner bases: a computational approach to commutative algebra*. London, UK: Springer-Verlag, 1993.

[26] V. Weispfenning, “The complexity of linear problems in fields,” *J. Symb. Comput.*, vol. 5, no. 1-2, pp. 3–27, Feb. 1988. [Online]. Available: http://dx.doi.org/10.1016/S0747-7171(88)80003-8

[27] ——, “Quantifier elimination for real algebra — the quadratic case and beyond,” *Applicable Algebra in Engineering, Communication and Computing*, vol. 8, no. 2, pp. 85–101, 1997. [Online]. Available: http://dx.doi.org/10.1007/s002000050055

[28] B. F. Caviness and J. R. Johnson, *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Vienna: Springer-Verlag, 1998.