A Fixed-Point Algorithm for Automated Static Detection of Infinite Loops

Andreas Ibing, Alexandra Mai
Chair for IT Security
TU München
Boltzmannstrasse 3, 85748 Garching, Germany
{*ibing,mai*}*@sec.in.tum.de*

***Abstract*—We present an algorithm for automated detection of infinite loop bugs in programs. It relies on a Satisfiability Modulo Theories (SMT) solver backend and can be run conveniently with SMT-constrained symbolic execution. The algorithm detects infinite loop bugs for single-path, multi-path and nested loops. We prove soundness of the algorithm, i.e. there are no false positive detections of infinite loops. Part of the algorithm is a fixed-point based termination check for 'simple' loops, whose soundness is a consequence of Brouwer's fixed-point theorem. The algorithm further yields no false negative detections for context-sensitive detection of periodic loop orbits with sum of prefix iterations and periodicity of up to the analysis loop unroll depth (bounded completeness), if the SMT solver answers the fixed-point satisfiability query in time. We describe an example implementation as plug-in extension of Eclipse CDT. The implementation is validated with the infinite loop test cases from the Juliet test suite and benchmarks are provided.**

***Keywords*-static analysis; loop (non-)termination; symbolic execution**

I. Introduction

For some applications an infinite loop is desired program behaviour. In the majority of cases however an infinite loop is not intended by the developer and therefore a bug, known under the common weakness enumeration [1] as CWE-835 ('loop with unreachable exit condition'). If infinite recursion can happen with regular program input, the infinite loop is a bug which prevents the program from functioning correctly. If an infinite loop can be triggered by unexpected input which is not properly validated, it can be used by an attacker for a denial of service attack.

Since loop behaviour may depend on the program input, the detection of infinite loop bugs with static analysis is attractive. Symbolic execution [2] is a path-sensitive static analysis method, where software input is regarded as variables (symbolic values). It is used to automatically explore different paths through software, and to compute path constraints as logical equations (from the operations with the symbolic input). An automatic theorem prover (constraint solver) is used to check program paths for satisfiability and to check error conditions for satisfiability. Symbolic execution has been used to automatically analyze source-code, intermediate code and binaries (machine code). It is typically applied by unrolling loops up to a certain depth as bounded model checking. An overview of symbolic execution techniques and available tools is given in [3], [4].

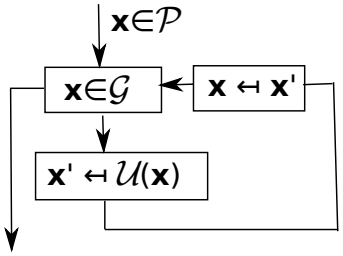

Figure 1. Single-path loop with stem path constraint $\mathcal{P}$, guard set $\mathcal{G}$, and loop update function $\mathcal{U}$.

The current state in automatic theorem proving are Satisfiability Modulo Theories (SMT) solvers [5]. An SMT solver contains a SAT solver for Boolean logic running the DPLL algorithm [6], and for linear arithmetic often also a theory solver running the simplex algorithm [7]. Current solvers typically also support some non-linear arithmetic with polynomial bases (Gröbner bases [8]). A standard SMT solver interface has been defined with the SMTlib [9]. An example of a state-of-the-art SMT solver is [10].

Proving that loops always terminate (loop termination) has long been a topic e.g. for Hoare-style verification calculus. Loop termination is typically proved by ranking function generation [11]. A 'guess and check' based approach for generating invariants of linear loops (single-path loops which only use linear arithmetic) is described in [12]. Since the Halting problem is undecidable in general, the complementary approach of detecting infinite loop bugs has also received attention. [13], [14] use a 'guess and check' pattern based algorithm to detect infinite loops. [14] notes the equivalence to invariant generation like in [12].

This paper presents a detection algorithm for infinite loops based on fixed-point analysis. Loops which terminate with a number overflow or loops which can run infinitely only as a consequence of a previous number overflow are not detected, as the common weakness enumeration rather classifies such bugs as 'number overflows'. The remainder of the paper is organized as follows. The following section II gives the problem formulation. Section III describes the algorithm. In section IV an example implementation as extension of

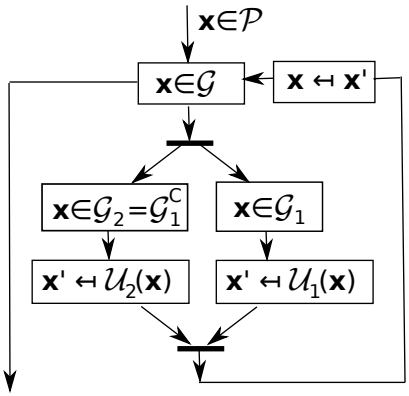

Figure 2. Example multi-path loop, corresponding to an `If/Else` branch after the loop guard.

a symbolic execution engine for Eclipse CDT is presented. Section V provides an evaluation of the implementation with test programs from the Juliet suite [15], [16]. Section VI provides a soundness proof and also proves bounded completeness (if the solver answers the fixed-point satisfiability query in time). Related work is discussed in section VII, and section VIII disusses the results.

II. Problem formulation

Loops can be categorized in single-path loops, multi-path and nested loops. The loop model is illustrated for a single-path loop in figure 1. The figure shows the vector $\mathbf{x}$ of loop variables, a path constraint $\mathcal{P}$ on which the loop is reached from the program entry point, the loop guard set $\mathcal{G}$ and the loop update function $\mathcal{U}(\mathbf{x})$. The vector $\mathbf{x}'$ denotes the loop variables after update, so that one loop update computes

$$\mathbf{x}' = \mathcal{U}(\mathbf{x})$$

The problem we are interested in is whether the program contains a loop which runs infinitely for any program input. This includes the question whether the loop can be reached on a satisfiable path with this input.

A loop may run a different number of iterations for different input. This means that it computes different functions for different sets of input. We consider the sequence $(\mathbf{x}^{(k)})_{k\in\mathbb{N}}$ of values of $\mathbf{x}$ for each iteration and denote it as the orbit of $\mathbf{x}$. The loop effect is described with function composition of the update functions of different iterations. If there is input for the single-path loop from figure 1 for which it runs n iterations, then it computes the orbit:

$$\begin{aligned}\mathbf{x} = \mathbf{x}^{(0)} \quad &\mapsto \quad \mathbf{x}^{(1)} = \mathcal{U}(\mathbf{x}^{(0)}) \\ &\mapsto \quad \mathbf{x}^{(2)} = \mathcal{U}(\mathbf{x}^{(1)}) = \mathcal{U}^2(\mathbf{x}^{(0)}) \\ &\mapsto \quad \dots \\ &\mapsto \quad \mathbf{x}^{(n)} = \mathcal{U}^n(\mathbf{x})\end{aligned}$$

A multi-path loop contains different loop update functions for each path. A two-path loop is illustrated in figure 2.

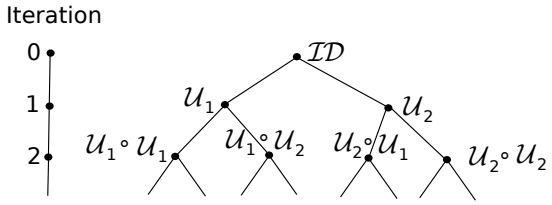

Figure 3. Transformation monoid corresponding to two-path loop from figure 2.

Different paths through the loop may be taken for different input sets in each iteration. The possible compositions of update functions form a transformation monoid, which is illustrated in figure 3. If the loop is not taken (0 iterations), this corresponds to the identity mapping $\mathcal{ID}$ of $\mathbf{x}$. For one iteration the resulting function is either $\mathcal{U}_1$ or $\mathcal{U}_2$. For a two-path loop (e.g. an `If/Else` statement within a loop) the transformation monoid is a binary tree. For more than two paths the illustration would be a Cayley tree. The problem is to detect whether a loop can be reached with adequate input so that an infinite orbit is satisfied.

III. Algorithm

Standard programming language number formats are discrete and finite, so that an infinite orbit (without overflow) must be a periodic orbit, i.e.

$$\mathbf{x}^{(t+p)} = \mathbf{x}^{(t)} \quad \text{for some } p > 0,\ t \geq 0$$

The algorithm performs two types of tests. The first type is a context-free analysis to decide whether 'simple' loops terminate for all input, or whether they are infinite for all input. The context-free analysis is performed only once per loop. The second type performs context-sensitive non-termination analysis of the loops which have not yet been decided in the first step. Both types are run during symbolic execution. The checker method for context-free checks is given as pseudo-code in algorithm 1. The method for context-sensitive non-termination check is depicted as pseudo-code in algorithm 2.

A. Context-free termination check

This check is performed for single-path loops with linear arithmetic and convex guard set, which do not contain the modulo operator (which would destroy the precondition of Brouwer's theorem).

The loop guard can be expressed in matrix notation as:

$$G\mathbf{x} \leq \mathbf{g}, \qquad G \in \mathbb{R}^{k\times n},\ \mathbf{g} \in \mathbb{R}^n$$

The loop guard is a convex set if it consists of conjunctively connected linear inequalities (no disjunctions). The loop update function is then:

$$\mathbf{x}' = \mathcal{U}(\mathbf{x}) = U\mathbf{x} + \mathbf{u}, \qquad U \in \mathbb{R}^{n\times n},\ \mathbf{x}, \mathbf{x}', \mathbf{u} \in \mathbb{R}^n$$

The loop always terminates if the orbit leaves the guard set for all possible input vectors $\{\mathbf{x} \mid G\mathbf{x} \leq \mathbf{g}\}$ after a finite number of iterations. In section VI it is explained in detail that as a consequence of Brouwer's fixed-point theorem [17] there can only be a periodic orbit if the loop update function $\mathcal{U}(\mathbf{x})$ contains a fixed-point $\mathbf{x}^{(1)} = \mathbf{x}^{(0)}$ within the guard set:

$$\begin{aligned} & G\mathbf{x} \leq \mathbf{g} \\ \wedge \quad & \mathbf{x}' = U\mathbf{x} + \mathbf{u} \\ \wedge \quad & \mathbf{x}' = \mathbf{x} \end{aligned}$$

If the constraint solver answers `'unsat'` to this query, the loop is marked as terminating.

B. Context-free non-termination check

This check is performed for single-path loops (possibly non-linear). If the loop was already decided to always terminate, then this check is not performed. The loop is non-terminating for all input if:

$$\begin{aligned} & \forall(\mathbf{x} \in \mathcal{G}).\ \mathcal{U}(\mathbf{x}) \in \mathcal{G} \\ \Leftrightarrow \quad & \neg\exists(\mathbf{x}_i \in \mathcal{G}).\ \mathcal{U}(\mathbf{x}_i) \notin \mathcal{G} \end{aligned}$$

The constraint solver is therefore asked to check satisfiability of:

$$\mathbf{x} \in \mathcal{G} \ \wedge \ \mathbf{x}' = \mathcal{U}(\mathbf{x}) \ \wedge \ \neg(\mathbf{x}' \in \mathcal{G})$$

If the solver answers `'unsat'` to this query, the loop is marked as non-terminating.

```
checkContextFree(CFNode node, SymVars x, SymVars x')
if (isLinearNoModulo(node)) then
    // 1. check terminating:
    s1 = check-sat(Gx ≤ g ∧ x' = Ux + u ∧ x' = x);
    if (s1 == unsat) then
        markTerminating(node);
        return;
    end
end
if (isSinglePath(node)) then
    // 2. check infinite:
    s2 =
    check-sat(x ∈ G ∧ x' = U(x) ∧ ¬(x' ∈ G));
    if (s2 == unsat) then
        markNonTerminating(node);
        return;
    end
end
```

Algorithm 1: Context-free termination and non-termination checks for 'simple' loops.

C. Context-sensitive non-termination analysis of remaining loops

Loops which have not been decided by the context-free analysis are analysed context-sensitively during symbolic execution. Such loops are e.g. multi-path loops or loops which terminate only for a partial input set. A loop is reached with symbolic execution from a program entry point on a path which we call stem path (as in [14]). If the loop guard is satisfiable for the stem path context, then symbolic execution unrolls the loop once and checks guard satisfiability again. It is detected that a loop is closed when the same decision node on the path is revisited. If the guard condition is satisfiable also after unrolling the loop body, then the loop variables are identified and the non-termination analysis algorithm is called. If the same decision node is already several times on the path, then several loops are closed at the same time. The algorithm is the run for each of the closed loops. The algorithm searches for a satisfiable periodic orbit in the respective composite function of the transformation monoid which corresponds to the taken loop unrolling branches. The algorithm checks the existence of a fixed-point in the unrolled loop. A fixed-point of e.g. $\mathcal{U}_1^4$ is a periodic orbit of $\mathcal{U}_1$ with periodicity 4. A fixed-point of an n-fold function composition from the transformation monoid is a periodic orbit through the loop with periodicity $p = n$. The loop body path constraint contains the constraints for correct domains of the respective update functions. By checking for fixed-points for all loop closed events on the current path, also periodic orbits with t prefix iterations and periodicity $p = n - t$ are found.

```
checkContextSens(CFNode node, SymVars x, SymVars x')
P(x) = getPathConstraint();
L(x') = getUnrolledLoopConstraint();
s = check-sat(P(x) ∧ L(x') ∧ (x' == x));
if (s == sat) then
    reportError(node, 'infinite Loop');
    return;
else
    return;
end
```

Algorithm 2: Context-sensitive non-termination check for remaining loops.

IV. Implementation in Eclipse CDT

A. Integration with symbolic execution

For an example implementation we extend [18], which is a symbolic execution engine as plug-in extension to Eclipse CDT. It uses the CDT parser and the control flow graph (CFG) builder from CDT's code analysis framework (Codan [19]). Path satisfiability and bug conditions are

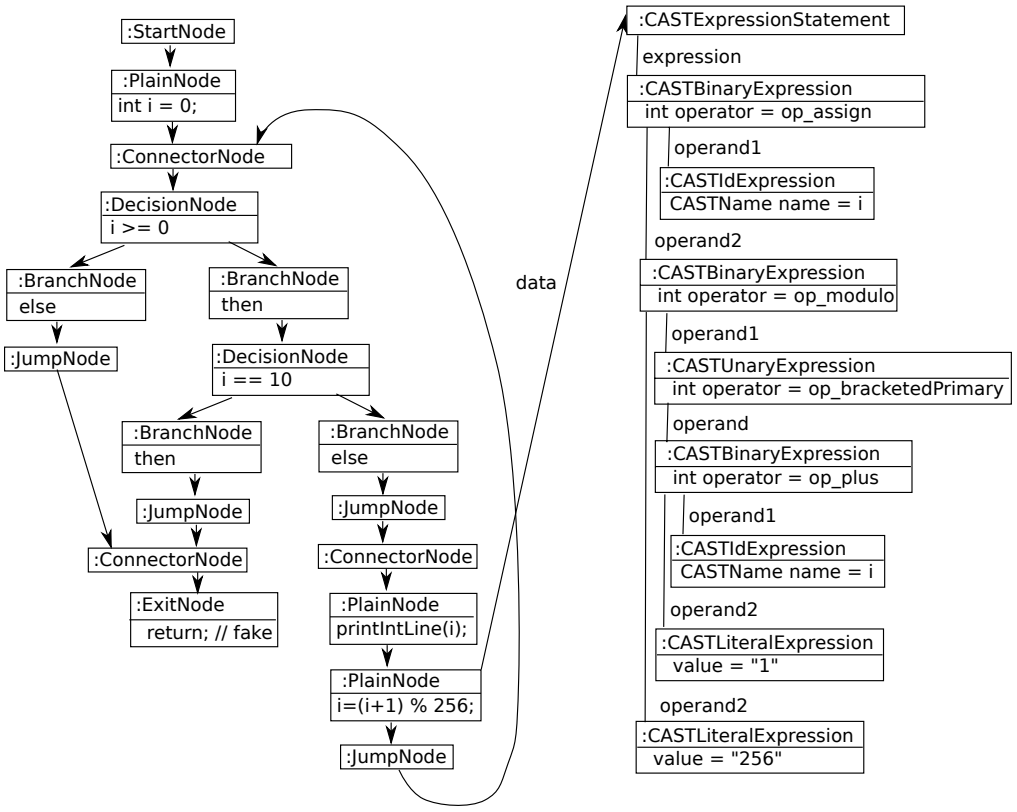

Figure 4. Control flow graph of the function in listing 1 and part of the AST subtree referenced by a plain node.

checked in [18] with an SMT solver in the SMTLIB logic of arrrays, uninterpreted functions and nonlinear integer and real arithmetic (AUFNIRA). The symbolic execution engine unrolls loops up to a configurable loop depth bound. It was successfully applied in [18] to detect buffer overflow bugs in a test suite for static analyzers.

We implemented the infinite loop bug detection algorithm in a new checker class. The engine was extended to detect loop closed events, to identify loop variables and to trigger the infinite loop checker during symbolic execution. Differing from [18] here we use [10] as SMT solver backend. Analysis with symbolic execution starts by parsing the source files into abstract syntax trees (ASTs) and constructing CFGs for AST subtrees rooted in a function definition. As example consider the C function in listing 1. The function contains a multi-path loop which terminates for all input. Figure 4 shows the function's CFG and part of the AST subtree referenced by a plain node of the graph.

Listing 1. Example from [16]. The function contains a multi-path loop which terminates for all input. The corresponding CFG is illustrated in figure 4.

```
static void good1(){
    int i = 0;
    while(i >= 0) {
        // FIX: Add a break point if i=10
        if (i == 10) {
            break;
        }
        printIntLine(i);
        i = (i + 1) % 256;
    }
}
```

B. Loop detection and identification of loop variables

The engine [18] keeps the program path in memory which is symbolically interpreted at the moment. 'Loop closed' events are detected during symbolic execution with bounded path coverage, where loops are unrolled at least once. When an already visited decision node (i.e. which is on the current path) is visited again, the algorithm to test for an infinite loop is called. The path is split into stem and loop body at each previous occurance of the loop decision node. The engine also contains a class `Actionlog` which tracks during interpretation for which CFG node which new symbolic variables are created. This `Actionlog` is used to identify the loop variables x_i' from the loop body and the corresponding x_i from the stem as their previous values.

C. Loop property determination

The properties of interest are whether the loop is linear with convex guard set, whether it contains the modulo

operator and whether it is a single-path (i.e., not multi-path or nested) loop. These properties are determined with a class `LoopPropertyFinder` which extends CDT's `ASTVisitor`. The AST subtree corresponding to the loop compound statement is accessed through a reference from the decision node in the CFG. The loop properties are determined by traversing this AST subtree according to the visitor pattern [20]. The property determination needs to run only once per loop.

D. Context-free termination and non-termination loop tests

The context-free tests are performed only once per loop, when the loop decision node is revisited for the first time. Stem path equations are not included in the equation system, which removes the context of the loop. The remaining constraints are the guard condition before the loop, the loop update and the guard condition after the update. Any re-computations later when the loop is reached with a different path constraint would therefore be redundant. The algorithm generates an additional constraint for termination and non-termination checks respectively and tests equation system satisfiable with the SMT solver (compare algorithm 1).

E. Test for periodic orbit with path constraint

If a loop is closed during symbolic execution which has not already been marked as terminating or non-terminating, then the path-sensitive non-termination check method (algorithm 2) is called. In addition to the stem path constraint and loop body constraint which are queried from the symbolic execution engine, the algorithm generates an additional fixed-point constraint and tests equation system satisfiable with the SMT solver. The fixed-point equations for the loop variables are in SMTLIB notation:

$$\begin{aligned} &\texttt{(assert (=}\ x_1\texttt{'}\ x_1\ \texttt{))} \\ &\qquad\qquad \ldots \\ &\texttt{(assert (=}\ x_n\texttt{'}\ x_n\ \texttt{))} \end{aligned}$$

where the x_i' and x_i are replaced with the actual loop variable names. If the solver answers SAT, an infinite loop is detected and reported. Corresponding input triggering the infinite loop bug can be queried from the solver with the SMTLIB `'get-value'` command.

V. Experiments

We validate the implementation with the infinite loop test programs from the Juliet test suite for static analyzers [15], [16]. The small test programs combine common software weaknesses systematically with different control and data flow variants. They contain both 'good' and 'bad' (flawed) functions in order to measure a tool's false negative and false positive detections. An example 'good' function has been given in listing 1. The multi-path loop is unrolled up to ten times during analysis, at which point termination is proven. Two example 'bad' functions are given in listings 2 and 3. Listing 2 contains a single-path empty for-loop which is non-terminating for all input. The loop guard is simply 'true', and the bug is detected during the context-free non-termination check. Listing 3 contains another single-path for-loop which is non-terminating for all input and is decided with the context-free test. The context-free termination check is not performed in this case because the loop contains a modulo operator. The test programs are analyzed in Eclipse 4.3 on 64-bit Linux kernel 3.2.0 on a Core i7-4770 CPU. The symbolic execution engine is configured to run single-threaded. The tests are run as JUnit plug-in tests with the Eclipse GUI. Results are shown in figure 5. All 6 test programs are correctly decided within less than three seconds each. A screenshot of error reporting with the Codan framework is shown in figure 6.

Listing 2. Example from [16]. Single-path empty for-loop which is non-terminating for all input.

```
void CWE835_Infinite_Loop_for_empty_01_bad(){
    int i = 0;
    // FLAW: Infinite Loop
    for(;;) {
        printIntLine(i);
        i++;
    }
}
```

Listing 3. Another example from [16]. The function contains a single-path non-terminating loop.

```
void CWE835_Infinite_Loop__for_01_bad() {
    int i = 0;
    // FLAW: Infinite Loop
    for (i = 0; i >= 0; i = (i + 1) % 256) {
        printIntLine(i);
    }
}
```

VI. Algorithm Properties

This section assumes that the SMT solver gives the correct `'sat'` or `'unsat'` answer to each equation system query within the available computation time.

A. Soundness

Proposition 1. *The algorithm does not compute any false positive detections.*

Proof: An infinite loop is reported in two cases:

Case 1. *by the context-free check if:*

$$\textit{check-sat}\Big(\mathcal{G}(\mathbf{x}) \wedge (\mathbf{x}' = \mathcal{U}(\mathbf{x})) \wedge \neg(\mathcal{G}(\mathbf{x}'))\Big) == \texttt{unsat}$$

i.e. there exists no loop input $\mathbf{x}_i \in \mathcal{G}(\mathbf{x})$ for which the next iteration is not also taken. This means that the loop is infinite if it can be reached with any input. The check function is only called if the loop is reached with satisfiable input during

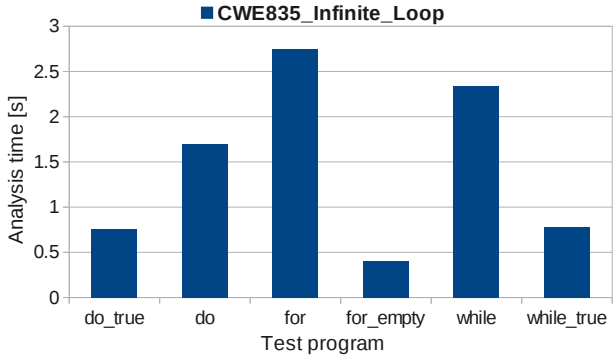

Figure 5. Analysis time for infinite loop test programs from [16].

symbolic execution. The program therefore indeed contains a reachable infinite loop.

Case 2. *by the context-sensitive check if:*

$$\textit{check-sat}\Big(\mathcal{P}(\mathbf{x}) \wedge \mathcal{L}(\mathbf{x}') \wedge (\mathbf{x}' == \mathbf{x})\Big) == \mathit{sat}$$

i.e. there is a program input for which the loop is reached with $\mathcal{P}(\mathbf{x}_i)$, so that loop guard and an unrolled loop body are satisfiable and have a fixed-point $\mathcal{L}(\mathbf{x}_i)$. The program therefore indeed contains a reachable infinite loop.

■

B. Bounded completeness

In order to prove bounded completeness, it must first be ensured that the context-free termination check never wrongly marks a loop as terminating if the loop does not always terminate. Otherwise this would prevent the context-sensitive non-termination check from being performed and would lead to false negative detections. This means:

Proposition 2. *The context-free termination check is sound, i.e. when a context-free single-path loop with convex guard set and linear update function without modulo operator does not have a fixed-point, then there is no infinite orbit through it with any stem path constraint.*

Proof: By contradiction. The proof is a minor adaptation from [21]. Assume there is a periodic orbit $(\mathbf{x}^{(k)})$, $k \in \mathbb{N}_0$, through the loop, i.e. $\forall k.\ \mathbf{x}^{(k)} \in \mathcal{G}.\ \wedge\ \mathbf{x}^{(t+p)} = \mathbf{x}^{(t)}$. Because all possible combinations of stem and loop body with $t + p = k$ are tested, we can without loss of generality assume $t = 0$, i.e. k loop iterations belong to the unrolled loop body and there are no prefix iterations. Let $\mathcal{V}$ be the affine space of the lowest dimension that contains the orbit. By intersecting $\mathcal{G}$ and $\mathcal{V}$ we get a compact convex set $\mathcal{G} \cap \mathcal{V}$. Furthermore $\mathbf{x}^{(k)} \in \mathcal{G} \cap \mathcal{V};\ 0 \leq k \leq p$. We build the convex hull $\mathcal{X} = \text{Conv}(x^{(k)})$. Then $\mathcal{X} \subset \mathcal{V} \cap \mathcal{G}$ and $\forall \mathbf{x}_i \in \mathcal{X}, \forall q \in \mathbb{N} : \mathcal{U}^q(\mathbf{x}_i) \in \mathcal{X}$, i.e. $\mathcal{X}$ is a 'positively invariant set'. The closure $\overline{\mathcal{X}}$ is a compact convex positively invariant set in $\mathcal{V}$. Now we can apply the Brouwer fixed-point theorem [17] on the loop update function $\mathcal{U} \colon \overline{\mathcal{X}} \to \overline{\mathcal{X}}$. It guarantees us the existense of a fixed-point in $\mathcal{U}$. ■

Now bounded completeness can be considered:

Proposition 3. *When all loops are unrolled up to a depth of n, then no infinite loop bug with t prefix iterations and an orbit periodicity of $p \leq n - t$ is missed.*

Proof: Unrolling all loops up to a depth of n (if satisfiable) means building all composition functions of the loop's transformation monoid up to level n. Any orbit with periodicity $p \leq n-t$ is a fixed-point of a composite function from the transformation monoid with level $l \leq n - t$ and is therefore not missed. ■

VII. Related Work

A large body of related work is available in the area of loop (non-)termination proving, especially [11], [22], [23], [13], [14], [24], [25], [26]. The special case of (non-nested) single-path loops with linear operations and convex guard set (i.e. linear loops) is dealt with in [11], [22], [23], [25], [26]. In [11] a complete algorithm for termination proving by synthesis of linear ranking functions is given. It is complete with respect to the existence of linear ranking functions, and it notes that there are terminating linear loops without existing linear ranking function. In [22] linear loops with real arithmetic are considered. It is shown that termination is decidable, and a 'guess and check' non-deterministic algorithm for termination is given. It uses a transformation of the loop update matrix to Jordan normal form and the introduction and constraining of an extra variable to homogenize an inhomogeneous problem (where the affine loop update mapping contains a non-zero translation). [22] also shows that termination becomes undecidable for multi-path loops. In [23] linear loops with integer arithmetic are considered. It is shown that some loops may terminate over integers, but not over reals. It generalizes [22] and shows that homogeneous linear loops with integer arithmetic are decidable. In [26] necessary and sufficient conditions for the termination of linear homogeneous loops are given. In

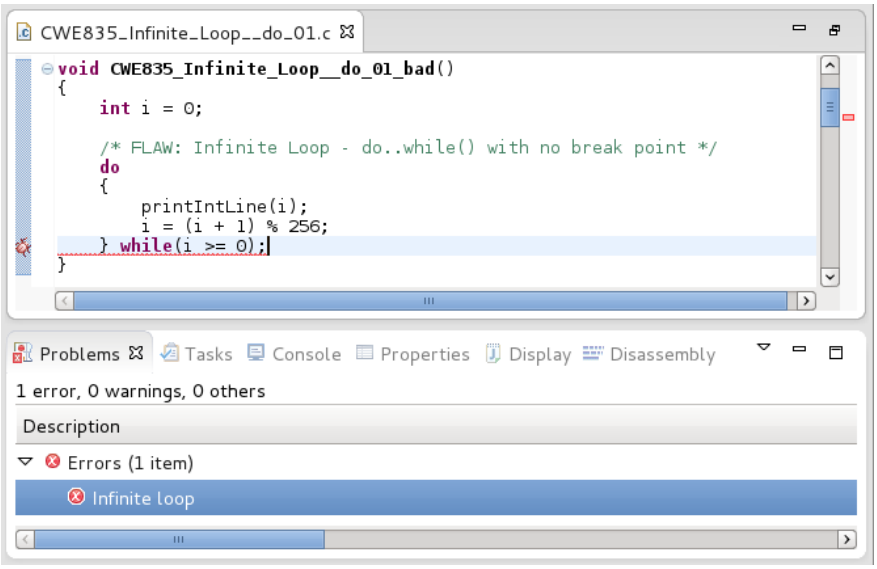

Figure 6. The algorithm is implemented as extension of [18] and uses the CDT/Codan error marker framework for reporting.

[25] a decision procedure for inhomogeneous linear loops with integer arithmetic and diagonalizable update matrix is presented. In [13], [14], [24] algorithms for the detection of non-terminating nested and multi-path loops are presented. In [13] and [14] loops are unrolled into all satisfiable paths while testing non-termination with a template-based 'guess and check' invariant generation algorithm. A low-complexity algorithm to detect infinite loops at runtime is presented in [24].

The proposed approach differs from these works as follows. A minor difference is that these works detect loops which run infinitely with arbitrary-precision numbers. This means that they also mark loops as 'infinite' which in fact terminate with a number overflow on any real machine. According to the common weakness enumeration such errors would be marked as number overflow, while only loops infinite on real machines are marked as infinite loops. The proposed algorithm sticks to the CWE terminology without false positives. This difference is however minor because the available algorithms could be applied to the problem considered in this paper by applying an additional test to the spotted loops to check whether or not loop variables stay within valid range. The presented context-free termination check differs from [11] first in that Brouwer's theorem holds independent of the existence of a linear ranking function (compare Sec. 6 Proposition 2), and second in that it would also hold for non-linear but compact convex guard sets (e.g. in SMTlib sublogic NRA). It differs from [22] in that reasoning with a fixed-point is simpler than performing homogenization and transformation to Jordan normal form. Brouwer's theorem is in fact mentioned in [22], but it is used quite differently (to prove existence of a certain Eigenvector in the homogeneous case). The presented algorithm for detection of infinite loops differs from the one presented in [24] because in [24] dynamic analysis is used, so that coverage depends on the available test set. The proposed algorithm is static, it can be run with arbitrary coverage (e.g. bounded path coverage) independent of any test set. In relation to the algorithm given in [14] the proposed context-sensitive non-termination check is as a special case of the search for recurrent sets, by constraining the recurrent set to be a (fixed-)point. Speed-ups are achieved both by combining termination and non-termination checking, and by the context-free checks which avoid re-checking simple loops.

VIII. Discussion

The presented algorithm is sound and context-sensitively detects infinite loop bugs in single-path, multi-path and nested loops. The number of satisfiable leaf functions of the loop's transformation monoid grows exponentially in the worst case. This limits the unrolling depth and therefore also the periodicity of detectable infinite loop orbits in practice. Integration with a symbolic execution engine is described. Since the algorithm is implemented in [18], in runs together with other bug checkers (including a buffer overflow checker [18]) sharing the contexts provided by the symbolic execution engine. The implementation has only been evaluated on the test programs from [16] so far and not on production applications. On the other hand the key algorithm properties of soundness and of bounded completeness have been proven.

Acknowledgement

This work was funded by the German Ministry for Education and Research (BMBF) under grant 01IS13020.

REFERENCES

[1] R. Martin, S. Barnum, and S. Christey, "Being explicit about security weaknesses," in *Blackhat DC*, 2007. [Online]. Available: http://cwe.mitre.org/documents/being-explicit/BlackHat_BeingExplicit_WP.pdf

[2] J. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[3] C. Cadar, K. Sen, P. Godefroid, N. Tillmann, S. Khurshid, W. Visser, and C. Pasareanu, "Symbolic execution for software testing in practice – preliminary assessment," in *Int. Conf. Software Eng.*, 2011.

[4] C. Pasareanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *Int. J. Software Tools Technology Transfer*, vol. 11, pp. 339–353, 2009.

[5] L. deMoura and N. Bjorner, "Satisfiability modulo theories: Introduction and applications," *Communications of the ACM*, vol. 54, no. 9, 2011.

[6] M. Davis, G. Logeman, and D. Loveland, "A machine program for theorem proving," *Communications of the ACM*, vol. 5, no. 7, 1962.

[7] G. Dantzig, "Maximization of a linear function of variables subject to linear inequalities," 1949, in T. Koopmans: "Activity Analysis of Production and Allocation", 1951 Wiley & Chapman-Hall, pp. 339-347.

[8] B. Buchberger, "Theoretical basis for the reduction of polynomials to canonical forms," *ACM SIGSAM Bull.*, vol. 10, no. 3, 1976.

[9] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard version 2.0," in *Int. Workshop Satisfiability Modulo Theories*, 2010.

[10] L. deMoura and N. Bjorner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[11] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," in *Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2004.

[12] M. Colon, S. Sankaranarayanan, and H. Sipma, "Linear invariant generation using non-linear constraint solving," in *Int. Conf. Computer Aided Verification (CAV)*, 2003.

[13] H. Velroyen and P. Rummer, "Non-termination checking for imperative programs," in *Tests and Proofs (TAP)*, 2008.

[14] A. Gupta, A. Rybalchenko, T. Henzinger, R. Xu, and R. Majumdar, "Proving non-termination," in *Symp. Principles of Programming Languages (POPL)*, 2008.

[15] T. Boland and P. Black, "Juliet 1.1 C/C++ and Java test suite," *IEEE Computer*, vol. 45, no. 10, 2012.

[16] *Juliet Test Suite v1.2 for C/C++*, United States National Security Agency, Center for Assured Software, May 2013. [Online]. Available: http://samate.nist.gov/SARD/testsuites/juliet/Juliet_Test_Suite_v1.2_for_C_Cpp.zip

[17] L. Brouwer, "Über Abbildungen von Mannigfaltigkeiten," *Mathematische Annalen*, no. 71, 1911.

[18] A. Ibing, "Parallel SMT-constrained symbolic execution for Eclipse CDT/Codan," in *Int. Conf. Testing Software and Systems*, 2013.

[19] A. Laskavaia, "Codan- C/C++ static analysis framework for CDT," in *EclipseCon*, 2011.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[21] L. Habets, P. Collins, and J. Schuppen, "Reachability and control synthesis for piecewise-affine hybrid systems on simplices," *IEEE Trans. Automatic Control*, vol. 51, no. 6, 2006.

[22] A. Tiwari, "Termination of linear loops," in *Int. Conf. Computer Aided Verification (CAV)*, 2004.

[23] M. Braverman, "Termination of integer linear programs," in *Int. Conf. Computer Aided Verification (CAV)*, 2006.

[24] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen, "Looper: Lightweight detection of infinite loops at runtime," in *Int. Conf. Automated Software Engineering*, 2009.

[25] J. Ouaknine, J. Pinto, and J. Worrell, "On termination of integer linear loops," 2014. [Online]. Available: http://arxiv.org/abs/1407.1891v1

[26] R. Rebiha, A. Moura, and N. Matringe, "Characterization of termination of linear homogeneous programs," Tech Report, Universidade Estadual de Campinas, 2014. [Online]. Available: http://www.ic.unicamp.br/~reltech/2014/14-08.pdf