

Finding Inconsistencies in Programs with Loops^{*}

Temesghen Kahsai¹, Jorge A. Navas², Dejan Jovanović³, Martin Schäfer³

¹ Carnegie Mellon University, USA

² NASA Ames Research Center / SGT, USA

³ SRI International, USA

Abstract. Inconsistent code is an important class of program abnormalities that appears in real-world code bases and often reveals serious bugs. A piece of code is inconsistent if it is not part of any safely terminating execution. Existing approaches to inconsistent code detection scale to programs with millions of lines of code, and have led to patches in applications like the web-server Tomcat or the Linux kernel. However, the ability of existing tools to detect inconsistencies is limited by gross over-approximation of looping control-flow. We present a novel approach to inconsistent code detection that can reason about programs with loops without compromising precision. To that end, by leveraging recent advances in software model checking and Horn clause solving, we demonstrate how to encode the problem as a sequence of Horn clause queries enabling us to detect inconsistencies that were previously unattainable.

1 Introduction

Static analysis techniques are notoriously plagued by false alarms. In an effort to build a static analyzer that report close to zero false alarms, we have seen an increasing interest in inconsistent code detection. Broadly speaking, inconsistent code comprises code where two program locations make contradicting assumptions about the execution of the program. This includes, for example, checking if a chunk of memory is properly allocated only after it has already been dereferenced, or accessing an array at an index that is guaranteed to be out of bounds.

Formally, inconsistent code is defined as a program location that only occurs on executions that must reach an error state. In other words, a code fragment is said to be inconsistent if it is never part of a “normal” execution of the program. For example, unreachable code is inconsistent because it has no execution. Previous techniques have demonstrated that inconsistent code is a very practical methodology to find likely bugs in a scalable fashion. For example, in [11] inconsistent code is used to reveal bugs in the Linux kernel, and in [27] the authors found inconsistent code in the web-server Tomcat and the project management Maven.

^{*} This material is based upon work supported by the National Science Foundation under Grant No. 1422705, AFRL contract No. FA8750-15-C-0010, and NASA contract NNX14AI05A.

Inconsistent code detection algorithms (e.g., [11,18,22,27,34]) share the same basic architecture. They analyze a program one procedure at a time. For each procedure, they over-approximate the feasible executions and try to enumerate the feasible control-flow paths. Everything that cannot be covered is provably inconsistent code.

So far, all implementations that detect inconsistent code or subsets of it use very coarse abstractions to handle looping control-flow which limits their ability to detect inconsistencies significantly. All of these approaches over-approximate the effect of loops, by simply replacing them with non-deterministic assignments to the variables modified inside the loop body. Some of the approaches additionally add an unwinding of the last loop iteration (to detect typical off-by-one errors).

In this paper, we present a novel algorithm to detect inconsistent code that is able to reason more efficiently about looping control-flow. To that end, we follow the recent trend and reformulate the problem of detecting inconsistencies as a problem of solving a system of constrained Horn clauses (CHC). Instead of unwinding looping control-flow, we allow for recursive Horn clause definitions. As in previous approaches that detect inconsistencies, we encode programs into logic such that a model for this Horn clause system can be mapped to a feasible control-flow path in the program, but, unlike existing approaches, our encoding does not require loop elimination.

Each time we find such a feasible path, we block it and check for the existence of another model that exercises a different control-flow path. However, since our Horn clause definitions may be recursive, enumerating all feasible path may not be possible as there may be infinitely many. For this case, we make use of the recent developments in software model checking for solving Horn clause systems which may be able to prove unsatisfiability by inferring local invariants using, for example, property-directed reachability (PDR)/IC3 [7]. For cases where such a proof exists, our analysis can find inconsistencies that existing tools could not find. For cases where such a proof does not exist, we can still fall back to previous approaches by abstracting the looping control-flow.

As a side effect, the invariants produced by our Horn clause solver can be used to implement existing fault localization techniques for inconsistent code [32].

We evaluate our approach on a set of handcrafted problems which we made available on-line. In our experiments, our approach only times out in a single case and finds several inconsistencies that cannot be detected with current tools that checks for inconsistency.

2 Related Work

The idea that code inconsistencies represent an interesting class of possible defects goes back to Engler et al. [11]. Their technique to detect inconsistencies was mostly based on syntactic comparison but it is already able to find bugs in the Linux kernel and other major pieces of software. The work by Dillig et al. [10] uses the term *semantic inconsistencies* to refer to contradicting assumptions on

control-flow paths. While their work also detects inconsistencies as defined by Engler et al., they also include inconsistencies on individual paths (even though each statement on these paths might have a feasible execution). That is, they scan for a larger class of errors but introduce possible false alarms as they cannot guarantee that the inconsistent paths they report are in fact feasible in a larger context.

The idea of using deductive verification to prove inconsistencies has been presented in [22], [34], and [18]. Janota et al. [22] use a variation of the Boogie tool [2] to verify that code is unreachable in an annotated program. This is only a subset of inconsistent code, but the detection algorithm could easily be extended to detect inconsistent code. Hoenicke et al. [18] prove the existence of inconsistent code but use the term *doomed program points*. Tomb et al. [34] use a very similar approach and also give a definition of inconsistent code that we are going to reuse in this paper. In our earlier work, we have developed a tool to detect inconsistent code [27] and demonstrated that it finds relevant bugs in popular open-source Java applications.

In [6], the authors use inconsistency detection to prioritize error messages produced by a static analyzer. Their approach post-processes static analysis warnings and gives them a high priority if the warning contains an *abstract semantic inconsistency bug*, which is inconsistent code on an abstract model of the code.

An approach that is similar in spirit but not immediately related is the work by Wang et al. [35] where the authors try to identify local invariants to detect undefined behavior of C programs. While the class of errors that we want to detect are not immediately comparable, we share the idea of searching for invariants to prove the presence of errors while accepting false negatives in return for a low false positive rate.

The local invariants computed by our approach when proving code to be inconsistent can be seen as *error invariants* [12] which can be used for fault localization. The approach presented in [32] shows how these invariants can be used to explain inconsistent code.

Constraint Horn clauses have been used as the basis for software model checking [9,13] of concurrent systems and its use in software verification tools is rapidly growing. For example, they have been adopted in Threader [29], UFO [1], SeaHorn [16], HSF [14], VeriMAP [8], Eldarica [31], and TRACER [21]. Our tool has many similarities with some of these tools and in fact, our current implementation is built on the top of SeaHorn. However, ours is the first available implementation based on Horn clauses that detects inconsistent code.

3 Running Example

We illustrate the different steps of our approach along the running code example in Figure 1. The procedure `foo` takes an integer `x` as input and computes the sum of $10/i$, for all `i` between `-x` and `x`. That is, for any `x` less or equal to zero, the procedure skips the loop and returns 0 immediately. For any `x` greater

zero, however, the procedure will perform a division by zero once the iterator i becomes zero resulting in undefined behavior. Executing this undefined behavior causes the program to terminate with an exception (when compiled with gcc). Since the loop is iterating from $-x$ to x , any execution that enters the loop must raise this exception. Hence, line 4 is *inconsistent code*. We acknowledge that this is a fairly artificial example but it is designed so that its Horn clause representation and the invariants used to prove the inconsistency are succinct for presentation reasons.

```

1  int foo(int x) {
2      int ret = 0;
3      for (int i=-x; i<x; i++) {
4          ret += 10/i;
5      }
6      return ret;
7  }

```

Fig. 1. Illustrative example. The procedure `foo` takes an integer x as input and sums up the integer divisions $10/i$ for all i between $-x$ and x . For any $x > 0$, the division in line 4 must raise an exception once the iterator i becomes zero. For any $x \leq 0$, line 4 cannot be reached.

Let us quickly discuss the concepts of reachability, feasibility, and inconsistency using this example. Every line in this procedure is (forward) *reachable*, meaning that, for each statement of the procedure, we can find a sequence of statements that reaches it from the entry of the procedure and is *feasible*. That is, assuming that x can take arbitrary values, there exist a concrete value for x that triggers an execution ending in the statement. If, on the other hand, we would assume that $x \leq 0$, then the body of the `for`-loop would not have any feasible executions and would thus be *unreachable*. However, every feasible execution of line 4 must terminate exceptionally later due to unavoidable division by 0. Hence, we declare line 4 as *inconsistent* because any feasible execution containing this line must terminate exceptionally.

Now, we want to use formal techniques to prove the inconsistency in line 4. In the literature one finds several algorithms that (among other things) prove the existence of inconsistencies (e.g, [10, 18, 22, 34]). However, none of the existing algorithms would be able to detect the inconsistency because of their inability to handle looping control-flow. Unwinding the loop is not an option either in this example because the bounds are unknown at compile time. Even though each unwinding would reveal the error, this is not sufficient to prove the inconsistency (because, for the statement to be inconsistent, the error has to occur on every iteration). Hence, we need an approach that is able to infer an inductive invariant that allows us to prove that every feasible execution containing line 4 must terminate exceptionally.

In the following sections, we first describe how we encode the running example from Figure 1 as a system of constrained Horn clauses and then we present an algorithm to prove that line 4 is in fact inconsistent.

4 Horn Clause Encoding

In this section, we describe how we encode the example from Figure 1, as a system of constrained Horn clauses. First, we describe the syntax and semantics of Horn clauses.

Given a set \mathcal{F} of function symbols (e.g., $+$, $=$, etc), a set \mathcal{P} of predicate symbols, and a set \mathcal{V} of variables, a *Constrained Horn Clause (CHC)* is a formula:

$$\forall \mathcal{V}. (p[X] \leftarrow \phi \wedge p_1[X_1] \wedge \dots \wedge p_k[X_k])$$

for $k \geq 0$, where ϕ is a constraint over \mathcal{F} and \mathcal{V} . Each $p_i[X_i]$ is the application of a predicate $p_i \in \mathcal{P}$ for first-order terms constructed from \mathcal{F} and \mathcal{V} . We refer to the left-hand side of the implication as *head* and to the right-hand side as the *body* of the Horn clause. A clause is called a *query* if its head is \mathcal{P} -free, and otherwise, it is called a *rule*. We say a clause is *linear* if its body contains at most one predicate symbol \mathcal{P} , otherwise, it is called *non-linear*. For scalability reasons, our algorithm, described in Section 6, for detecting inconsistencies is intra-procedural. As a result, all of our CHCs will be linear⁴. Finally, we will follow the CLP convention of writing Horn clauses as $h[X] \leftarrow \phi, p_1[X_1], \dots, p_k[X_k]$.

A system of CHCs is satisfiable if there exists an interpretation \mathcal{J} of the predicate symbols \mathcal{P} such that each constraint ϕ is true under \mathcal{J} . If satisfiable, we assume that the CHC solver (e.g., GPDR [17], Eldarica [31], or Spacer [25]) also returns a *model*: assignments of values to variables.

We assume that the reader is familiar with the basic concepts of how to encode programs using Horn clauses (see e.g., [5, 16, 30] for details). For simplicity, we use in this presentation an encoding based on small-step operational semantics [28] and describe informally how to translate programs to CHCs. Note that our approach is not limited to a particular encoding and we can also use other encodings, for instance, large-step [4, 15] (*a.k.a.* Large Block Encoding, or LBE).

Thinking in terms of programs and basic blocks (sequence of statements without branching), the predicates $p_1 \dots p_k$ encode the control-location where a program could have been before reaching the current basic block. The constraint ϕ encodes the transition relation of the statements in this basic block, and the predicate p in the head of the Horn clause indicates where control flows if the transition relation ϕ allows for a feasible execution of the basic block.

Figure 2 shows a CHC encoding of our running example from Figure 1. Each predicate p_0, \dots, p_4 corresponds to a control location in our program. The predicate p_0 in the first line is the procedure entry encoding that the entry of

⁴ In presence of function calls, a CHC will have at least two predicate symbols in its body: one that represents the callee and the other modelling the successor. If callsites are ignored then CHCs will have only one predicate symbol modelling the successor.

$$\begin{array}{ll}
p_0(x, ret, i) \leftarrow true. & \\
p_1(x, ret', i') \leftarrow ret' = 0 \wedge i' = -x \wedge p_0(x, i, ret). & \textit{procedure entry} \\
p_2(x, ret, i) \leftarrow i < x \wedge p_1(x, ret, i). & \textit{loop conditional} \\
p_3(x, ret, i) \leftarrow i \geq x \wedge p_1(x, ret, i). & \\
p_1(x, ret', i') \leftarrow i \neq 0 \wedge ret' = 10/i \wedge i' = i + 1 \wedge p_2(x, ret, i). & \textit{loop body} \\
p_4(x, ret, i) \leftarrow p_3(x, ret, i). & \textit{procedure return}
\end{array}$$

Fig. 2. Constrained Horn clause representation of the example from Figure 1. The $i \neq 0$ colored in red is the implicit runtime assertion introduced by the division in line 4 of our running example.

the function is always reachable. The second line states that if we are at the procedure entry p_0 and $ret' = 0$ and $i' = -x$ can be established, we are allowed to proceed to the loop head p_1 . The next two lines state that, if we are at the loop head and the loop condition $i < x$ holds, we proceed into the loop body p_2 , or otherwise, we go to the loop exit p_3 . The next line represents the loop body. Note the $i \neq 0$ colored in red. This is the implicit run-time assertion that needs to hold when executing the division in the loop body of our running example in Figure 1. We assume that these assertions have been introduced during the translation. If the loop body is executed successfully, control moves back to the loop head p_1 . The last line is the loop exit. For brevity, we do not model the return statement and just assume that ret is visible to the outside.

Once obtained the system of constrained Horn clauses representing a program, we can add a query. A typical query for our example from Figure 2 would be:

$$p_4(x, ret, i) \tag{1}$$

which checks if the control location associated p_4 at the end of the procedure is reachable. If this query is satisfiable, the CHC solver produces a *model*. For our encoding, a model also encodes a program state, and the existence of this state witnesses that there is a *feasible path* reaching the associated control-location. For short: if a model for p_4 exists then `foo` has a feasible complete path.

If no such model exists, the CHC solver provides a proof that the program has no feasible execution (that reaches the end of `foo`). For our example, we can find a model that sets x to a value less or equal to zero. For this input, the execution of the procedure skips the loop and terminates normally.

The challenge now is, how do we check if there is another model that executes a different path? Let us assume that our previous query provided us with a path through p_0 , p_1 , p_3 , and p_4 . Now we want to check if there is also a path through p_2 . It would be tempting to build the following query

$$p_4(x, ret, i) \wedge p_2(x, ret, i). \tag{2}$$

$$\begin{aligned}
p_0(x, ret, i, \top, \perp, \perp, \perp, \perp) &\leftarrow true. \\
p_1(x, ret', i', r_0, \top, r_2, r_3, r_4) &\leftarrow ret' = 0 \wedge i' = -x \wedge p_0(x, i, ret, r_0, r_1, r_2, r_3, r_4). \\
p_2(x, ret, i, r_0, r_1, \top, r_3, r_4) &\leftarrow i < x \wedge p_1(x, ret, i, r_0, r_1, r_2, r_3, r_4). \\
p_3(x, ret, i, r_0, r_1, r_2, \top, r_4) &\leftarrow i \geq x \wedge p_1(x, ret, i, r_0, r_1, r_2, r_3, r_4). \\
p_1(x, ret', i', r_0, \top, r_2, r_3, r_4) &\leftarrow i \neq 0 \wedge ret' = 10/i \wedge i' = i + 1 \\
&\quad \wedge p_2(x, ret, i, r_0, r_1, r_2, r_3, r_4). \\
p_4(x, ret, i, r_0, r_1, r_2, r_3, \top) &\leftarrow p_3(x, ret, i, r_0, r_1, r_2, r_3, r_4).
\end{aligned}$$

Fig. 3. Constrained Horn clause representation of the example from Figure 1 with crumb variables. Each crumb variable r_i corresponds to the a program location associated with the predicate p_i . In the head of each clause, we can see that the crumb variable for that predicate is set to true (\top) which corresponds to updating the variable to \top when the location is reached. For p_0 all the crumb variables are set to false (\perp) except the one that corresponds to the entry of the procedure (r_0).

Unfortunately, this query does not ask for a path that passes through p_2 and p_4 . Instead, it asks whether p_2 and p_4 are reachable (not necessarily on the same execution) with the same values for x , ret , and i . Even if we rename the variables of p_2 and p_4 to disjoint sets of variables the same problem remains. The reason is that our Horn clause encoding allows only for checking forward reachability. However, our aim is to check if a particular location can be passed during an execution that reaches the end of a procedure. Hence, we have to extend our encoding to capture which locations have been visited on a path.

5 Crumb Variables

To extend our encoding in a way that allows us to extract a feasible path directly from the model returned by the Horn clause solver while blocking paths that we have already covered, we add auxiliary Boolean variables to our encoding. Our approach is inspired by a similar approach using Integer variables that has been presented in [3]. Thinking in terms of programs and executions, the idea is to add one Boolean variable r_i per control location (i.e., per predicate p_i in the Horn clause system). All these variables are initially set to false. If a control location is reached, the corresponding Boolean variable is set to true. Now, we can obtain a path from a model by looking at the values of these Boolean variables at the last program location. Throughout the rest of this paper, we refer to these variables as *crumb variables* because we disperse them in the encoding so that the Horn clause solver can find a path while constructing a model.

Figure 3 shows how we encode the procedure `foo` from our running example into another system of Horn clauses. For each predicate $p_0 \dots p_4$ we introduced a crumb variable $r_0, \dots r_4$. In the head of each Horn clause, we enforce that the crumb variable r_i is set to true when transferring control to p_i (alternatively,

we could update r_i in the body of the Horn clause like a proper assignment, but this representation is shorter).

Note that, in practice, we do not need one crumb variable per location. It is sufficient to add crumb variables for the minimal subset of locations that need to be covered to ensure that all locations can be covered (in our example this would be p_2 and p_3). The definition of this minimal set is given in [3].

For example, using crumb variables, the incorrect query from (2) in previous section, is encoded correctly as follows:

$$p_4(x, ret, i, \top, r_1, \top, r_3, \top) \quad (3)$$

That is we are asking if it is possible to reach the end of the procedure (by enforcing that p_4 has to hold) in a state where r_0 , r_2 , and r_4 have been set to true. Thus, we only allow models representing executions of complete paths that visit p_2 at least once.

Lemma 1. *Given a system of CHCs for a program P with a set of predicates $p_0 \dots p_n$ and a set of crumb variables $r_0 \dots r_n$, and $p_i(\vec{v}, r_0, \dots, r_n)$, where \vec{v} is a vector of program variables. A query $p_i(\vec{v}, r_0, \dots, r_n)$ has a model m if and only if there exists a feasible path in P that reaches the control location associated with p_i . Further $m(r_i)$ is true for all r_i associated with control-locations on this path.*

From Lemma 1 follows that querying the predicate that represents the exit of a program allows us to check for the existence of a feasible path. Further, by adding additional conjuncts to the query that certain crumb variables have to be true, we can check if a feasible path through certain locations exists.

Also note that, unlike in encodings that eliminate loops, a model m encodes paths with loops (that is a path rather than a walk in terms of graph theory). Hence, if $m(r_i)$ is true we know that there exists a feasible path through p_i , but we do not know how often p_i is visited when executing this path.

A proof of Lemma 1 in the context of programs and control-flow graphs is given in [3]. Assuming that our Horn clause representation captures the semantics of this control-flow graph as described in [16], this proof also holds for our Horn clause representation of programs.

Fault Localization. In query (3), we checked for the existence of a feasible path that passes through the loop body (represented by p_2). Since no such path exists, the query is unsatisfiable and a Horn clause solver will give us an *invariant* for each predicate. Such invariants can be used to apply static fault localization techniques such as [32] or [23].

For our running example, a Horn clause solver would provide us the following invariants:

$$\begin{aligned} p_0(x, ret, i) &\leftarrow \text{true}, & p_1(x, ret, i) &\leftarrow i < 0, & p_2(x, ret, i) &\leftarrow i < 0, \\ p_3(x, ret, i) &\leftarrow \text{false}, & p_4(x, ret, i) &\leftarrow \text{false} \end{aligned}$$

The first statement proves that the program location p_0 is valid. Once we enter the loop at p_1 , the invariant $i < 0$ holds and takes us into the loop body at

p_2 where the invariant still holds. Further, for p_3 and p_4 the invariant is false meaning that the execution of the program must end once the invariant $i < 0$ does not hold anymore. This exactly describes the error. If we enter the loop, we must have a negative i . We can iterate the loop until i becomes zero and then we crash.

One can think of different ways of presenting this information to a programmer. For example, using automaton based representation as described in [32], or a compressed trace with annotations. In summary, using a Horn clause solver to detect inconsistent code provides us a fault localization for free. This is a significant improvement over previous approaches where the fault localization had to be computed manually.

6 Inconsistency Detection through Horn clause coverage

Using our Horn clause encoding with the crumb variables from the previous section, we are able to ask for any location in a program whether it is inconsistent or not. However, in practice, we want to know if a procedure contains *any* inconsistencies. Checking each location individually would not be very efficient (see [3]). Instead, we propose an algorithm that repeatedly asks the solver for a feasible path and then blocks this path to ensure that, in the next query, the solver will exercise a different path that visits at least one control location that has not been visited previously. In a nutshell, we want to compute a *path coverage* for the generated system of Horn clauses.

Algorithm 1: Horn clause coverage algorithm.

Input: \mathcal{HC} : constrained Horn clause encoding of a program with crumb variables.
Output: $feasible$: Crumb that can occur on a feasible path.

```

1 begin
2    $crumbs \leftarrow getCrumbs(\mathcal{HC})$ ;
3    $p_{sink} \leftarrow getSink(\mathcal{HC})$ ;
4    $feasible \leftarrow \emptyset$ ;
5    $blocking \leftarrow true$ ;
6   while  $query(p_{sink} \wedge blocking)$  do
7      $model \leftarrow getModel(p_{sink})$ ;
8      $blocking \leftarrow blocking \wedge getBlockingClause(model)$ ;
9      $feasible \leftarrow feasible \cup \{r \mid r \in crumbs \wedge model(r)\}$ ;
10  end while
11  return  $feasible$ ;
12 end
```

Algorithm 1 shows our covering algorithm for Horn clause systems. The algorithm takes a program encoded as system of constrained Horn clauses \mathcal{HC} augmented with crumb variables as input and returns the set of crumb variables

feasible that occurred on feasible paths. To that end, the algorithm first uses the helper function `getCrumbs` to collect all crumb variables from the input Horn clauses. Then, the algorithm calls `getSink` to get the predicate associated with the last control location p_{sink} in the program encoded by \mathcal{HC} . This location is needed later on to query if there exists a feasible complete path (that is a path reaching p_{sink}). Further, the algorithm uses the helper variable *blocking*, which is initially `true` to exclude all models representing program paths that have already been visited.

The main loop of Algorithm 1 repeatedly checks if p_{sink} in conjunction with the blocking clause *blocking* has a model. It uses the helper function `query` which either returns `true` or `false` (or runs forever). If `query` returns `false`, we have a proof from the solver that no feasible path exists and we return the set *feasible*. If `query` returns `true`, we use the helper function `getModel` to obtain a *model* from the solver that assigns each variable in p_{sink} to a value. In particular, *model* contains an assignment for each crumb variable r such that set of crumb variables assigned to `true` represent a feasible path.

Using the *model* obtained from the solver, we now extend our blocking clause *blocking* to exclude the feasible path represented by *model*. To that end, we use the function `getBlockingClause` which constructs a conjunction of all crumb variables where the variables that are assigned to `false` in *model* occur in negated form, and the ones assigned to `true` in positive form:

$$\text{getBlockingClause}(\text{model}) = \neg \left(\bigwedge_{r \in \text{getCrumbs}(\mathcal{HC})} \begin{cases} r & \text{if } \text{model}(r) = \text{true} \\ \neg r & \text{if } \text{model}(r) = \text{false} \end{cases} \right)$$

One important difference between computing such a blocking clause for Horn clause systems with loops compared to Horn clause systems without loops is that our blocking clause also *must* include conjunctions for the crumb variables that are assigned to `false` in the *model*. If we would only include those crumb variables set to `true` by the *model*, we would also block all paths that visit a superset of the locations visited on this path. Think, for example, of a program containing a single loop with one conditional choice in its body. Let us further assume the then-branch must be visited in the first iteration of the loop, and the else-branch must be visited in all other iterations. If our Horn clause solver gives us a *model* in which the then-branch is visited but not the else-branch, and we would add a blocking clauses containing only the crumb variables that are `true` in *model*, we would also block all feasible paths through the else-branch. This is because any path through the else-branch must go through the then-branch in the first iteration of the loop. Hence it must set all crumb variables to `true` that are `true` in *model* and, in addition to that, the crumb variable for the else-branch.

After updating our blocking clause, Algorithm 1 adds all crumb variables assigned to `true` by *model* to the set *feasible*. This loop is iterated until no new model can be found. Then, the algorithm returns the set *feasible* of all crumb variables which correspond to feasible control locations in the program. All other control locations in the program are inconsistent.

Note that Algorithm 1 is guaranteed to terminate if `query` terminates: each iteration of the loop extends the blocking clause in a way that the next iteration has to visit at least one new control location. Since the number of control locations is finite (even if the number of paths is not), the loop must reach a point where all control locations that occur on feasible paths have been visited. Then it is up to `query` to prove that there is no more feasible path (this, however, is undecidable).

Remarks on soundness and completeness. Since finding inconsistent code is not safety checking, let us briefly clarify what *soundness* means in the context of inconsistent code detection: An inconsistent code detection algorithm is *sound*, if every inconsistency reported in \mathcal{HC} is in fact an inconsistency in the program it encodes (i.e., a proof that no feasible paths through a control location exists in the Horn clause encoding also is a proof that no such path exists in the original program). So, to be sound, our Horn clause encoding must over-approximate the feasible executions of the original program (which is usually easier than over-approximating the failing executions which is needed to prove safety). Our implementation for \mathcal{C} is not sound as we will discuss later. Further, our algorithm is only sound if our Horn clause solver is sound. That is, Algorithm 1 does not introduce unsoundness, but our implementation used in the evaluation is unsound.

Completeness in the context of inconsistent code detection means that an algorithm detects all inconsistencies. Our algorithm is complete if the employed Horn clause solver is complete - which is not the case since the problem is undecidable. Further, we lose completeness during the translation into Horn because we cannot guarantee that the translation preserves all feasible executions of the original program which we will discuss later.

7 Experimental Evaluation

We have implemented our technique on top of the SeaHorn framework [16]. Our tool uses SeaHorn capabilities for translating LLVM-based programs into a set of recursive Horn clauses. This saved us a huge amount of work since SeaHorn deals with the translation from \mathcal{C} to LLVM bytecode, performs LLVM optimizations and runs some useful transformations (e.g. mixed-semantics transformation) as well as a pointer analysis. This also allowed us to support programs with pointers and arrays without extra effort. We implemented a variant of the small-step encoding in order to accommodate the crumb variables. We leave for future work the extension to more efficient encoding such as Large-Block Encoding (LBE).

We have implemented the algorithm described in Section 6 in Python. The code that is publicly available at [33]. The algorithm is applied on each function separately rather than the whole program. Although our prototype analyzes several functions concurrently one important limitation is that it generates the Horn clauses for the whole program. This has limited us significantly with real applications. For future work, we will instead generate Horn clauses for each function.

Benchmark	Inconsistency detected				# iterations
	with loops	time (sec)	with abstraction	time (sec)	
example 1	✓	0.09	✗	0.045	1
example 2	✓	0.065	✓	0.06	2
example 3	✓	5.78	✓	5.78	33
example 4	✗	TIMEOUT	✗	0.22	?
example 5	✓	0.33	✗	0.03	3
example 6	✓	0.04	✓	0.085	1
example 7	✓	0.12	✗	0.09	3
example 8	✓	2.956	✓	0.085	5
example 9	✓	0.01	✓	0.02	1
example 10	✓	3.66	✗	1.15	4
example 11	✓	7.91	✗	0.08	2
example 12	✓	44.64	✗	0.14	2
example 13	✗	0.33	✗	0.03	2

Table 1. Results of applying our inconsistent code detection on a set of benchmarks. The benchmarks are handcrafted in the spirit of SV-COMP benchmarks challenge the algorithm with different categories of loops. We check for each benchmark if the inconsistency can be detected by our approach (with loops) and by an approach where loops are abstracted (with abstraction). We further record the number of iterations of our algorithm (i.e., number of feasible paths) and the computation time.

For this purpose we will still need a scalable and precise pointer analysis that can analyze the whole program in presence of pointers and arrays. Fortunately, SeaHorn relies on a heap analysis called *Data Structure Analysis (DSA)* which has been very effective for real applications [26].

Experimental setup. To evaluate our approach we handcraft a set of benchmark problems. The idea is to create a set of small but hard benchmarks in the spirit of what is being used in the software verification competition that will allow us to compare different Horn clause solving strategies in the future. All our benchmarks are available online⁵ and contain different kinds of inconsistencies which we will describe below in more detail.

For the experiments we used SeaHorn running Spacer [24] in the backend to solve the generated CHCs. All experiments are run on a Macbook Pro with 2.4Ghz and 8 GB or memory.

Discussion. Table 1 shows the results of running our tool on the set of handcrafted benchmarks. The first column shows the name of our benchmark, the second column shows a ✓ if our tool detects the inconsistency in the benchmark, and a ✗ if it fails to do so.

All examples contain inconsistent code. Example 1 is our running example. The other examples represent different challenging problems for Horn clause solvers. The examples 2 and 3 do not contain loops, and hence can be solved by approaches that abstract loops. The examples 6, 8, 9, and 10 are taken from [20],

⁵ <https://github.com/seahorn/seahorn/tree/inconsistency/play/inconsistency>

a Wikipedia list of common loop errors, and [27]. They represent cases where inconsistencies in code can be found even with abstraction. Example 6 is an inconsistency that must happen in the first iteration of the loop. For 8 and 9, the inconsistency is local to the loop body and thus can be detected without considering the loop. Example 10 is a typical off-by-one error that can still be detected using, e.g., the loop abstraction in [19] or [34].

Our approach fails on the examples 4, and 13. Example 4 is a faulty implementation of binary search that sets the mid point in a way that leads to an endless loop. Our Horn clause solver is not able to infer a suitable invariant to prove this and infinitely unwinds the loop. Example 13 contains two loops. The first loop allocates a two dimensional matrix but erroneously iterates over the wrong variable which results in unallocated fields in the matrix. The second loop assigns all fields leading to an inevitable segmentation fault. SeaHorn currently does not check if memory is allocated, hence we cannot find this inconsistency.

For all other examples, our approach is able to find inductive invariants that are sufficient to prove the existence of the inconsistency. That is, our approach only times out on a single example and is able to identify six instances of inconsistent code that went undetected before. Hence, we believe that our approach of using CHCs to detect inconsistencies is viable in practice (in particular because we can always fall back to abstraction-based approaches if we timeout).

Comparing the computation time of our approach with inconsistent code detection that abstracts loops shows that our approach is not significantly slower on examples that can be solved by both, and sometimes even faster. On examples that can only be solved by our approach, the overhead is sometimes significant (e.g., 8, 11, and 12) but we believe that there is still room for improvement.

Threats to Validity. We report on several threats to validity. Our internal validity is affected by choosing SeaHorn as a frontend to translate C into CHCs, and by using Spacer as a backend to solve those CHCs. Using different frontends or backends may give completely different results. However, we do not claim that our setup is more effective than others. In fact, we encourage readers to try other setups that outperform our approach. The other obvious internal threat to validity is selection bias. We cannot guarantee that our handcrafted benchmarks resemble real inconsistencies. However, we believe that, as a first step, these experiments are sufficient to motivate that inconsistent code detection in the presence loops is an interesting problem, and that our benchmark programs can serve as a baseline for researchers.

A threat to external validity (i.e., generalizability of the results) for any inconsistent code detection algorithm is that we cannot quantify the number of false negatives (because it is undecidable). Hence, we cannot quantify how much better our approach performs at finding inconsistencies than previous approaches. However, by design, we can say that it finds at least the same inconsistencies as previous approaches and maybe more. A suitable way to evaluate this would be by injecting inconsistencies into real code. However, there is no related empirical work on how *realistic* inconsistencies can be injected into code. Another way to reduce the threat to external validity is to run our tool on industrial benchmarks.

8 Conclusion

In this paper, we have presented a novel approach to detect inconsistent code in the presence of looping control-flow. Our approach encodes the problem of detecting inconsistent code into the problem of solving a system of constrained Horn clauses. Unlike existing approaches, we do not need to abstract looping control-flow in a preprocessing step. Hence, our ability to detect inconsistencies is only limited by the employed Horn clause solver. This allows us to detect a larger class of inconsistencies than any existing techniques. Moreover, this represents an interesting novel application of Horn clause solving.

We propose a set of benchmark programs containing inconsistent code that we made available online. Our experiments show that our implementation is able to detect several inconsistencies in these programs that could not be detected by other tools at a reasonable overhead. In particular, we can always fall back to abstraction-based approaches if our technique does not converge. In fact, to achieve better scalability we envision a technique that integrates our methods with abstraction-based approaches. In the future we plan to evaluate our approach in industrial scale code base.

References

1. A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In *CAV*, pages 672–678, 2012.
2. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.
3. C. Bertolini, M. Schäf, and P. Schweitzer. Infeasible code detection. In *VSTTE*, pages 310–325, 2012.
4. D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, pages 25–32, 2009.
5. N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, pages 24–51, 2015.
6. S. Blackshear and S. K. Lahiri. Almost-correct specifications: a modular semantic framework for assigning confidence to warnings. In *PLDI*, pages 209–218, 2013.
7. A. R. Bradley. IC3 and beyond: Incremental, inductive verification. In *CAV*, page 4, 2012.
8. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verimap: A tool for verifying programs through transformations. In *TACAS*, pages 568–574, 2014.
9. G. Delzanno and A. Podelski. Model checking in CLP. In *TACAS*, pages 223–239, 1999.
10. I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *PLDI*, pages 435–445, 2007.
11. D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP*, 2001.
12. E. Ermis, M. Schäf, and T. Wies. Error invariants. In *FM*, pages 187–201, 2012.

13. C. Flanagan. Automatic software model checking via constraint logic. *Sci. Comput. Program.*, 50(1-3):253–270, 2004.
14. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.
15. A. Gurfinkel, S. Chaki, and S. Sapra. Efficient Predicate Abstraction of Program Summaries. In *NFM*, pages 131–145, 2011.
16. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *CAV*, pages 343–361, 2015.
17. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, pages 157–171, 2012.
18. J. Hoenicke, K. R. M. Leino, A. Podelski, M. Schäf, and T. Wies. It’s doomed; we can prove it. In *FM*, pages 338–353, 2009.
19. J. Hoenicke, K. R. M. Leino, A. Podelski, M. Schäf, and T. Wies. Doomed program points. *FMSD*, pages 171–199, 2010.
20. D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *PASTE*, pages 9–14, 2007.
21. J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. In *CAV*, pages 758–766, 2012.
22. M. Janota, R. Grigore, and M. Moskal. Reachability analysis for annotated code. In *SAVCBS*, pages 23–30, 2007.
23. M. Jose and R. Majumdar. Bug-Assist: Assisting Fault Localization in ANSI-C Programs. In *CAV*, pages 504–509, 2011.
24. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. In *CAV*, pages 17–34, 2014.
25. A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke. Automatic abstraction in smt-based unbounded software model checking. In *CAV*, pages 846–862, 2013.
26. C. Lattner and V. S. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *PLDI*, pages 129–142, 2005.
27. T. McCarthy, P. Rümmer, and M. Schäf. Bixie: Finding and understanding inconsistent code. In *ICSE*, pages 645–648, 2015.
28. J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. In *SAS*, pages 246–261, 1998.
29. C. Popeea and A. Rybalchenko. Threader: A verifier for multi-threaded programs - (competition contribution). In *TACAS*, pages 633–636, 2013.
30. P. Rümmer, H. Hojjat, and V. Kuncak. Classifying and solving horn clauses for verification. In *VSTTE*, pages 1–21, 2013.
31. P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for horn-clause verification. In *CAV*, pages 347–363, 2013.
32. M. Schäf, D. Schwartz-Narbonne, and T. Wies. Explaining inconsistent code. In *ESEC/FSE*, pages 521–531, 2013.
33. SeaHorn Inconsistency Checker.
Available at: <https://github.com/seahorn/seahorn/tree/inconsistency>.
34. A. Tomb and C. Flanagan. Detecting inconsistencies via universal reachability analysis. In *ISSTA*, pages 287–297, 2012.
35. X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. A differential approach to undefined behavior detection. *ACM Trans. Comput. Syst.*, 33(1):1:1–1:29, 2015.