Loop Invariant Generation for Deductive Verification of Embedded Systems

IOAN - ALEXANDRU ZAMBORI, University of Twente, The Netherlands

Embedded systems are highly specialized computer systems integrated within larger, typically electronic or mechanical systems. Unlike general computing on personal computers or enterprise servers, which aim to facilitate a vast set of operations applicable across many domains, embedded systems serve highly specific tasks and, as such, are generally purpose-built. Since embedded systems are present in safety-critical systems, suitable verification is paramount to ensure correctness. One strategy is the use of static analysis for deductive verification. VerCors is a toolset that can be used for such verification of concurrent systems, and previous work has generated tooling that can transform SystemC code into VerCors' Prototypal Verification Language, alongside most of its respective annotations. This research aims to expand upon this work in the sector of loop invariant generation. This is achieved by simulating loop behavior and keeping track of over-approximations over the set of possible values of program variables. In doing so, we can provide numerical bounds for said program variables, which are used to provide invariants with respect to loop bounds.

Additional Key Words and Phrases: Embedded System, Loop Invariant Generation, Loop Bound Analysis, Deductive Verification, VerCors, SystemC

1 INTRODUCTION

With the advance of technology, embedded systems are becoming more and more common and fundamental to the smooth operation of a large variety of systems. Applications in which embedded systems are present are vast and ever-increasing. These range from relatively simple wearable devices, such as sensors within smartwatches, to safety-critical applications, such as autonomous driving. Thus, there is a need for systematic methods to validate the correctness of embedded systems, as highlighted by [15]. Verification is important in this context for two main reasons: the cost of testing and development of technologies can be decreased by finding bugs early, but more importantly, correct behavior of embedded systems is necessary to ensure the safety of people in safety-critical systems.

Static analysis is a family of software verification techniques that analyzes input programs without actually running them directly. It can be used to reason about program behavior through the use of logic and constraint analysis, which can yield sound proofs about the correctness of the program.

Embedded systems generally use specific hardware with associated software, which tends to be parallelized in nature in order to optimize its performance, and as such, support for concurrency is necessary in order to perform static verification on it. SystemC is a design language implemented as a C++ library, used in simulating hardware designs, which can be used to prototype embedded system architectures. In this study we use SystemC to illustrate our work.

TScIT 43, July 4, 2025, Enschede, The Netherlands

One tool that offers such support for parallelized systems is VerCors [1], which is built specifically with the intent of studying concurrent systems in languages such as C and Java. While VerCors does not natively offer support for languages used in embedded system design, previous work [14] has been done to create tools that can transform SystemC code [9], which is widely used in industry, to VerCors' internal Prototypal Verification Language (PVL). Alongside code transformation, limited annotation inference is offered, thus easing the process of verification of the designed system. These annotations take the form of functional contracts, specifying the pre- and post-conditions of methods, and invariants, highlighting parts of the program state that do not change throughout certain sections of the code.

Static analysis needs loop invariants to be able to reason about loops. This is because loops cannot be inspected directly, since they are blocks of code that may be executed arbitrarily many times. Thus, loop invariants are used to assume certain properties of the loop throughout its execution. Inferring loop invariants is, however, quite difficult. This is because they largely depend on the actual implementation of the loop, unlike functional contracts, which are abstract entities defining how a piece of code must behave [6]. There are a number of previous works [2, 4–6, 12] that define different methods of loop invariant generation, employing methods derived from the domain knowledge of the particular applications under analysis, but the problem of loop invariant generation is undecidable for the general case.

We approach this problem with a technique that relies on **symbolic execution** - the execution of a program where, instead of concrete numerical inputs, variables are initialized with symbolic values, after which the program is run with said symbolic values. For this method, we have chosen to represent our symbolic values with intervals that represent an overapproximation over the set of all possible values for a given variable during execution. In doing so, we generate concrete numerical bounds for program variables, which are then transformed into loop invariants. This shows promise in the context of embedded systems verification using VerCors because of the intrinsic use of concrete integer values throughout SystemC designs.

This study aims to explore the applicability of loop bound analysis (the study of upper and lower execution bounds of loops) techniques, namely abstract interpretation through interval abstract domains, to the domain of loop invariant generation for embedded systems design. The idea to use such methods in this domain came as inspiration from the work of Čadek et al. [2], who have successfully generated a variety of loop invariants based on the observation that loop bounds and invariants are tightly related, allowing extrapolation of one from the other. In their paper, the authors primarily tackled the use of ranking functions, which provide an upper bound to the iteration count of a loop, and metering functions, which provide similar lower bounds.

Apart from the work of Čadek et al., this study tries to expand on the work of Tasche et al. [14], by adding automated generation for loop invariants representing concrete numerical bounds. The resulting PVL and annotations can then be used as input to VerCors, which uses separation logic to reason about different properties of concurrent systems, thus being a good fit for embedded system verification.

The resulting method of loop invariant generation, alongside the implementation and evaluation of this technique, represents the main contribution of this paper.

The structure of this paper is as follows: first, the aims of the study are formulated in the form of research questions, followed by related work and how this study differentiates itself from it. Afterwards, the necessary background is provided before delving into the proposed method, alongside its evaluation. Finally, we conclude by answering the research questions and offering suggestions for future work.

2 PROBLEM STATEMENT

Research into loop invariant generation is extensive; however, no general, all-encompassing solution exists. Thus, studies such as this one aim to slowly advance the area towards solutions that are applicable to real-life scenarios. This study strives to take existing knowledge of loop invariant generation, specifically through the use of loop bound analysis, which has been previously demonstrated to be valid for sequential programs [2], and apply it to the domain of embedded system design verification. With this in mind, the research question is posed:

RQ: How can loop bound analysis techniques be used in loop invariant generation for embedded system design verification?

When investigating this query, the following sub-research questions will be used:

1. **SRQ: What types of loop invariants can be generated based on loop bound analysis?**
2. **SRQ: What kinds of loops can have invariants generated?**
3. **SRQ: How effective are solutions found for generating useful loop invariants?**

3 RELATED WORK

In this section we look at work related to the present study and how this study fits within the domain of loop invariant generation.

A variety of loop invariant generation techniques have been proposed over the years. As no general solution for this problem exists, existing tools use a diverse set of approaches based on the domain of the application under verification. Present solutions tend to fall into either a static generation category, dynamic generation, or, more recently, hybrid approaches [8].

Using loop bound analysis methods for inferring loop invariants has been shown as a promising approach to the problem in a paper by Čadek et al. [2]. In their work, the authors claim that loop bounds and loop invariants can have intrinsic relations, allowing one to infer one based on the other. They argue that the field of loop bound analysis presents many different techniques and approaches that were not used in the domain of invariant generation prior. The authors have implemented their approach in a tool [3], which they used to supplement invariants generated by state-of-the-art tooling. By doing so, they demonstrated that invariants found through loop bound analysis differ from loop invariants generated through different means and improve the performance of those tools noticeably on the SV-COMP [11] 2018 benchmark.

This work aims to expand upon the ideas presented by Čadek et al. by applying loop bound analysis techniques in the context of embedded systems, expanding the invariant generation capabilities of previous work, such as [14]. We take the primary idea of using loop bounds to generate loop invariants, and we propose a method for generating loop invariants representing concrete numerical bounds for program variables by means of symbolic execution, specifically using interval abstract domains to simulate how said bounds mutate by means of symbolic execution of the loop. The proposed method differentiates itself from previous work by applying different techniques, namely symbolic execution, as opposed to making use of ranking functions. Additionally, previous work places focus on symbolic relations, whereas this study tries to compute numerical bounds.

4 BACKGROUND

4.1 VerCors

VerCors [1] is a tool set for software verification, specifically geared towards concurrent systems. It supports a wide array of programming languages as input, such as Java and C, but more importantly, its own Prototypal Verification Language (PVL), which is an object-oriented programming language that aims to support a large array of language features, with a primary focus on integer and boolean data types. We are interested in PVL specifically because in this work we use VeSUV, which transforms SystemC to PVL. In order to check properties of the program under verification, VerCors uses contract-based reasoning by means of method contracts, so pre-conditions (e.g. $requires\ n > 0$), post-conditions (e.g. $\backslash result > 0$), and loop invariants (e.g. $loop\_invariant\ i <= n$), which are annotations of the input code that specify its behavior. VerCors will use these annotations to reason about the program behavior, replacing method calls by their respective contracts and verifying subsections in isolation.

4.1.1 VeSUV. VeSUV [14] is a tool in the VerCors toolset that is used to transform a SystemC design into an equivalent program written in VerCors' PVL, thus enabling verification. First, the SystemC code is parsed with an external tool into an Abstract Syntax Tree (AST), which is a structured representation of the code. This AST is then parsed yet again into SystemC Internal Representation (SystemC IR). The SystemC IR is further translated by VeSUV into the Common Object Language (COL), an internal representation used by VerCors, by encoding timing events and processes. These, alongside scheduling information, are embedded into a Reachable Abstract States Invariant (RASI) [13], that encapsulates the possible state space in such a way as to avoid state space explosion and is turned into a global invariant that VerCors can use to reason about the current state of the program during verification. During the translation into COL, annotations are also generated: the RASI, but also loop invariants based on path conditions. It is at this point that we wish to generate further loop invariants.

4.2 Symbolic Execution

In this section we provide an overview of and necessary definitions for symbolic execution, which are to be used in the proposed method for generating loop invariants.

Symbolic execution [7] is a technique for reasoning about program behavior by utilizing symbols instead of inputs for program variables and executing the program based on those symbols rather than concrete values. The fundamental idea is to represent a variable by a symbol and apply mutations to said symbol according to the provided piece of code. In our case, instead of a single symbol, we map program variables to symbolic intervals (e.g. $i : [5 - j, 5 + j]$) and symbolically execute the code under inspection, which we then use to infer numerical bounds for the variables in the loop. The idea to use intervals is inspired by [15].

A **symbol** is a singular program variable that may or may not have a known value at a certain point in the program execution (e.g., "N", "i", etc.). To be used in abstract interpretation, symbols are mapped to abstract interval domains, which we will call **symbolic intervals**. These represent the set of all possible values that a variable might have at a certain point in program execution. The bounds of these intervals are represented by linear expressions (linear combinations of symbols, alongside a possible constant term) and considered to be inclusive, unless they are unbounded on one side (i.e., that bound is infinite). The mapping itself is referred to as a **symbolic state**, which represents the relation between all symbols at a given point during program execution to their respective symbolic intervals.

A **linear expression** is a linear combination of symbols with all coefficients being constant terms, alongside a trailing constant. (e.g., "$i + 1$", "$2 \cdot i + 3 \cdot j - 1$", etc.). The general form of a linear expression is:

$$C + \sum_{i=0}^{n-1} c_i \cdot x_i,$$

where n is the number of symbols in the expression, C is the trailing constant, c_i are the constant coefficients, and x_i are symbols, with $C, c_i \in \mathbb{Z}, \forall i \in [0, n]$ (adapted from [10]).

5 INVARIANT GENERATION

In this section we provide an overview of the proposed method of loop invariant generation.

5.1 The Basic Concept

The fundamental idea is to keep track of the program state as the loop is symbolically executed, thus allowing us to generate accurate concrete intervals for the values that certain symbols may have upon termination of the loop of interest. By constantly joining intervals between loop iterations, we can expand these intervals to cover the entire range of values of the given symbols across the entire execution of the loop. These bounds can then be trivially converted to invariants representing the concrete numerical bounds of the symbols present in the loop.

For example, if we find that a symbol $i \in [0, 2]$, we can translate this into an invariant stating

$$loop\_invariant\ 0 <= i \ \&\&\ i <= 2$$

5.1.1 Assumptions and Limitations. For our approach, we considered a subset of possible loop expressions. While the main focus has been placed on for loops, these concepts can easily be generalized for *while* and *do . . . while* loops. The for loop must have exactly one counter variable attributed to it. This variable must be present in the initializer of the loop, the loop condition, and the iterator. Furthermore, all sub-expressions of the loop, meaning the initializer, condition, iterator, and instructions in the loop body, must be in the form of linear expressions. For the condition, a comparator must also be present (i.e. $<, \leq, >, \geq$), while for instructions (including the initializer and iterator), an assignment must be made (e.g. $i = i + 1$). Finally, we assume that variables used in the loop are initialized to concrete values before the loop itself. If the loop does not adhere to this format, generation is not possible. The general form of such a for loop is given by

```
initialization
for (int i = start; i op end; i = expression)
    body
```

where $op \in \{<, \leq, >, \geq\}$

5.2 Representation

In order to be able to generate loop invariants, a representational model is necessary. This model stands as the basis of our abstract interpretation of the piece of code provided as input. We consider a for loop to be comprised of the following: the initializer, the condition, the iterator, the body, and the context. By context we understand the set of instructions present before the for loop itself, which must contain the initialization of the program variables.

To go from the SystemC IR representation into a form that we can work with, we translate the initializer into a linear expression. The condition is represented by two linear expressions alongside their relation (i.e., $\{<, \leq, >, \geq\}$). For example, $i < n$. The body is translated into a list of assignment operations (an assignment operation simply being a mapping between a symbol and a linear expression), to which the iterator is added at the end, similarly translated (e.g. $i = i + 1$). Finally, the context is transformed into a symbolic state (including the counter variable). This state is represented by the mapping between the symbol and the symbolic interval for that symbol. All symbols are initially considered to be mapped to an interval containing only their assigned value. For example, if before the loop body we initialize a symbol $n = 5$, then its corresponding initial state is $\{n : [5, 5]\}$.

5.3 High-Level Algorithms

In this subsection we provide explanations for the algorithms used and how they are applied.

5.3.1 The basic example. We begin by means of a simple example. Consider this snippet of code:

```
j = 5;
for (i = 0; i < 5; i = i + 1) {
    j = j - 1;
}
```

Code Snippet 1. Example code

All this piece of code does is increment i while decrementing j. From simply looking at the example, we can infer the bounds $i, j \in [0, 5]$, which can trivially be translated into invariants about the loop, i.e.:

```
loop_invariant 0 <= i;
loop_invariant i <= 5;
loop_invariant 0 <= j;
loop_invariant j <= 5;
```

Code Snippet 2. Desired invariants

5.3.2 Symbolic Execution. Although the basic example is obvious for a person, we need a mechanism to systematically infer this information so we can generate it automatically. The basic algorithm is provided by Algorithm 1.

Algorithm 1 Symbolic Execution

```
1: function symbolicExecute(initial_state, body, cond)
2:     next_state ← initial_state
3:     repeat
4:         current_state ← next_state
5:         next_state ← executeAndJoin(body, current_state)
6:     until current_state = next_state or not satisfies(current_state, cond)
7:     return current_state
8: end function
```

Here *executeAndJoin*, shown in Algorithm 2, is a simple routine that does as the name implies, symbolically executes the body on the *current_state*, which results in a new state. This new state is then joined with the *current_state*, resulting in the next state. What this effectively achieves is a single iteration of the loop body, enlarging the state intervals of our symbols. For example, the instruction $i = i{+}1$ would be stored in the list as $(i, i{+}1)$. Here, the body is an ordered list of tuples, with the first element being the *symbol* on which the transformation is done, while the second is the *transformation* itself. The body also includes the loop update, added as a final instruction.

Figure 1 is a visual representation of how this algorithm works. The point X is the initial value of a symbol, and the squiggly contour represents the set of all possible values that symbol can take during program execution. Subsequent iterations of the symbolic executions provide increasingly large intervals that overapproximate this set.

A **join(⋈)** is an operation applied on two intervals, with the result being a new minimal interval that contains both of the original intervals. This can be viewed as the over-approximation over the union of the two intervals (e.g., $[0, 3] \cup [4, 5] \subseteq [0, 3] \bowtie [4, 5] = [0, 5]$). The **join** of two states is then defined as joining the symbolic interval of a symbol from the first state with the symbolic interval of

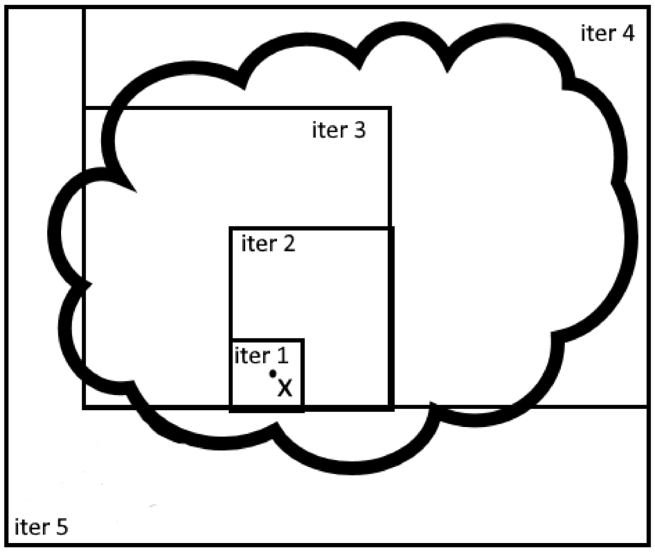

Fig. 1. Visual representation of symbolic execution

the same symbol in the second state, done for all symbols between the two states. If the interval bounds are concrete, then we have

$$[a_1, b_1] \bowtie [a_2, b_2] = [min(a_1, a_2), max(b_1, b_2)].$$

If the interval bounds are not concrete, the join is not computable in the general case (since min and max are not either), unless the difference between the bounds is a constant term.

Algorithm 2 Execute and Join

```
1: function executeAndJoin(body, initial_state)
2:     current_state ← initial_state
3:     for all (symbol, transformation) ∈ body do
4:         update(current_state, symbol, transformation)
5:         reduce(current_state)
6:     end for
7:     result ← join(initial_state, current_state)
8:     return result
9: end function
```

One important part to note of this algorithm is the use of the *reduce* function. This function tries to resolve a symbolic state to its most concrete form by iteratively applying replacement on its symbols. We say that the state of a symbol is concrete if it is not an expression of other symbols in the state (e.g., $\{i : [0, 5]\}$ is a concrete state, while $\{i : [0, N - 1]\}$ is not). By replacement we mean that a symbol a is replaced by its bound in the expressions of the symbolic interval of the symbol b (of course, only if b depends on a), such that we respect the minimum and maximum properties of the bounds of the symbol b. A replacement is considered to respect the minimum (respective maximum) property of the interval bounds if no other replacement exists such that a lower (respective higher) bound can be computed (i.e., this gives us the largest possible interval for the symbol b).

It may be possible, however, that no set of replacements will resolve a symbolic state into a concrete state; thus, we consider the most concrete state. We define the most concrete state as a state that cannot be further reduced by any number of replacements. We are interested in finding the most concrete state because this state yields the simplest possible expressions to be used in the next iterations, but also because we might otherwise reach an expression in the loop condition that cannot be computed otherwise, as finding the smaller (or larger) between two expressions is non-solvable for the general case. This makes the use of non-concrete states generally unusable.

We account for possible mutual dependencies between symbols by applying the *reduce* function between every instruction. For example, a body $i = j + 1, j = i + 1$ would lead to an unsolvable state if the state is not reduced between instructions. This would happen because the first instruction makes the bounds of i into an expression of j, while the second makes the bounds of j an expression of i, thus no replacements can exist that solve this situation.

5.3.3 Resolving to the most concrete state. The algorithm for resolving to the most concrete state is given by Algorithm 3.

Algorithm 3 Reduce State

```
1:  function $reduce(initial\_state)$
2:      $current\_state \leftarrow initial\_state$
3:      $constants \leftarrow getConstants(current\_state)$
4:      $result \leftarrow \emptyset$
5:      while $constants \neq \emptyset$ do
6:          $result.putAll(constants)$
7:          $current\_state \leftarrow current\_state.removeAll(constants)$
8:          $new\_state \leftarrow \emptyset$
9:          for all $(symbol, interval) \in current\_state$ do
10:             $new\_interval \leftarrow evaluate(interval, result)$
11:             $new\_state.put(symbol, new\_interval)$
12:         end for
13:         $current\_state \leftarrow new\_state$
14:         $constants \leftarrow getConstants(current\_state)$
15:     end while
16:     $result.putAll(current\_state)$ ▷ Add all remaining symbols
17:     return result
18: end function
```

The algorithm works by repeating a simple routine of removing concrete symbols and evaluating the rest of the state based on those concrete symbols until this can no longer be done.

5.3.4 Evaluating an interval for a given state. The function *evaluate*, is what ensures that we keep the minimum and maximum properties of our interval states. To evaluate the largest possible bounds for the states, we effectively wish to replace symbols by possible values such that the lower bound becomes the smallest it can after the replacement (respective upper bound becomes the largest). Let us consider a simple example. Take the state $\{N : [-2, 2]; i : [5 - N, 5 + N]\}$. We can see that this is not the most concrete form of this state, because the symbol i depends on the symbol N, which is concrete; thus, a replacement is possible. Looking at the lower bound expression of i, $5 - N$, and replacing the symbol by its bounds, we get the new possible bounds $5 - (-2) = 7$ and $5 - 2 = 3$, of which 3 is clearly smaller. We apply this similarly to the upper bound $5 + N$: the possible bounds are $5 + (-2) = 3$ and $5 + 2 = 7$, of which 7 is larger. Then we can conclude that the largest concrete interval for i is $[3, 7]$. This result can be generalized. When we wish to compute a lower bound, we use the largest value for that symbol if its coefficient is negative and the smallest value for that symbol if its coefficient is positive. For the upper bound, the inverse is done: add the largest value if the coefficient is positive or the smallest if the coefficient is negative.

5.4 Verification

5.4.1 Preliminary Verification. With the algorithms described above, we can automatically infer numerical bounds for program variables with respect to the execution of a loop. Looking back at our example 5.3.1, we do indeed get the desired invariants 2.

Plugging the generated and annotated PVL back into VerCors, however, does not verify, even though the bounds are correct. This gives us a verification error: ***This invariant may not be maintained, since this expression may be false***, referring to the expression *loop_invariant* $0 <= this.j$. This happens because VerCors verifies by isolation. While we may have used the loop context, $this.j = 5$, to generate these invariants, VerCors does not take this into account when doing the verification. VerCors isolates the loop, assuming its invariants to be true, then verifies the loop body to see if they are maintained. In our example, VerCors assumes that $j \in [0, 5]$, executes the first iteration of the loop, which results in the next state of $j \in [-1, 4]$, but since $[-1, 4] \nsubseteq [0, 5]$, the invariant cannot be maintained.

5.4.2 Loop Condition Mutation. In order to make our generated invariants verifiable by VerCors, we need to provide it with the additional context in which the loop is executed, thus allowing for verification. What we can do here is apply a mutation to the loop condition that will give the necessary context. We choose to mutate the condition specifically because VerCors uses it as part of the verification process.

This new condition must satisfy two conditions: it must not change the behavior of the loop, and it must provide VerCors with information about the state of the program at the beginning of the loop execution. The proposed mutation for this is adding the program state at the iteration prior to exiting the loop to the condition with a logical **AND** operator. We can obtain this state by slightly altering our *symbolicExecute* function to also keep track of the previous state and return it alongside the final state.

Let us illustrate this concept by applying *symbolicExecute*, Algorithm 1, on the initial example 5.3.1. Translating the loop into our representation, we have

$$\begin{aligned}&\text{the initial state}\{i : [0, 0],\ j : [5, 5]\}\\&\text{the condition } i < 5\\&\text{and the loop body } [j = j - 1,\ i = i + 1]\end{aligned}$$

We will apply the loop body to the state, joining previous states until the loop condition is no longer satisfied. Applying it once, we get the state

$\{i : [1, 1],\ j : [4, 4]\}$,

which joined with the initial state becomes $\{i : [0, 1],\ j : [4, 5]\}$

On the next iteration, we get the state

$$\begin{aligned}&\{i : [0, 2],\ j : [3, 5]\},\\ &\text{then } \{i : [0, 3],\ j : [2, 5]\},\\ &\text{then } \{i : [0, 4],\ j : [1, 5]\},\\ &\text{and finally } \{i : [0, 5],\ j : [0, 5]\}\end{aligned}$$

at which point the loop condition is no longer satisfied. Thus, the final state that satisfies the loop condition is $\{i : [0, 4], j : [1, 5]\}$. Indeed, looking at the interval of the symbol i, it matches the original condition, with the upper bound being equivalent to the condition itself. Let us name the iteration associated with this state $iter_{final}$. Since the symbolic interval of a variable is an over-approximation of the set of all values that symbol can take up to that point of execution in the program, we know that $j \in [1, 5]$ is true at least for all iterations of the loop. This means that a condition $1 <= j \&\& j <= 5$ could not terminate the loop execution before the original condition $i < 5$. It is not necessarily a concern whether the state after the execution of iteration $iter_{final}$ still satisfies this new condition, because combining it with the original condition via a logical **AND** guarantees that the loop cannot run for more iterations than the original. Thus we reach the mutated condition

$$i < 5 \ \&\& \ (1 <= j \ \&\& \ j <= 5)$$

which can be confirmed to not change the behavior of the loop. Although not explicitly tested with non-monotonous functions, this line of reasoning should also hold in such situations.

5.4.3 Correctness. Let us look at this concept more generally. We say that a state S is included in another state S' if the state S' contains all symbols present in state S, and for every such symbol, the interval in the state S is a subinterval to the one S' for that respective symbol. We represent this relation as $S \subseteq S'$. Based on the implementation of *executeAndJoin*, Algorithm 2, at every iteration, the current state is included in the next. This is because the set of symbols does not change, and for every symbol, the resulting interval is joined with the previous (thus guaranteeing the subintervals property). So $S_i \subseteq S_{i+1}$ is true for any iteration i of the loop. By induction, $S_i \subseteq S_{iter_{final}}$ is also true for any iteration i prior to $iter_{final}$.

Consider a condition *cond* that refers to a symbol i present in the state S, with the mapping of i in S satisfying *cond*, for example, $S = \{i : [0, 3], j : [1, 2]\}$ and $cond : i < 5$). Then the state S can be said to satisfy the condition *cond*. We can generate a new condition $cond_S$ that expresses that all symbols in S must be within the bounds described by S. If the condition $cond_S$ holds, then the condition *cond* must also hold by conjunction elimination. For the example, if $(0 <= i \ \&\& \ i <= 3) \ \&\& \ (1 <= j \ \&\& \ j <= 2)$ is true, then $i < 5$ is also true. If a state S satisfies a condition *cond*, then any state $S' \subseteq S$ (that contains all symbols in *cond*) must also satisfy *cond*, because the mapping of all symbols in S' is included in the respective mapping in state S.

Based on this, if we generate a condition that expresses the bounds of state $S_{iter_{final}}$, it must imply the original condition of the loop. And since any prior state of the loop is included in this state, they must satisfy the generated condition. Then this new condition cannot terminate the execution of the loop before the original condition, so combining them by a logical **AND**, we get a condition that is equivalent to the original. Using this equivalent condition, we can now provide VerCors with the additional information necessary for verifying the generated invariants. This works because VerCors can no longer try to apply the loop body on the found bounds directly, which would have let it find states outside of those bounds, resulting in the invariants not being maintained.

5.4.4 Turning mutated conditions into invariants. Mutating the loop condition allows for the verification of concrete numerical bounds of program symbols. While the section above provides an informal proof of equivalence to the original program, there are some downsides still: the possible introduction of errors and the strengthening of the loop condition. It may be possible that modifying the loop condition in this way can introduce unforeseen bugs. To mitigate this, program equivalence verification would be necessary between the original and the modified. Also, the strengthening of the loop condition results in a weaker negation, and since the negation of the loop condition is used after the loop in further proofs, this may lead to failure of verification of additional program properties.

An alternative could be to turn the mutated condition into an invariant itself, using logical implication. For a mutated condition *cond && additional*, where *cond* is the original loop condition, and *additional* is the condition generated based on the symbol states, then we can generate an invariant *cond ==> additional*. This new invariant circumvents the issues of mutating the condition itself, but based on our experiments, we have not been able to verify this invariant. The likely explanation is that we once again have concrete numerical bounds in our invariant, which puts us in the same situation as before mutating the loop condition in the first place.

6 EVALUATION

In this section we present the running examples used throughout the development of the proposed method, alongside their respective invariants and verification status. These examples are generated by the researcher to provide increasingly challenging cases for the proposed method.

6.1 Case Study

The basic case for which invariants have been generated is a simple for loop that sums all the values of its counter variable. This loop satisfies the assumptions presented in section 5.1.1.

```
total = 0;
for (i = 0; i < 5; i = i + 1) {
    total = total + i;
}
```

Case 1. Base loop case

This case is used to validate the generation of bounds for very simple loops, which only have constant terms for the initial value, the condition, and the iterator. Initially, the bounds of *total* have been disregarded until multiple symbols could be supported. Several variations of this case have also been considered, each one presenting additional complexity to the base case (differences highlighted in bold and italic).

```
total = 0;
n = 5;
for (i = 0; i<n; i = i + 1) {
    total = total + i;
}
```

Case 2. Condition with additional symbol

This variation of the loop includes an additional symbol n in the condition. Whether we can generate concrete bounds for i in this case depends on whether we have initialized n with a concrete value.

```
total = 0;
n = 5;
s = 0;
for (i = 0; i < n; i=i+s) {
    total = total + i;
    s = s + 1;
}
```

Case 3. Variable step

This variation introduces a variable step between loop iterations. At this point we are forced to generate bounds for the s variable as well; otherwise, the bounds of i cannot be verified, as i is an expression of s.

6.2 Results

For base case 1, we get the following generated invariants:

```
1   loop_invariant 0 <= this.i;
2   loop_invariant this.i <= 5;
3   loop_invariant 0 <= this.total;
4   loop_invariant this.total <= 10;
```

Code Snippet 3. Invariants for base case (simplified)

Like mentioned earlier, we disregard the bounds of *total* (lines 3 and 4) for the time being. Plugging invariants 1 and 2 into VerCors results in successful verification. If we were to also add the invariants about *total* at this point, verification fails for similar reasons to section 5.4.1. As discussed in section 5.4.2, we can apply condition mutation to this loop, which yields the new condition *this.i* < 5 && (0 <= *this.total* && *this.total* <= 6). Adding this in, verification is now successful.

For case 2, the numerical bounds of i and i are the same as in the base case; however, we now also get an invariant about n being a constant term:

```
loop_invariant 0 <= this.i;
loop_invariant this.i <= 5;
loop_invariant this.n == 5;
loop_invariant 0 <= this.total;
loop_invariant this.total <= 10;
```

Code Snippet 4. Invariants for condition case (simplified)

In this case, the mutation of the condition does not require the inclusion of the symbol n, as it is a constant term and we can simply add the associated invariant. The difference between n and *total* is that VerCors can internally represent n as a single constant value, while *total* is considered a range. Since n is a constant, it can be directly replaced in expressions containing it; thus, the invariant already provides all necessary information for verification.

Finally, for the variable step (case 3):

```
loop_invariant 0 <= this.i;
loop_invariant this.i <= 6;
loop_invariant this.n == 5;
loop_invariant 0 <= this.s;
loop_invariant this.s <= 3;
loop_invariant 0 <= this.total;
loop_invariant this.total <= 4;
```

Code Snippet 5. Invariants for variable step case (simplified)

Here we cannot verify unless we add the mutated the condition *this.i* < *this.n* && (0 <= *this.s* && *this.s* <= 2) && (0 <= *this.i* && *this.i* <= 3) && (0 <= *this.total* && *this.total* <= 1). It is important to note the added condition that $i \in [0, 3]$, which is different from the initial condition $i < 5$. However, observing the actual behavior of the loop, the conditions are equivalent. The symbol i starts at 0 and is monotonically increasing; thus $i \geq 0$. When i reaches the value 3, s is equal to 3, so the next value of i will be 6, which is greater than 5; thus, we stop iterating and $i \leq 3$. From these two observations, the old and new conditions are equivalent for this particular loop. Verifying this with VerCors is successful.

6.3 Discussion

The evaluation of the presented cases shows that the proposed method of generating numerical bounds as loop invariants is promising. This method is able to correctly capture the behavior of different *for* loops, including the effect of execution on the program state. There are, however, some points to consider, especially with respect to the applicability of these results across various SystemC designs.

The assumptions needed to apply this technique are fairly strong. It is reasonable to assume that symbols can only take integer values, as per the specification of VerCors, but it is not always the case that all instructions are simple assignments made of linear expressions. This severely shrinks the set of programs for which this method is applicable. However, for simple loops iterating over array entries, for example, this seems like a useful technique. It is difficult to judge how much the cases presented are representative of other common uses of SystemC, mainly due to the limited experience of the researcher with this framework.

Additionally, the validity of condition mutation in section 5.4.2 is not fully explored. Based on the informal proof and examples provided, it seems to be a working concept, but a proper formal proof would be necessary to ensure correctness. Furthermore, its limitations are still unknown, as there might be cases for which it gives a false result. One known issue is that it strengthens the loop condition, which may be an issue for verification software that relies on the negation of the condition in proofs.

The main strength of this method for generating loop invariants lies in that it can provide invariants that may be difficult to deduce but are still reasonably easy to verify manually should anything go wrong.

7 CONCLUSION

In this research project, VeSUV has been extended to support the generation of loop invariants that specify numerical bounds for symbols used in and mutated by the execution of the loop. This has been achieved through the use of symbolic execution on abstract interval domains, which can generate upper and lower bounds for each individual symbol present in the loop. These invariants have, however, limited usability in verification of the initial designs, mainly due to the way VerCors isolates the loop during verification. This means that context, which allows the assertion of the generated invariants, is not provided, leading to unmaintained invariants. To circumvent this limitation, we also propose a way to mutate the loop condition, which is used in the verification step, such as to reintroduce the missing context, thus creating of an equivalent but verifiable transformation.

For the rest of this section, we provide answers for the research questions posed in section 2.

SRQ: What types of loop invariants can be generated based on loop bound analysis? The kind of invariants found refer to numerical bounds of program variables. These invariants are based on the symbolic execution of the target piece of code, alongside the context in which said piece of code resides. These bounds represent over-approximations over the actual set of possible values for a given symbol. Being over-approximations, correctness of the bounds can be ensured, but in exchange for a loss of granularity. Depending on the actual mutation of the symbols across iterations, it is likely to represent states that are impossible in practice, which may result in issues with verification of specific properties.

SRQ: What kinds of loops can have invariants generated? The proposed method imposes a set of assumptions and limitations on the types of loops for which it can be applied. One major limitation currently present is the necessity to initialize all symbols used in the loop. This noticeably shrinks applicability. If, for example, a registered routine requires reading sensor input, this would make generation impossible for that loop. To alleviate this concern, assumptions about the possible ranges of values for all symbols would be necessary. An additional restriction is on the form of individual instructions. Currently, support is limited to only linear expressions, which are a suitable starting point but are likely unable to capture the complexity of possible designs.

SRQ: How effective are solutions found for generating useful loop invariants? The proposed solution is able to generate invariants useful in the verification of simple designs. How effective it is in the context of real-world applications, which are likely significantly more complex and possibly not within the assumptions, remains unclear still, and further exploration should be done.

RQ: How can loop bound analysis techniques be used in loop invariant generation for embedded system design verification? Loop bound analysis techniques, namely abstract interpretation through the use of interval abstract domains, show promising applicability in the field of embedded systems design verification. The found technique can provide numerical bounds for the set of possible values that a program variable may have throughout the execution of a loop, as well as before and after, in the form of the generated invariants. Although said invariants rely on having a definite initial state, and they themselves represent concrete numerical bounds, both of those may still be reasonable assumptions in the context of embedded systems design. We cannot immediately expect that all symbols are initialized to concrete values, but it is fair to assume that the range of possible values for any specific input symbol is known during the design phase. Similarly, having concrete numerical bounds for different symbols could likely still be useful, for example, for ensuring safe access of memory locations.

7.0.1 Future Work. Based on the current findings of this study, we propose some directions possibly worth exploring further in future research. One immediately obvious path is looking into the applicability of these results in real-world designs, trying to find the exact limits of the proposed approach. Another line of study could try to further generalize these findings, potentially relaxing the assumptions being made, or attempt to find bounds as relations between symbols instead of concrete values. Finally, loop condition mutation could be explored more in-depth, first by means of a proper formal proof of equivalence to the original condition, and second by trying to generalize this concept such that it may be used with non-concrete states.

REFERENCES

[1] Lukas Armborst, Pieter Bos, Lars B. van den Haak, Marieke Huisman, Robert Rubbens, Ömer Şakar, and Philip Tasche. 2024. The VerCors Verifier: A Progress Report. In *Computer Aided Verification*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer Nature Switzerland, Cham, 3–18. https://doi.org/10.1007/978-3-031-65630-9_1

[2] Pavel Čadek, Clemens Danninger, Moritz Sinn, and Florian Zuleger. 2018. Using loop bound analysis for invariant generation. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 1–9. https://doi.org/10.23919/FMCAD.2018.8603005

[3] Pavel Čadek, Jan Strejček, and Marek Trtík. 2016. Tighter loop bound analysis. In *Automated Technology for Verification and Analysis: 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings 14*. Springer, 512–527. https://doi.org/10.1007/978-3-319-46520-3_32

[4] Carlo A Furia and Bertrand Meyer. 2010. Inferring Loop Invariants Using Postconditions. *Fields of logic and computation* 6300 (2010), 277–300. https://doi.org/10.1007/978-3-642-15025-8_15

[5] Juan Pablo Galeotti, Carlo A Furia, Eva May, Gordon Fraser, and Andreas Zeller. 2014. Dynamate: Dynamically inferring loop invariants for automatic full functional verification. In *Hardware and Software: Verification and Testing: 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings 10*. Springer, 48–53. https://doi.org/10.1007/978-3-319-13338-6_4

[6] Juan P Galeotti, Carlo A Furia, Eva May, Gordon Fraser, and Andreas Zeller. 2015. Inferring loop invariants by mutation, dynamic analysis, and static checking. *IEEE transactions on software engineering* 41, 10 (2015), 1019–1037. https://doi.org/10.1109/TSE.2015.2431688

[7] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[8] Sophie Lathouwers. 2023. Exploring annotations for deductive verification. (2023). https://doi.org/10.3990/1.9789036558464

[9] Preeti Ranjan Panda. 2001. SystemC: a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th international symposium on Systems synthesis*. 75–80. https://doi.org/10.1145/500001.500018

[10] Junkil Park, Miroslav Pajic, Oleg Sokolsky, and Insup Lee. 2017. Automatic verification of finite precision implementations of linear controllers. In *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I 23*. Springer, 153–169. https://doi.org/10.1007/978-3-662-54577-5_9

[11] SV-COMP. n.d.. Software Verification Competition (SV-COMP). https://sv-comp.sosy-lab.org/. Accessed April 29, 2025.

[12] Asmae Heydari Tabar, Richard Bubel, and Reiner Hähnle. 2022. Automatic loop invariant generation for data dependence analysis. In *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering*. 34–45. https://doi.org/10.1145/3524482.3527649

[13] Philip Tasche, Paula Herber, and Marieke Huisman. 2024. Automated invariant generation for efficient deductive reasoning about embedded systems. In *International Conference on Software Engineering and Formal Methods*. Springer, 404–422. https://doi.org/10.1007/978-3-031-77382-2_23

[14] Philip Tasche, Raúl E Monti, Stefanie Eva Drerup, Pauline Blohm, Paula Herber, and Marieke Huisman. 2023. Deductive verification of parameterized embedded systems modeled in SystemC. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 187–209. https://doi.org/10.1007/978-3-031-50521-8_9

[15] Iñigo Ugarte and Pablo Sanchez. 2005. Verification of embedded systems based on interval analysis. *International Journal of Parallel Programming* 33 (2005), 697–720. https://doi.org/10.1007/s10766-005-8909-9