Symbolic Detection of Assertion Dependencies for Bounded Model Checking

Grigory Fedyukovich[1], Andrea Callia D'Iddio[2], Antti E. J. Hyvärinen[1], and Natasha Sharygina[1]

[1] Formal Verification Lab, Università della Svizzera italiana, Switzerland
[2] University of Rome Tor Vergata, Rome, Italy

Abstract. Automatically generating assertions through static or run-time analysis is becoming an increasingly important initial phase in many software testing and verification tool chains. The analyses may generate thousands of redundant assertions often causing problems later in the chain, including scalability issues for automatic tools or a prohibitively large amount of information for final processing. We present an algorithm which uses a SAT solver on a bounded symbolic encoding of the program to reveal the implication relationships among spatially close assertions for use in a variety of bounded model checking applications. Our experimentation with different applications demonstrates that this technique can be used to reduce the number of assertions that need to be checked thus improving overall performance.

1 Introduction

An important part of many of the approaches for increasing software quality through formal methods is to infer potential correctness properties from a program. Such properties can be obtained in the form of assertions from the source code, or behavior observed during run time [19,6,17,13,24]. The assertions can then be verified against the source code using static-analysis methods such as model checking [5,22]. In the paper, we study how Bounded Model Checking [1] (BMC) can be used in verifying assertions generated by automated software analysis.

We propose a generic framework for identifying implication relations between assertions, and study how obtaining information about the implication relation between assertions can be used in finding redundant assertions. This knowledge becomes useful when the number of assertions generated automatically grows large. For instance, in our experiments, independently on the settings of the assertion synthesiser, the number is typically in the order of hundreds and sometimes much higher.

The machine-generated assertions are often redundant in the sense that a BMC algorithm only needs to verify a subset of these assertions and can safely skip the rest if the verification was successful. This observation opens new opportunities for speeding up the computationally expensive BMC algorithms. For

example, the DAIKON program invariant generator [10] might produce the following set of assertions for a return value of a function:

$$A_1 : \texttt{assert}(\texttt{ret} = 0); A_2 : \texttt{assert}(\texttt{ret} \leq 0); A_3 : \texttt{assert}(\texttt{ret} \geq 0);$$

Clearly if the assertion A_1 holds then also the assertions A_2 and A_3 hold. In this paper we formalize the intuition that an assertion may imply other assertions and provide an algorithm and an implementation for discovering such implications for programs written in the C language. In our experiments detecting assertions that are implied to avoid redundant checking consistently decreases the time required to compute the set of true assertions in programs where such redundancy exists. We observe a similar positive result in a model checking approach based on *function summarization* [24] where the summary sizes typically decrease by 30% as a result of our algorithm.

Our approach is ultimately based on determining whether an assertion in the software implies another assertion using a SAT query on a propositional encoding of the program. The number of queries in a straightforward approach would be quadratic in the number of assertions which for realistic programs is infeasible. To achieve a level of performance required for a practical approach we use an analysis that is sensitive to the control flow of the program, to the SAT query, and to the distance between assertions to filter out checks which either cannot result or would very unlikely result in detected implications. These optimizations result in the approach having a relatively low overhead and they do not compromise the soundness of the verification approach.

BMC has proven particularly successful in safety analysis of software and has been implemented in several tools, including CBMC [4], LLBMC [18], VeriSoft [14], and FunFrog [24]. While BMC assumes a loop-free approximation of the program, there are several recent techniques for transforming programs into loop-free programs which, if successful, do not sacrifice soundness or completeness of the verification results. Examples of such techniques include unwinding assertions [4], automatic detection of recursion depth [12], k-induction [9], and loop summarization [15]. While we believe that these techniques are compatible with our method for computing assertion implications, we leave this study for future work.

Related Work. To the best of our knowledge both the approach for detecting implications between assertions using bounded model checking and the use of the detected implications to remove redundant assertions for enhancing bounded model checking are new. The implementation of DAIKON [10] includes an approach for pruning dependent assertions. Our approach extends this by considering also assertions that do not appear in the same program location, and by using propositional logic in deducing the relations between assertions. While [8] presents several approaches for generating assertions in implicative form as potential invariants for recursive algorithms, our goal is at removing redundant implied assertions. Furthermore, [8] detects potential implications through procedure return analysis, a straightforward static analysis, clustering, random selection, and context-sensitive analysis, but applies to assertions at the same program location. Yang et al. [26] further extended the idea of assertion implications

to the case of software evolution (to reflect the change impact between program versions). In contrast to these approaches, our method uses BMC encoding and is able to identify redundant assertions at different program locations. A related approach complementary to ours is presented in [16] where the idea is to lift assertions located in the nested function calls towards the main function to achieve verification performance speedup.

We identify two main approaches for synthesizing assertions for a program. The *dynamic invariant detection* exploits software executions obtained, for instance, from regression test suites, to generate likely invariants (see, e.g., [10,19,6]); while the *static invariant detection* uses static analysis of the source code and symbolic execution to construct potentially helpful invariants (see, e.g., the HOUDINI tool [13]). For our experimentation we selected one representative assertion synthesizer from both approaches: the dynamic invariant synthesizer BCT [17], and a tool based on the CPROVER [4,24] framework for static invariant generation.

Finally, decreasing the sizes of function summaries computed through Craig interpolation [7] can be done through proof compression [3] and the careful selection of interpolation algorithms [23]. This work provides an orthogonal approach where the interpolant is optimized on a higher level by dropping unnecessary verification conditions using domain-specific information.

This paper is organized as follows. Section 2 formalizes the bounded model checking framework we use in the paper. Section 3 explains in detail the basic ideas to detect implications between assertions and the techniques to implement them. Section 4 shows the applications of the approach and the experimental results. Section 5 summarizes the results of our work and discusses open problems and starting points for future work.

2 Preliminaries

Our discussion is based on the *unwound static single assignment* (USSA) approximation of the program, where loops and recursive function calls are unwound up to a fixed limit, and each variable is only assigned once. We have implemented the computing of assertion implications for the C language, but follow the usual approach of presenting the theory in a simpler abstract language to render the discussion more approachable. The USSA approximation serves as an intermediary step in transforming a program to a propositional formula for software bounded model checking. It also serves as a framework for unambiguously defining the assertion implications in Sec. 3 and gives a natural interpretation for a distance between assertions in Sec. 3.4.

The USSA approximation is heavily influenced by techniques used in software bounded model checking (see, e.g. [4]) and consists of an ordered sequence of assignments called instructions, guarded by Boolean valued enabling conditions:

Definition 1. *An* Unwound Static Single Assignment *(USSA) approximation is a finite sequence* $U = (S_1, S_2, \ldots, S_n)$ *of* guarded instructions S_i *having the form* $C \to I$ *where* C *is a condition, called* guard*, and* I *is an instruction.*

```
void main(){
  int x, y, z;
  x = some_value();
  y = some_value();
  if (x >= y){
    if (y >= 0){
      assert(x >= 0);
      z = x + y;
      assert(z >= 0);
    }
  }
  x = some_value();
  y = some_value();
  z = f(x, y);
  assert(z >= 0);
  assert(z >= x + y);
  assert(x <= z - y);
}

int f(int a, int b){
  int i = 0;
  while (i < a + b){
   if (i < a){
     i = i + a;
   } else {
     i = i + b;
   }
  }
  assert(i >= a + b);
  return i;
}
```

(a) C code

$$
\begin{aligned}
&true \rightarrow x_1 := \texttt{some\_value}_1; && (1)\\
&true \rightarrow y_1 := \texttt{some\_value}_2; && (2)\\
&x_1 \geq y_1 \wedge y_1 \geq 0 \rightarrow assert(x_1 \geq 0); && (3)\\
&x_1 \geq y_1 \wedge y_1 \geq 0 \rightarrow z_1 := x_1 + y_1; && (4)\\
&x_1 \geq y_1 \wedge y_1 \geq 0 \rightarrow assert(z_1 \geq 0); && (5)\\
&true \rightarrow x_2 := \texttt{some\_value}_3; && (6)\\
&true \rightarrow y_2 := \texttt{some\_value}_4; && (7)\\
&true \rightarrow \mathtt{f}_{a_1} := x_2; && (8)\\
&true \rightarrow \mathtt{f}_{b_1} := y_2; && (9)\\
&true \rightarrow \mathtt{f}_{i_1} := 0; && (10)\\
&(\mathtt{f}_{i_1} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \wedge (\mathtt{f}_{i_1} < \mathtt{f}_{a_1}) \rightarrow \mathtt{f}_{i_2} := \mathtt{f}_{i_1} + \mathtt{f}_{a_1}; && (11)\\
&(\mathtt{f}_{i_1} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \wedge (\mathtt{f}_{i_1} \geq \mathtt{f}_{a_1}) \rightarrow \mathtt{f}_{i_3} := \mathtt{f}_{i_1} + \mathtt{f}_{b_1}; && (12)\\
&(\mathtt{f}_{i_1} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \rightarrow \mathtt{f}_{i_4} := \mathtt{phi}(\mathtt{f}_{i_2}, \mathtt{f}_{i_3}, (\mathtt{f}_{i_1} < \mathtt{f}_{a_1})); && (13)\\
&(\mathtt{f}_{i_1} \geq \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \rightarrow \mathtt{f}_{i_5} := \mathtt{f}_{i_1}; && (14)\\
&true \rightarrow \mathtt{f}_{i_6} := \mathtt{phi}(\mathtt{f}_{i_4}, \mathtt{f}_{i_5}, (\mathtt{f}_{i_1} \leq \mathtt{f}_{a_1} + \mathtt{f}_{b_1})); && (15)\\
&(\mathtt{f}_{i_6} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \wedge (\mathtt{f}_{i_6} < \mathtt{f}_{a_1}) \rightarrow \mathtt{f}_{i_7} := \mathtt{f}_{i_6} + \mathtt{f}_{a_1}; && (16)\\
&(\mathtt{f}_{i_6} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \wedge (\mathtt{f}_{i_6} \geq \mathtt{f}_{a_1}) \rightarrow \mathtt{f}_{i_8} := \mathtt{f}_{i_6} + \mathtt{f}_{b_1}; && (17)\\
&(\mathtt{f}_{i_6} < \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \rightarrow \mathtt{f}_{i_9} := \mathtt{phi}(\mathtt{f}_{i_7}, \mathtt{f}_{i_8}, (\mathtt{f}_{i_6} < \mathtt{f}_{a_1})); && (18)\\
&(\mathtt{f}_{i_6} \geq \mathtt{f}_{a_1} + \mathtt{f}_{b_1}) \rightarrow \mathtt{f}_{i_{10}} := \mathtt{f}_{i_6}; && (19)\\
&true \rightarrow \mathtt{f}_{i_{11}} := \mathtt{phi}(\mathtt{f}_{i_9}, \mathtt{f}_{i_{10}}, (\mathtt{f}_{i_6} \leq \mathtt{f}_{a_1} + \mathtt{f}_{b_1})); && (20)\\
&true \rightarrow assert(\mathtt{f}_{i_{11}} \geq \mathtt{f}_{a_1} + \mathtt{f}_{b_1}); && (21)\\
&true \rightarrow \mathtt{f}_{ret_1} := \mathtt{f}_{i_{11}}; && (22)\\
&true \rightarrow z_2 := \mathtt{f}_{ret_1}; && (23)\\
&true \rightarrow assert(z_2 \geq 0); && (24)\\
&true \rightarrow assert(z_2 \geq x_2 + y_2); && (25)\\
&true \rightarrow assert(x_2 \leq z_2 - y_2); && (26)
\end{aligned}
$$

(b) USSA approximation (bound = 2)

Fig. 1. Converting a C program into USSA

In the process of constructing the USSA approximation the program loop conditions and branches are encoded into guards, while the rest of the encoding consists of constructing assignments. Given an unwinding limit k, a `while` loop is transformed into a chain of k nested `if` constructs in order to represent at most k iterations of the loop. Similarly, recursive functions are inlined k times. To encode values which depend on `if` branches and `while` loops we use the `phi`-function, which returns its first or second argument depending on the truth value of its third argument as follows:

$$\mathtt{phi}(e_1, e_2, e_3) = \begin{cases} e_1 \text{ if } e_3 \text{ is true;} \\ e_2 \text{ otherwise.} \end{cases}$$

Finally, guarded assignment S can be annotated with guarded assertions of the form $A = C \rightarrow assert(C')$ evaluated right after S.

Instead of giving an exact definition for the process of constructing the USSA approximation we give an artificial but illustrative example in Fig. 1 showing how a program written in the C language, on the left, is transformed into the USSA approximation on the right. The program consists of two functions `main` and `f` in addition to a nondeterministically treated function `some_value`.[3] The call from `main` to `f` is inlined on lines 10 - 22. Function input and output parameters are assigned on lines 8, 9, and 23; and the loop inside `f` is unwound 2 times on lines 11 - 15 and again on lines 16 - 20. The six assertions in C code appear on the USSA approximation on lines 3, 5, 24, 25, 26, and 21.

Propositional Encoding. Given a USSA approximation U, the propositional formula $\pi(U)$ consists of the propositional encoding of all guarded assignments of U. We extend the definition of the propositional encoding operator π to guards C and *arguments* C' of assertions A. Once the USSA approximation is converted into a propositional formula, determining the validity of the assertions with respect to the program reduces to conjoining the negations of the assertions and then deciding the unsatisfiability of the resulting formula.

3 The Assertion Implication Relation

The aim of the techniques presented in this paper is to enable efficient analysis of a program with respect to a large number of assertions. We are interested in determining whether, in a given USSA approximation, there are assertions A, A' such that A' holds whenever A holds in all executions of the USSA approximation. For performance reasons we do not compute the above, but instead the *assertion implication relation* (AIR), which consists of a subset of such pairs where the implication follows from the statements between the assertions A and A'. In cases where the program contains this type of redundancy in assertions and AIR is not empty, the information can often be used to significantly improve the efficiency of model checking.

Definition 2. *Given a USSA approximation* $(S_1, \ldots, S_n)$ *containing two assertions* $A_i = C_i \rightarrow assert(C'_i)$ *and* $A_j = C_j \rightarrow assert(C'_j)$*, we say that the assertion* A_i locally implies *the assertion* A_j *iff* $1 \leq i < j \leq n$ *and the following formula is valid:*

$$(\pi(C_i) \rightarrow \pi(C'_i)) \wedge \pi(S_{i+1}, \ldots, S_{j-1}) \rightarrow (\pi(C_j) \rightarrow \pi(C'_j))$$

An alternative way of viewing the definition is to say that A_i locally implies A_j iff the *Hoare triple*

$$\{C_i \rightarrow C'_i\}(S_{i+1}, \ldots, S_{j-1})\{C_j \rightarrow C'_j\}$$

[3] To simplify the discussion we ignore arithmetic overflows and underflows. However, the implementation does not have the limitation.

is valid. Given the valid local implication relation between assertions A_i and A_j, we will refer to A_i as a *stronger* assertion, and to A_j as a *weaker* assertion. Notably, this relation is transitive, but not symmetric.

We present a high-level overview of the algorithm for detecting assertion implications (DAI) in Algorithm 1. The algorithm computes the AIR from a USSA approximation provided as an input. We compute the implication relation in two phases: (i) by detecting classes of *dependent* assertions AD using a syntactic analysis on the USSA approximation first directly on the variables in the USSA form (line 1) and then extending the variable dependency to assertions (line 2); and (ii) by detecting implications among the elements of AD using the propositional encoding of the USSA approximation (line 3) with queries to SAT solvers. In the following subsections we describe in details the three subroutines of the algorithm.

Algorithm 1 DAI(P)

Input: A USSA approximation $P = (S_1, \ldots, S_n)$
Output: AIR — the assertion implication relation
Data:
VD — disjoint sets of variables corresponding to the variable dependency classes;
AD — The assertion dependency relation
1: $VD \leftarrow$ dependent variables in P (see Def. 3)
2: $AD \leftarrow$ dependent assertions in P based on VD (see Def. 4)
3: $AIR \leftarrow \{(i,j) \in AD \mid \textsc{implies}(i,j) = true\}$

3.1 Detecting Dependent Variables

We say that two variables x and y are *dependent* when the value of x potentially affects the value of y. The idea of variable dependencies dates back to *program slicing* [25]. We adapt this notion to the USSA approximation of a program, and create dependencies from the assignments and the guards. For example, after the execution of a guarded assignment $G \to x := E$, the updated value of x may depend on the values of variables in E and G. However, due to the final propositional encoding the assignment creates potential dependency also from x to all the variables in E and G. To obtain an over-approximation of the dependency relation, it is enough to assume that all the variables in a guarded instruction depend on each other. The dependency relation is reflexive, transitive and symmetric and therefore an equivalence relation which groups all variables into *dependency classes*. This leads to the following definition of the dependency relation:

Definition 3. *Two variables x, y are said to be* directly dependent *if there exists a guarded instruction $S = C \to I$ such that $\{x, y\} \subseteq Vars(S)$. The general* dependency *relation is the transitive closure of direct dependency.*

Computing the dependency relation from the USSA form can be done efficiently with a union-find algorithm. Furthermore, since the local implication only considers guarded instructions between two assertions, it is sufficient to compute the dependency between two assertions that are currently being checked.

3.2 Finding Assertion Dependencies

To speed up the assertion checking by reducing the number of implication checks, the dependency relation should be extended from variables to assertions. Two assertions A and A' are said to be dependent if there exists a variable x in A and a variable x' in A' such that x and x' are dependent. Unlike a variable, if an assertion A depends on an assertion A', and the assertion A' depends on an assertion A'', this does not imply that A depends on A'', since the dependencies might result from variables not shared by A and A''.

The assertion dependency relation of a USSA approximation is constructed using a variable dependency relation (line 2 of DAI). This is an iterative procedure over the set of assertion pairs. For each pair, it explores the dependency classes of the variables involved in the assertions. If two assertions contain variables of the same dependency class, the assertions are dependent and are going to be included into the relation AD. The dependency of assertions is defined as follows:

Definition 4. *Two assertions* $A_1 = C_1 \rightarrow assert(C_1')$ *and* $A_2 = C_2 \rightarrow assert(C_2')$ *are* dependent *if there is a variable* $x_1 \in Vars(A_1)$ *and a variable* $x_2 \in Vars(A_2)$ *such that* x_1 *and* x_2 *are dependent.*

Example 1. Based on the definitions 3 and 4, the dependent assertions in Fig. 1(b) include $assert(x_1 \geq 0)$ and $assert(z_1 \geq 0)$ on lines 3 and 5; $assert(z_2 \geq x_2 + y_2)$ and $assert(x_2 \leq z_2 - y_2)$ on lines 25 and 26; and $assert(\mathtt{f}_{i_{11}} \geq \mathtt{f}_{a_1} + \mathtt{f}_{b_1})$ and $assert(z_2 \geq x_2 + y_2)$ on lines 21 and 25. The assertions $assert(z_1 \geq 0)$ and $assert(z_2 \geq 0)$ on lines 5 and 24 are not dependent since the set of common symbols is empty (program variables x, y, and z were reassigned independently on the previous values).

3.3 Finding Assertion Implications

In the last phase of constructing AIR, the assertion dependency relation is refined to contain only the pairs of assertions (A, A') such that A locally implies A'. This is done by constructing the formula corresponding to Def. 2 and invoking the SAT solver through the IMPLIES call on line 3 in Algorithm 1.

Example 2. The assertion implication relation computed from the USSA approximation given in Fig. 1(b) consists of $(assert(x_1 \geq 0), assert(z_1 \geq 0))$ on lines 3 and 5, and $(assert(z_2 \geq x_2 + y_2), assert(x_2 \leq z_2 - y_2))$ on lines 25 and 26.

Finally, the AIR defines an *assertion implication graph* representing all revealed implication relationships between the guarded assertions. More formally,

Definition 5. *Given a USSA approximation* $U = (S_1, \ldots, S_n)$*, the* assertion implication graph *of* U *is a graph* $G_U = (V, E)$ *where*

$$V = \{A_i \in U \mid A_i \text{ is an assertion}\}$$

and

$$E = \{(A_i, A_j) \mid A_i \text{ locally implies } A_j \text{ in } U\}.$$

The algorithms proposed in this section are able to detect implications of assertions only in forward direction (i.e., the assertion on the left-hand side of an implication should be located before the assertion on the right-hand-side according to the USSA approximation). However the algorithms might be adapted to deal also with reverse direction. The next subsection will consider other optimizations making the approach applicable in practice.

3.4 Further Optimizations

Given a USSA approximation with k assertions, if Algorithm 1 identifies all assertions as dependent, the total number of assertion implication checks is $k(k-1)/2$. For USSA approximations containing thousands of assertions this number can be prohibitively large. On the other hand, if an assertion implication check needs to be performed between two assertions, one of which is close to the beginning of the USSA approximation and the other which is close to the end of the USSA approximation, the resulting formula might be very large. This often results in the check being computationally expensive. In the experiments we use a threshold to skip checking dependencies of assertions if there are more than n instructions between them in the USSA approximation. For example, none of the pairs of assertions (line 3 and 24; line 3 and 25; line 5 and 26) will be checked for a threshold 5. In our applications this does not break the soundness of the approach, since checking a subset of pairs of assertions will just under-approximate the assertion implication graph. In case when an assertion implication remains undetected, the approach will need to perform more work compared to the case when the assertion implication would have been discovered.

4 Applications

Algorithm 1 returns the assertion implication relation *AIR*, which then can be used for various BMC applications that deal with large sets of assertions. In this section we present two of those applications, namely *Optimizing Assertion Checking Order* and *Assertion Implication Checking in Function Summarization.*

We will study two research questions related to two different applications of assertion implication checking in this section.

R1 In the first application a BMC tool checks the validity of a set of assertions. We determine whether the number of verification runs can be reduced by skipping assertions whose validity is implied by already performed checks and *AIR*.

R2 In the second application a BMC tool constructs *function summaries* based on a set of assertions. We study whether excluding weak assertions using *AIR* reduces the size of function summaries.

Implementation. We implemented the approach for detecting assertion implications (DAI) as a preprocessor for the FunFrog [24] tool. FunFrog is built on

top of CBMC [4], features interpolation-based function summarization (see the description in Section 4.2) for C programs, and uses the OpenSMT solver [2] for solving propositional formulas and interpolation.

The tool uses the CProver[4] framework. In particular, it accepts a pre-compiled goto-binary, a representation of the C program in an intermediate goto-cc language which is further unwound to create the USSA approximation. The analysis is then conducted on this USSA approximation. For each pair of assertions being checked, the analysis identifies the USSA steps corresponding to the assertions and the instructions between them. The USSA form is then bit-blasted and sent to OpenSMT for solving. FunFrog was run with the default configuration which employs for instance an implementation of slicing [24].

We evaluated the performance of FunFrog+DAI in summarization-based BMC on a range of academic and industrial benchmarks widely used in model checking experiments. The assertions for the benchmarks were obtained from the user, the BCT dynamic assertion generator [17] that internally uses Daikon [10], and static invariant synthesizers implemented in FunFrog [24]. In the rest of the section, we describe the details of both applications and provide experimental evidence of the positive effect from using DAI.

4.1 Optimizing Assertion Checking Order

Dynamic analysis tools such as Daikon are often used for producing assertions. Such tools observe program behavior to form a set of the expressions over values of the program variables, which is then turned into a set of assertions V. Since the assertions are obtained by monitoring the execution of the program over a limited set of input parameters, there is no guarantee that such assertions hold for every execution of the program. BMC is used in [20] to check which of those assertions hold. While precise, a model checking run might consume a significant amount of time and require high amounts of memory. Therefore any optimization in the process immediately renders the technique more applicable to a wider set of benchmarks.

Given the USSA approximation U of a program that contains a set of assertions $V \subseteq U$, let $G_U = (V, E)$ be its assertion implication graph. We propose to traverse G_U during the BMC run to minimize the search for holding assertions and avoid checking all assertions one by one. Our solution is based on the two following ideas: 1) If an assertion A_i is proven to hold, all weaker assertions A_j (i.e., $\{A_j \in V \mid (A_i, A_j) \in E\}$) are implicitly proven to hold. 2) If an assertion A_k is proven to fail, all stronger assertions A_j (i.e., $\{A_k \in V \mid (A_j, A_k) \in E\}$) are implicitly proven to fail.

We further expand these ideas into the two complementing strategies for the efficient detection of assertions which hold in the program. We denote the nodes of G_U that do not have incoming edges as $\{A_s\}$. These correspond to the *strongest* assertions in the program. Similarly, we denote the edges with no

[4] `http://www.cprover.org/`

Bench	#USSA Steps	#Asserts	#Checks	Strategy	#DAI Impl	DAI Time	FunFrog+DAI	FunFrog
token_ring	11769	108	34	F	90	36.5	312.4	498.0
mem_slave	2843	146	116	F	61	24.6	70.9	108.9
ddv	537	152	103	F	93	14.9	162.1	240.2
diskperf	1730	192	34	B	172	75.8	65.5	332.5
s3	1733	131	47	B	265	4.4	20.6	55.5
cafe	2686	146	101	B	97	42.2	216.3	301.8

Table 1. Verification of a set of assertions by FunFrog and FunFrog+DAI. The timing values are given in seconds.

outgoing edges as $\{A_w\}$, and these correspond to the *weakest* assertions in the program.

In the first (*forward*) strategy, a BMC tool traverses G_U starting from $\{A_s\}$ in the depth-first order. For each assertion node A_i, if there exists a holding predecessor A_j, the BMC tool concludes that A_i also holds. Otherwise, it verifies the program with respect to A_i. This strategy is efficient in cases when there are many holding assertions in the program.

Example 3. Given the USSA approximation in Fig. 1(b) and its assertion implication graph, in the forward strategy, a BMC tool starts with checking the assertion $assert(x_1 \geq 0)$ on line 3 and proves that it holds. Then, the tool skips checking assertion $assert(z_1 \geq 0)$ on line 5. Next, the tool proves $assert(z_2 \geq x_2 + y_2)$ on line 25 and skips checking $assert(x_2 \leq z_2 - y_2)$ on line 26. To terminate model checking, the tool iteratively checks assertions $assert(\mathtt{f}_{i_{11}} \geq \mathtt{f}_{a_1} + \mathtt{f}_{b_1})$ on line 21 and $assert(z_1 \geq 0)$ on line 24. The forward strategy results in checking 4 of 6 assertions. We expect the overall performance speed up to be approximately 30%.

In the second (*backward*) strategy, a BMC tool traverses G_U in reverse, starting from $\{A_w\}$. For each assertion node A_k, if there exists a failing successor A_j, the BMC tool concludes that A_k also fails. Otherwise, it verifies the program with respect to A_k. This strategy is efficient in cases when there are many assertions which fail in the program.

Example 4. Given the USSA approximation in Fig. 1(b) and its assertion implication graph, in the backward strategy, a BMC tool explicitly checks all 6 assertions. Since all assertions hold in the given example, this strategy does not produce any performance speed up.

Experiments. We report the effect of DAI on the assertion checking in Table 1. In the experiment we are given a benchmark (represented as a USSA form with the corresponding **#USSA Steps**) and a set of assertions (**#Asserts**). First, FunFrog+DAI constructs the *AIR* (that reveals **DAI Impl** implications and takes **DAI Time** (*excluded from* **FunFrog+DAI**)). Then, FunFrog+DAI proceeds to assertion verification following one of the two strategies (**Strategy** = **F** (forward) or **B** (backward)), in which **#Checks** was actually performed. Finally, we compare the time spent on verification by **FunFrog+DAI** with the time needed to verify each assertion by the vanilla **FunFrog**. The assertions for these benchmarks come from the BCT tool.

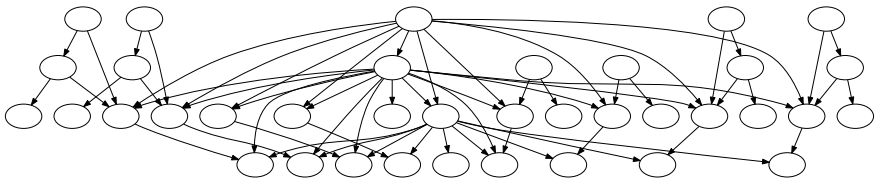

Fig. 2. The assertion implication relation for benchmark instance `mem_slave`. Note that the figure only contains assertions that imply another assertion.

In all our benchmarks FunFrog+DAI is able to reduce the total number of checks needed to perform the verification. In the best case scenario we observe run times that are more than two times faster than the vanilla FunFrog (see, `diskperf`). Note that for benchmarks containing many redundant assertions it is possible to detected more implications than the number of existing assertions in the code. For example, the benchmark instance `s3` has 131 assertions but over 200 implications. We illustrate the redundancy of assertions in Fig. 2 showing the assertion implication relation computed for the benchmark instance `mem_slave`.

4.2 Assertion Implication Checking in Function Summarization

FunFrog is an incremental model checker that maintains a set of *function summaries* in order to speed up consequent verification runs and checking correctness of software upgrades [11]. FunFrog relies on partitioning the assertion set V into smaller disjoint subsets $\{\mathcal{A}\}_0^k$. Each set $\mathcal{A}_i \subseteq V$ is then checked with a separate run of the model checker. FunFrog encodes the program into the USSA approximation (see Sec. 2 and Fig. 1). FunFrog conjoins the USSA approximation with disjunction of the negations of the assertions $a \in \mathcal{A}_i$ to be checked. The resulting *BMC formula* ϕ_i is then bit-blasted and sent to the SAT solver. If it is proven that ϕ_i is unsatisfiable then the program is correct with respect to $\mathcal{A}_i$ and the proof of unsatisfiability can be used to over-approximate function behaviors by means of Craig interpolation.

In propositional logic, for every unsatisfiable pair of formulas (A, B) there exists an interpolant I that can be constructed from the proof of unsatisfiability [21] and has the properties that $A \rightarrow I$ and $I \wedge B$ is unsatisfiable [7]. For each function call f and the set of assertions $\mathcal{A}_i$, we define the BMC formula ϕ_i as $\phi_i \equiv A_f \wedge B_{\mathcal{A}_i}$, where A_f encodes the function call f and $B_{\mathcal{A}_i}$ encodes the rest of the program and the assertions from the set $\mathcal{A}_i$. Given a proof of unsatisfiability, we use an interpolating solver to generate the *function summary* for the function call f as an interpolant I_f, such that $A_f \rightarrow I_f$.

While verifying the program with respect to another set of assertions $\mathcal{A}_j \neq \mathcal{A}_i$, the BMC formula ϕ_j is constructed in such a way that the precise encoding of function calls is replaced by a function summary. By construction, a summary is accurate enough to prove the set of assertions $\mathcal{A}_i$. However, for $\mathcal{A}_j$, it may contain infeasible error paths due to the over-approximating nature of Craig

interpolants. In this case, ϕ_j is satisfiable, and FunFrog identifies the summaries responsible for the satisfiability. To continue the verification, FunFrog needs to replace responsible summaries by the precise function representations. It is worthwile to try to avoid this scenario through better organization of the checking since the procedure is computationally expensive and requires another FunFrog iteration. On the other hand, if ϕ_j is unsatisfiable then the substitution of summaries was sufficient to prove $\mathcal{A}_j$. In such cases verification with summaries is often faster than with the exact encoding. Independently of the result of the SAT solver, the size and the logical strength of Craig interpolants in ϕ_j affect the verification behavior of FunFrog [23].

The assertion implication graph $G_U = (V, E)$ can be used to reduce the size of function summaries. We propose to construct each subset $\{\mathcal{A}\}_0^k$ of V while traversing G_U. The method is based on the following observation. If each assertion $A \in \mathcal{A}_i$ is implied by some assertion $A' \in \mathcal{A}_j$ then the summaries constructed from BMC formula ϕ_j will be sufficient to prove both $\mathcal{A}_j$ and $\mathcal{A}_i$. On the other hand, if no implication is found between assertions $A \in \mathcal{A}_j$ and $A' \in \mathcal{A}_i$ then there is no guarantee that the summaries constructed from ϕ_j will be sufficient to prove $\mathcal{A}_i$. We propose to use the AIR to identify the set of strongest assertions and perform the verification only on this set. As a result we expect to obtain a strong summary that due to the simplicity of the resulting formula will be more compact (as our following experimental results confirm).

Example 5. Consider the example in Fig. 1. There are six assertions, which can be verified one by one[5], having been partitioned into singleton sets. Two of them (A_1 and A_2 at lines 3 and 5 respectively) are located before the function `f` is called, and do not rely on the function behavior. After one of them is verified, the summary of function `f` is going to be created. A likely summary of function `f` with respect to the assertions A_1 and A_2 is simply the formula $I_{\mathtt{f},A_1} \equiv I_{\mathtt{f},A_2} \equiv \mathit{true}$.

There are three assertions after call to `f`. Once they are verified, the summary reflects the behavior of `f` (lines 24, 25, 26): (A_3): $\mathit{true} \rightarrow \mathit{assert}(z_2 \geq 0)$, ($A_4$): $\mathit{true} \rightarrow \mathit{assert}(z_2 \geq x_2 + y_2)$, ($A_5$): $\mathit{true} \rightarrow \mathit{assert}(x_2 \leq z_2 - y_2)$. For example, after verifying assertion A_3, the summary of function call `f` should reflect that the return value of the function `f` is never negative, i.e., $I_{\mathtt{f},A_3} \equiv \mathtt{f}_{ret} >= 0$. In the next run, while verifying assertion A_4, the function call `f` is replaced by the previously computed summary $I_{\mathtt{f},A_3}$ which relates the returned variable and a constant 0. Since A_4 relies on a more sophisticated relation over the return value and the values of input/output parameters, this substitution is likely to lead to a spurious counterexample and, consequently, to an expensive further refinement, i.e., repeating verification from scratch.

Similarly, after successful verification of A_4, a new summary $I_{\mathtt{f},A_4}$ is generated. The summary relates the return value and the values of input/output parameters (i.e., $I_{\mathtt{f},A_4} \equiv \mathtt{f}_{ret} \geq \mathtt{f}_\mathtt{a} + \mathtt{f}_\mathtt{b}$). After two iterations, the resulting summary of `f` is a conjunction $I_{\mathtt{f},A_3,A_4} \equiv I_{\mathtt{f},A_3} \wedge I_{\mathtt{f},A_4}$. In the next run, while

[5] We intentionally chose only holding assertions to demonstrate how summary construction and its usage works.

Benchmark	#USSA Steps	#Asserts	#DAI Impl	DAI Time	#V	#Cl	#V'	#Cl'
floppy	15076	721	134	26.38	228357	11973	228659	12879
diskperf	6000	47	7	0.083	150413	49362	162902	83625
gd_simp	673	21	5	0.138	6091	15420	12119	33504
two_expands	183	4	1	0.033	735	1221	1087	2277
p2p_joints	759	146	24	1.71	158034	452427	307897	902016
goldbach	7502	1344	65	25.82	6159	13455	13237	34689

Table 2. Creation of function summaries by FunFrog and FunFrog+DAI. The timing values are given in seconds.

verifying assertion A_5, the function call `f` is replaced by $I_{\mathtt{f},A_3,A_4}$. Since A_5 is essentially the same as A_4, this substitution should be sufficient, so no refinement is needed to complete verification.

In this example, assertion implication checking can be used to simplify the model checking process in two ways. First, the *AIR* reveals that $A_4 \rightarrow A_5$, i.e., it is enough to show that A_4 holds to show that A_5 also holds. Second, no dependency is detected between A_3 and A_4, suggesting that no matter in which order the two are checked, it is likely that a refinement is needed afterwards. In order to avoid the expensive refinement procedure, it makes sense to combine the two assertions into a single verification run.

Experiments. Table 2 reports statistics on constructing function summaries with FunFrog when DAI was used as a preprocessor. Similarly to the experiment from Sec. 4.1, we are given a benchmark (with the size of **#USSA Steps**) and a set of assertions (**#Asserts**). First, FunFrog+DAI constructs the *AIR* (with **DAI Impl** relations). Then FunFrog+DAI obtains the set of strongest assertions to be encoded to the BMC formula, solved and used to create function summaries. Finally, we calculate the total number of variables and clauses in the resulting summary formula (**#V** and **#Cl** respectively). We compare these values with the ones collected after the vanilla FunFrog run (**FunFrog** time, **#V'** and **#Cl'** respectively). For these benchmarks we obtained the assertions using the cprover library underlying FunFrog.

The experimentation demonstrates that on our benchmark set the proposed approach improves the performance and the effect of BMC in the context of interpolation-based function summarization. Using particular optimization techniques (i.e., threshold for assertion locations and timeout for implication checks), in many cases it was possible to reduce the overhead of performing the implication checks. Note that at least in these benchmarks the construction of *AIR* requires a considerably smaller amount of time than needed for the actual assertions checking in the classic BMC approach.

5 Summary and Future Work

We presented a simple but effective approach to reveal the implication relationships between spatially close assertions. This technique addresses the problems arising from large number of redundant assertions in bounded model checking

and, as our experimentation on benchmarks containing redundant assertions demonstrates, in many cases reduces the total verification time. We observe a similar positive result in a model checking approach based on function summarization, where the summary sizes typically decrease by 30%.

As a potential future optimization we consider improving the condition in Def. 2. A propositional encoding π considers the instructions between a pair of assertions but does not take into account the variables assigned with equal values before the first assertion. For instance, in Fig. 1(b), $assert(\mathtt{f}_{i_{11}} \geq \mathtt{f}_{a_1} + \mathtt{f}_{b_1})$ and $assert(z_2 \geq x_2 + y_2)$ on lines 21 and 25 imply each other, but the implication between them can be proved only if two additional assignment instructions (on lines 8 and 9) are added to the SAT-query. While including all the USSA program to every implication query would likely be overly expensive, including parts of this information to the checking process would potentially increase the number of detected implications and pay off as a result of decreased assertion checks.

The assertion implication checking could be further improved heuristically by using more intelligent ways of ordering assertion implication checking. One approach for ordering the checking is to identify *likely implications*, as discussed in [8]. Another interesting source of assertions that is not discussed in this work is to consider also semantical properties of pointers and stack contents. However for this to work in practice it is likely that an alias analysis should be performed as a preprocessing step to reduce the number of assertion candidates.

Acknowledgements. We thank the reviewers for their valuable feedback. The work was supported by the SNF project number 138078. This work has been done during an internship of the second author at the Verification Lab of the Faculty of Informatics, Università della Svizzera Italiana, Lugano, Switzerland. The second author would like to thank the Verification Lab for this important collaboration, and at the same time he thanks Prof. Maurizio Talamo from the Tor Vergata University of Rome, for strongly encouraging this collaboration.

References

1. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS '99. LNCS, vol. 1579, pp. 193–207. Springer (1999)
2. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT solver. In: TACAS '10. LNCS, vol. 6015, pp. 150–153. Springer (2010)
3. Cabodi, G., Lolacono, C., Vendraminetto, D.: Optimization techniques for Craig interpolant compaction in unbounded model checking. In: DATE '13. pp. 1417–1422. EDA Consortium San Jose, CA, USA / ACM DL (2013)
4. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS '04. LNCS, vol. 2988, pp. 168–176. Springer (2004)
5. Clarke, E., Emerson, A.: Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In: Logic of Programs: Workshop. LNCS, vol. 131. Springer (1981)
6. Cobb, J., Jones, J.A., Kapfhammer, G.M., Harrold, M.J.: Dynamic invariant detection for relational databases. In: Proc. International Workshop on Dynamic Analysis 2011. pp. 12–17. ACM (2011)

7. Craig, W.: Three uses of the Herbrand-Genzen theorem in relating model theory and proof theory. Journal of Symbolic Logic 22(3), 269–285 (1957)
8. Dodoo, N., Donovan, A., Lin, L., Ernst, M.D.: Selecting predicates for implications in program analysis (2002), available at `http://homes.cs.washington.edu/~mernst/pubs/invariants-implications.ps`
9. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In: TACAS '10. vol. 6015, pp. 280–295. Springer (2010)
10. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering 27(2), 99–123 (2001)
11. Fedyukovich, G., Sery, O., Sharygina, N.: eVolCheck: Incremental Upgrade Checker for C. In: Proc. TACAS '13. LNCS, vol. 7795, pp. 292–307. Springer (2013)
12. Fedyukovich, G., Sharygina, N.: Towards Completeness in Bounded Model Checking through Automatic Recursion Depth Detection. In: Proc. SBMF '14. LNCS, vol. 8941, pp. 96–112. Springer (2014)
13. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Proc. FME '01. LNCS, vol. 2021, pp. 500–517. Springer (2001)
14. Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Efficient SAT-based bounded model checking for software verification. Theoretical Computer Science 404(3), 256–274 (2008)
15. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using state and transition invariants. Formal Methods in System Design 42(3), 221–261 (2013)
16. Lal, A., Qadeer, S.: A program transformation for faster goal-directed search. In: Proc. FMCAD '14. pp. 147–154. IEEE (2014)
17. Mariani, L., Pastore, F., Pezzè, M.: Dynamic analysis for diagnosing integration faults. IEEE Transactions on Software Engineering 37(4), 486–508 (2011)
18. Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In: Proc. VSTTE '12. LNCS, vol. 7152, pp. 146–161. Springer (2012)
19. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Using dynamic analysis to discover polynomial and array invariants. In: Proc. ICSE '12. pp. 683–693. IEEE (2012)
20. Pastore, F., Mariani, L., Hyvärinen, A.E.J., Fedyukovich, G., Sharygina, N., Sehestedt, S., Muhammad, A.: Verification-aided regression testing. In: Proc. ISSTA '14. pp. 37–48. ACM (2014)
21. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. Journal of Symbolic Logic 62(3), 981–998 (1997)
22. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Proc. 5th Colloquium on International Symposium on Programming. LNCS, vol. 137, pp. 337–351. Springer (1982)
23. Rollini, S.F., Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: PeRIPLO: A framework for producing effective interpolants in SAT-based software verification. In: LPAR '13. LNCS, vol. 8312, pp. 683–693. Springer (2013)
24. Sery, O., Fedyukovich, G., Sharygina, N.: FunFrog: Bounded model checking with interpolation-based function summarization. In: Proc. ATVA'12. LNCS, vol. 7561, pp. 203–207. Springer (2012)
25. Weiser, M.: Program slicing. In: Proc. ICSE '81. pp. 439–449. IEEE (1981)
26. Yang, G., Khurshid, S., Person, S., Rungta, N.: Property differencing for incremental checking. In: Proc. ICSE '14. pp. 1059–1070. ACM (2014)