Instantiation-Based Interpolation for Quantified Formulae in CSIsat

Vadim Mutilin
Institute for System Programming, RAS
mutilin@ispras.ru

Mikhail Mandrykin
Institute for System Programming, RAS
mandrykin@ispras.ru

***Abstract*—The paper describes an implementation of instantiation-based interpolation for quantified formulae in modified CSIsat tool. The tool supports interpolation for formulae with linear real arithmetic, uninterpreted functions and quantifiers. We propose in this paper using external SMT-solver CVC3 for quantified expressions instantiation, then we describe how we modified CSIsat and CVC3 tools in order to support quantified formulae interpolation. We also present results of benchmarking the modified CSIsat tool on SMTLIB test set as well as on our specially generated interpolation tasks.**

***Index Terms*—interpolation, Craig interpolant, quantifiers, instantiation, solver, axioms.**

I. Introduction

Among several currently predominant model checking techniques, predicate abstraction is one of the most widespread and successful approaches. This success is significantly promoted by the advance of SLAM2 [1] static verification tool, which uses this approach and which is extensively used in WDDK(Windows Driver Developer's Kit) tool set for static verification of windows device drivers. Program predicate abstraction is built with logical predicates. The set of all possible valuations of the predicates from the abstraction forms an abstract domain partitioning the total program state space into subsets with same predicate valuations[14]. The particular challenge here in predicate abstraction is identifying necessary predicates, since they determine the accuracy of the abstraction. In most state-of-the-art predicate abstraction tools predicate choice is fully or mostly determined by the program considered. So one of the most important problems arising with the use of predicate abstraction is the derivation of the predicate set suitable for the verification of each particular program. Most frequently used decision for that problem is using the CEGAR approach [12] — Counter-Example Guided Abstraction Refinement. The approach is based on the iterative construction of the program abstraction starting with the most coarse one and continuing consistently with one or several successive abstraction refinements based on corresponding infeasible counterexamples that arise from the verification of the coarser abstraction. So when the coarser abstraction gives a spurious (infeasible) counterexample, the fact of its infeasibility is somehow used for refinement of that inaccurate abstraction.

Among the modern static program verification tools which use predicate abstraction and implement the CEGAR approach, SLAM2[1], BLAST[7], [23] and CPAchecker[8] are the most extensively used in practice. These tools implement abstraction refinement in two rather similar and still a little different ways. All the tools somehow build a logical formula corresponding to the counterexample considered. SLAM2 uses weakest precondition for the statement sequence of the counterexample. BLAST and CPAchecker both build a path formula based on the SSA (Static Single Assignment) form. As soon as the counterexample is spurious, its weakest precondition and a path formula are unsatisfiable, if build precisely enough. This fact is used by the tools to derive new predicates and refine current abstraction. Here software model checkers bring some special tools into play. SLAM2 uses heuristic predicate derivation from the *unsatisfiable core* (i.e. a small unsatisfiable subset of clauses) of the counterexample's weakest precondition. The satisfiability check and unsatisfiable core extraction are performed by special tool called *SMT-solver* (SMT stands for Satisfiability Modulo Theories) intended to decide logical formulae with respect to combinations of background theories expressed in classical first-order logic with equality. SLAM2 uses Z3[13] SMT solver. Other syntax-based predicate derivation techniques[16] are used in SLAM2 as well. BLAST and CPAchecker derive new predicates locally for selected program locations (such as loop heads, functions calls or just any program statement) from *Craig interpolants* of the two path formula parts before and after each of such a location point. A Craig interpolant for a mutually inconsistent pair of formulae (A, B) is a formula that is implied by A, inconsistent with B, and expressed over their common uninterpreted symbols (variables and uninterpreted functions occurring both in A and in B). To find Craig interpolants both BLAST and CPAchecker use special *interpolating SMT-solvers*. An interpolating SMT-solver extends a decision procedure by taking a conjunction of a pair of logical formulae as an input and by producing one of the two possible results: either answer `SAT` — if the input conjunction turns out to be satisfiable — or a Craig interpolant for that conjunction (since in this case the conjunction is inconsistent and the corresponding Craig interpolant always exists by the Craig interpolation theorem). BLAST can use either FOCI[18] or

This work was partially supported by FTP "Research and development in priority areas of scientific and technological complex of Russia in 2007-2013" (contract number 11.519.11.4006)

CSIsat[9] interpolating solvers and for CPAchecker the same CSIsat and MathSAT[10] are suitable.

The predicates derived form the counterexample analysis are intended to eliminate the spurious execution path form the abstraction and thus prove the absence of corresponding particular error. And when the predicates obtained from Craig interpolation of the path formula are guaranteed to rule out the infeasible counterexample by definition (provided that the obtained interpolants are inductive, i.e. each subsequent interpolant is implied by the conjunction of the previous one with the corresponding path formula part), the predicates produced from the heuristic approaches not necessarily succeed. Another advantage of interpolation as a technique for refinement is that it not only discovers new predicates, but also determines the control locations at which these predicates are useful. At the same time syntax-based predicate derivation approaches are also complete for certain classes of programs and their actual implementations quite rarely fail to discover necessary predicates.

One of the most significant SLAM2 advantages over current versions of BLAST and CPAchecker tools is its good precision (which is judged by the number of given false alarms, i.e. spurious counterexamples the tool considers feasible) in analysis of programs with significant use of pointers, including the ones to dynamically allocated memory regions. Meanwhile the tool doesn't use any special analysis for dynamically allocated data structures (like Shape-analysis[6]) as well as any special background logical theory for heap objects representation (e.g. separation logic [22]). A simplified location-based logical memory model is used instead. This model is proposed in paper [2] to be used for efficient evaluation of pointer predicates with a modern SMT solver (such as Z3). The paper authors suggest using the special axiom set and representing pointer predicates with uninterpreted functions of integer values.

We decided to investigate the possibility of using some similar logical memory model in BLAST or CPAchecker. As we have already stated above, the new predicates derivation in these tools is based on Craig interpolation of several parts of the unsatisfiable path formula [15].

The conjunction of the ordered pair of formulae corresponding to the two parts of the infeasible error path is unsatisfiable. Hence there exists a Craig interpolant for the pair. The predicates occurring in the interpolant are used by BLAST and CPAchecker tools in refining current program abstraction in order to eliminate the infeasible counterexample. An interpolating decision procedure (also called "interpolating prover", "interpolating solver" or just "an interpolator") is used in obtaining the interpolant. Both BLAST and CPAchecker use interpolating solvers (CSIsat and MathSAT) for quantifier free fragments of the logical theories of linear real arithmetic (LA) and uninterpreted functions with equality (EUF). The solvers are able to find quantifier free interpolants for such formulae. When using a logical memory model with an axiom set we need also to find interpolants for quantified formulae. At the same time, the getting of quantified interpolants for such formulae is also acceptable.

This way we see that SLAM2 with logical memory model for pointers needs only a good SMT solver with the support of quantifiers and unsatisfiable core extraction. But an implementation of a similar model in either BLAST or CPAchecker tool would essentially need an interpolating decision procedure for quantified formulae. But even after a thorough search by the time of our investigations we hadn't found any suitable tool for that purpose.

At the same time a paper [11] presents an extension of McMillan's algorithm[18] for instantiation-based quantified formulae interpolation. The formulae may be given with respect to an arbitrary combination of background theories for which the original McMillan's algorithm is applicable. The extended McMillan's algorithm gives us a rather simple way to find possibly quantified interpolants for such formulae in case the set of required quantified expression instantiations is known in advance. This instantiations must be sufficient for proving the given input conjunction unsatisfiable.

The idea of implementing the extended McMillan's algorithm also appeared to be interesting in account of the fact the verifiers usually perform several corresponding SMT solver queries just before the interpolation. And the majority of modern SMT solvers implement quantified formulae interpolation support through the instantiation of quantified subexpressions. This means that if the solver finishes with an `UNSAT` result providing a proof of the input formula unsatisfiability, the interpolating solver will that way have the necessary instantiations in advance. So long as the extended McMillan algorithm's implementation in case of a priori given necessary quantifier instantiations is reasonably easy, this implementation might be as well used for preliminary benchmarking the logical memory model efficiency in static software verifiers using interpolation for abstraction refinement. This way we decided to implement the extended McMillan's algorithm based on some existing interpolating prover and an SMT solver with quantifier support. We also decided to estimate the efficiency of the new tool on specially generated benchmarks simulating the interpolation tasks a real model checker could give to our tool.

II. Our approach

To apply an extended McMillan's algorithm proposed in [11] we need a resolution proof of input conjunction unsatisfiability with necessary quantified expression instantiations used and so-called partial interpolants in the leafs of the tree evaluated. The tree can be obtained in several ways. One such way is to implement some quantifier instantiation heuristic (e.g. e-matching[20]) in a tool currently implementing original McMillan's interpolation algorithm. The other way is to use an external tool successfully implementing such heuristics, say SMT solver. In this case the necessary instantiations can be extracted from the unsatisfiability proof given by the solver.

If we only extract necessary instantiations from the proof, then the formula unsatisfiability will be discovered twice: once by the SMT solver supporting quantifiers (to obtain

instantiations) and then again by the interpolator, here with necessary instantiations and without quantifiers, — to extract the desired interpolant from the proof. Meanwhile we can't use the unsatisfiability proof from the SMT solver for the interpolation directly, as according to McMillan's algorithm we need a specific unsatisfiability proof using inference rules significantly different from the ones used in modern SMT-solvers. Yet the state-of-the-art SMT solvers are significantly more efficient than any of the interpolating decision procedures we knew to implement McMillan's algorithm (they were FOCI, CSIsat and several experimental implementations). So when using an external SMT solver the overhead of proving the formula unsatisfiable once again is more on the interpolator's side.

Despite this significant overhead arising from using an efficient SMT-solver together with considerably less efficient interpolating decision procedure we eventually decided to implement the approach due to its relative simplicity. For that we had to choose a suitable existing interpolating solver meeting the following requirements:

- The solver should implement the McMillan's interpolating algorithm form the paper [18]. This was required as the algorithm presented in [11] is an extension of this McMillan's algorithm.
- The solver was required to support interpolation for logical formulae with respect to the combination of theories used by verification tools BLAST and CPAchecker in their interpolation queries. These are the theories of linear integer (LIA) and real (LRA) arithmetics and the theory of uninterpreted functions with equality (EUF). Their combination is often referred as LA+EUF.
- The source code of the solver should be freely available for modification and the solver should be distributed under an appropriate license.

Among existing interpolating decision procedures we knew and considered the only one to meet all the requirements was CSIsat[9]. It's implemented in OCaml and its components responsible for reading an input formula, preprocessing it, deciding its satisfiability, generating partial interpolants and combining them are placed in several fairly independent modules. So it turned out we only needed to implement modified versions of some of the modules and then use them whenever an input formula included some quantifiers.

Our relatively simple approach chosen used only a set of necessary quantifier instantiations and had no need in thorough analysis of unsatisfiability proof produced by the SMT solver. So the primary criteria for the choice of an SMT solver were its support of quantifiers, high performance and also, preferably, the availability of its source code for easy integration. Based on the criteria we choose CVC3[5] SMT solver. There the most relevant issue of the solver was the use of quite many complicated and coarse-grained inference rules in its proofs. The issue seemed to be minor at the moment as we only needed to extract relevant quantifier instantiations and not to process the entire proof. But later it turned out that sometimes necessary instantiations are not included in the final proof by the solver, which causes the modified interpolator to terminate abruptly.

III. Related work

The significant overhead of proving the input conjunction unsatisfiable the second time in the interpolating solver suggests the idea of another interpolation technique. The transformation of the SMT solver proof tree into the inference system appropriate for inductive interpolant derivation is the another approach essentially different from the one proposed in this paper. This approach is also greatly differs from both the one used in CSIsat and the one described in the paper [11]. The approach is reported in paper [19] and quite possibly performs much better than the one we consider here. The paper [19] proposes this approach for the Z3 SMT solver. Its application for another common SMT solver for quantified formulae — CVC3 — is complicated with great number of inference rules used by the tool. We ought to mention also that the paper mentioned was published a couple of months after we had finished implementing the tool presented in this paper. We only state the implementation details and efficiency benchmarking results relevant to our modified version of CSIsat tool onwards.

IV. Implementation details

The modified CSIsat tool supports interpolation of quantified formulae with respect to the combination of the two background theories: the theory of linear real arithmetics and the theory of uninterpreted functions with equality (LA+EUF). Our implementation is based on the latest version of the tool from its original developers, CSIsat 1.2 dated back to July, 2008. The tool uses the patched version of CVC3 SMT solver permitting easy extraction of necessary quantifier instantiations. Let us itemize the modifications we made upon the CSIsat and CVC3 tools in order to implement our approach.

A. CSIsat tool modifications

- The modified version of CSIsat tool has the extended input formula format compatible with the one used in the FOCI interpolating solver (implementing the approach presented in [18]). The input formula syntax has been extended with the designations for existential and universal quantifiers.
- The modified tool supports interpolation for pairs of formulae only. With the use of an extra option one can specify three input formulae: A, B and C. The formula C must be a conjunction of universally quantified expressions. In this case the interpolant is produced for the pair of formulae $(A, B \wedge C)$, but the free symbols from the formula C are considered to be common for the formulae A and $B \wedge C$. Here universally quantified expression from C are in this way considered somewhat like theory axioms.
- We implemented some preliminary transformations of the input formula before reducing it into the conjunctive

normal form. Each formula of the input problem is subject for the following transformations:

- selection of the topmost quantified subexpressions,
- reduction of the selected subexpressions into the prenex normal form,
- *skolemization* of the subexpressions.

Skolemization is a way of removing existential quantifiers from a formula. Variables bound by existential quantifiers which are not inside the scope of universal quantifiers are simply replaced by constants. And when the existential quantifiers are inside the universal quantifiers, the bound variables are replaced by Skolem uninterpreted functions of the variables bound by the universal quantifiers.
If after the transformation the formula still includes at least one universal quantifier the extended McMillan's interpolation algorithm is applied. Otherwise the original algorithm is used.

- The support for SMTLIB v.2 [4] as output format was added. In case of quantified formula interpolation the transformed conjunction $A \wedge B$ is passed to the modified CVC3 SMT solver. If the formula is proven unsatisfiable, the modified tool also produces the set of essentially used quantifier instantiations. If the conjunction turned out to be satisfiable, both CVC3 and CSIsat terminate with `Satisfiable` verdict. As the SMT solver is incomplete in presence of quantifiers the `Unknown` verdict is also possible.
- The algorithm of *purification* of the essential quantifier instantiations from the *mixed terms* accordingly to the algorithm presented in [11] was implemented. If the conjunction is proven unsatisfiable, CSIsat purifies the obtained instantiations building auxiliary hash tables that contain the information about common symbols and newly introduced variables. It also computes the reflexive transitive closure of the inverse of the *support* relation over the newly introduced variables using the Warshall-Floyd algorithm.
- The extended McMillan's interpolation algorithm for quantified formulae was implemented. The algorithm works upon the proof tree obtained from CSIsat internal SMT solver together with the purified instantiations and the auxiliary hash tables generated on the previous steps. The algorithm may in general produce a quantified interpolant.

B. CVC3 tool modifications

The interpolation implementation requires the solver to produce necessary quantifier instantiations whenever it ends up with an `Unsatisfiable` verdict. To obtain the instantiations the generated unsatisfiability proof with explicit quantifier instantiation inference rules may be used. The CVC3 inference system has four such instantiation rules:

$$\frac{T \vdash \forall \bar{x}.e(\bar{x})}{T \vdash e(\bar{t})}\ universal\_elimination1$$

$$\frac{T \vdash \forall \bar{x}.e(\bar{x})}{T \vdash \psi \implies e(\bar{t})}\ universal\_elimination2,3$$

$$\frac{T \vdash \forall \bar{x_1}\bar{x_2}.e(\bar{x_1},\bar{x_2})}{T \vdash \psi \implies \forall \bar{x_2}.e(\bar{t},\bar{x_2})}\ partial\_universal\_instantiation$$

where $\bar{x} = (x_1, ..., x_n)$ is a bound variables vector, e is an expression in which the variables $x_1, ..., x_n$ occur free, $\bar{t} = (t_1, ..., t_n)$ is a vector of substitution terms, whose variables must occur free in $\forall \bar{x}.e$. For the $universal\_elimination1$ rule each x_i and t_i for every $i = \overline{1,n}$ must be of the same type. For other rules they should have the same basic types and in this case ψ is a predicate restricting the possible valuations of the substituted terms (to the subtype domain).

But it appeared in practice that CVC3 doesn't always include these inference rules into the final unsatisfiability proof, but sometimes replaces the branches emerging from quantifier instantiations with a much simplified inference rule of the form:

$$\frac{}{T \vdash e(\bar{t})}\ assump$$

Therefore we implemented a heuristic considering some unfinished proof tree fragments in search for the occurrences of quantifier instantiations. We implemented an instantiation simplification heuristic using the CVC3 internal expression simplification capabilities as well. The heuristics are switched with corresponding options.

V. Results

A. SMT-LIB benchmark set results

The modified tool has been tested on two distinct benchmark sets. The first one was obtained from the SMT-LIB[3] benchmark set by dividing the unsatisfiable formulae from the AUFLIA (AUFLIA stands for Arrays, Uninterpreted Functions and Linear Integer Arithmetic) and AUFLIRA (AUFLIRA stands for Arrays, Uninterpreted Functions, and Linear Integer and Real Arithmetic) logics randomly into two sub-formulae at the top-level conjunctions. The interpolation problems were translated into the modified CSIsat input format. Here all integer variables and functions were replaced with real ones and array operations (i.e. `select` and `update`) were represented using uninterpreted functions with the appropriate axiom set. Some of the problems became ill-posed (as the conjunction turned satisfiable) after the transformation and some other yielded degenerate interpolants e.g. `true` and `false`. The benchmarking was performed on the binary optimized version of the tool with the time limit of 5 s and the memory limit of 1 GiB. The results of the benchmarking in both the categories (AUFLIA and AUFLIRA together) are presented in table I.

TABLE I
SMT-LIB BENCHMARK SET RESULTS.

	Results	Number of tests	%
65.8% OK	Quantified interpolant	167	0.63%
	Ground interpolant	551	2.09%
	Degenerate interpolant `true`	8259	31.38%
	Degenerate interpolant `false`	8342	31.70%
	Satisfiable input conjunction	1	0.00%
34.2%	CVC3 gave `Unknown` verdict	536	2.04%
	Time limit exceeded (5 s)	7846	29.81%
	Memory limit exceeded (1 GiB)	71	0.27%
	Insufficient instantiation set	515	1.96%
	Miscellaneous errors	31	0.12%
	Total	26319	100.00%

B. Performance on specially generated benchmarks

The second benchmark set was generated as a collection of specially made-up simulated interpolation tasks a real model checker could give to our modified CSIsat tool in case it implemented a logical memory model similar to that of SLAM2 tool. The location-based logical memory model for pointer predicate derivation uses the following five uninterpreted functions:

- $A(l)$ — returns the address of the location l,
- $L(a)$ — returns the location corresponding the address a,
- $S(x, f)$ — returns a location for a composite type field or an array element, here x is the location of the composite type (or array) as a whole and f is the desired field number (or array index), counting from 0,
- $B(l)$ — returns the location of the a composite type (or an array) by the location of its element (l),
- $O(l)$ — returns the composite type field number by its location (l).

Here is the corresponding axiom set:

$$\forall x.(x > 0 \implies A(x) > 0) \tag{1}$$
$$\forall l.L(A(l)) = l \tag{2}$$
$$\forall a.A(L(a)) = a \tag{3}$$
$$\forall x.\forall f.S(x, f) > N \tag{4}$$
$$\forall x.\forall f.B(S(x, f)) = x \tag{5}$$
$$\forall x.\forall f.O(S(x, f)) = f \tag{6}$$

Locations here are denoted with strictly positive integer numbers. The first axiom states that the address of every normal basic location is strictly positive. The second and the third axioms together specify the functions $A(l)$ and $L(a)$ as mutually inverse for all the locations and their corresponding addresses. The fourth axiom states that the location of the composite type field (or an array element) do not coincide with any basic location. Basic locations correspond to explicitly allocated memory objects such as variables, arrays and structures as a whole. They all have location numbers in the range from 1 to a certain constant N. The composite field locations on the other side must have location numbers strictly greater than N, which is stated in the axiom (4). The axioms (5) and (6) specify the functions $B(l)$ and $O(l)$ and state that the elements of distinct composite values correspond to the distinct locations. Otherwise we'd get:

$$\begin{gathered} S(x_1, f_1) = S(x_2, f_2) \implies \\ B(S(x_1, f_1)) = B(S(x_2, f_2)), \\ O(S(x_1, f_1)) = O(S(x_2, f_2)) \implies (5, 6) \implies \\ x_1 = x_2, f_1 = f_2 \end{gathered}$$

To denote the location values we used a set of uninterpreted functions (V_i). Each such function corresponded to a state of the whole program memory between its two sequential updates. Here we didn't anyhow optimize the memory updates, so each of them gave an expression of the following form:

$$V_{i+1}(l_{upd}) = v \wedge \forall l.(l \neq l_{upd} \implies V_{i+1}(l) = V_i(l))$$

where l_{upd} is the number of the location whose value is updated to v.

The benchmarks making significant use of structures and arrays were performed with a slightly modified axiom set to take the structure and array first element address property into account. The address of a structure (or an array) is equal to the address of its first field (or element). So for such benchmarks we changed the axioms (4), (5) and (6) with the following ones correspondingly:

$$\forall l.A(l) = A(S(l, 0))$$
$$\forall x.\forall f.(f \neq 0 \implies S(x, f) > N) \tag{7}$$
$$\forall x.\forall f.(f \neq 0 \implies B(S(x, f)) = x) \tag{8}$$
$$\forall x.\forall f.(f \neq 0 \implies O(S(x, f)) = f) \tag{9}$$

Here the location of a structure or an array as a whole is merged with the address of its first field. The axioms (4), (5) and (6) are restricted to all the fields of an aggregate except the first one.

Let's illustrate the process of converting the pointer predicate derivation problem into the one of interpolating the quantified formula for one particular cut-point of an infeasible program error path. Here we use the following example from our benchmark set:

```
s_1->f_1 = 1;
s_1 = s_1->next;
s_1->f_1 = 2;
s_1 = s_1->next;
...
s_1->f_1 = n - 1;
s_1 = s_1->next;
s_1->f_1 = n;
s_2 = s_1;
s_1 = s_1->next;
-----------------------
check (s_2->f_1 != n);
ERROR:
```

Here n is a static parameter assigning the size of the test generated. When $n = 2$ the test will look like this:

```
s_1->f_1 = 1;
```

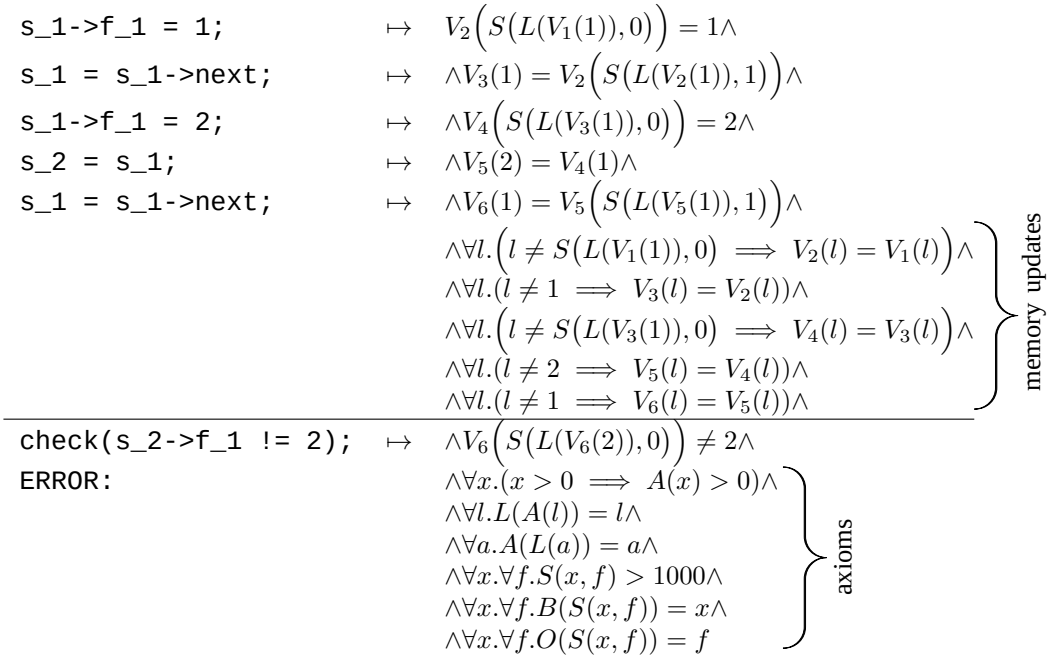


Fig. 1. Path formula example.

```
s_1 = s_1->next;
s_1->f_1 = 2;
s_2 = s_1;
s_1 = s_1->next;
----------------------
check(s_2->f_1 != 2);
ERROR:
```

Here we traverse a linked list of at least two elements and sequentially assign the values 1 and 2 to its elements with the help of the pointer `s_1`. The pointer `s_2` is assigned the address of the second element of the list. The `check` operator is used to designate the chosen branching condition. This means that the condition in the operator must be met on the considered error path. This way the necessary condition of getting onto the error label `ERROR` is the inequality of the second list element value to 2. It can't be fulfilled in case the program behaves correctly in terms of memory operations. The new pointer predicate proving the error path infeasibility should be derived for the cut-point marked with the dashed line separator.

Both the upper and the lower parts of the counterexample path correspond to certain logical formulae. For the newly derived predicates to make the abstraction eliminate the infeasible counterexample, the predicates must be implied by the logical formula for the upper part of the path and be unsatisfiable in conjunction with the formula for its lower part. Also to be possibly used again to eliminate other similar error paths the predicates should include only the values the program variables possess in the cut-point (not before and not after it). Otherwise it won't be possible to make the new abstraction independent from the currently considered counterexample. These requirements for the derived predicates exactly match the definition of the Craig interpolant for a pair of logical formulae, i.e. a third formula that is implied by the first formula, inconsistent with the second one, and expressed over their common uninterpreted symbols (variables and uninterpreted functions occurring in both formulae). Let's consider the formulae and the interpolant produced by our modified CSIsat tool for that particular counterexample. Here let the location number 1 correspond to the variable `s_1` and similarly for the location number 2 and the variable `s_2`, let the list structure have two fields (`f_1` and `next`) corresponding to the numbers 0 and 1, and finally let the constant N be equal to 1000. The path formula of the counterexample is shown in figure 1.

The interpolant produced by our modified CSIsat tool is

$$V_6\Big(S\big(L(V_6(2)),0\big)\Big) = 2 \vee S\big(L(V_6(2)),0\big) < 1000$$

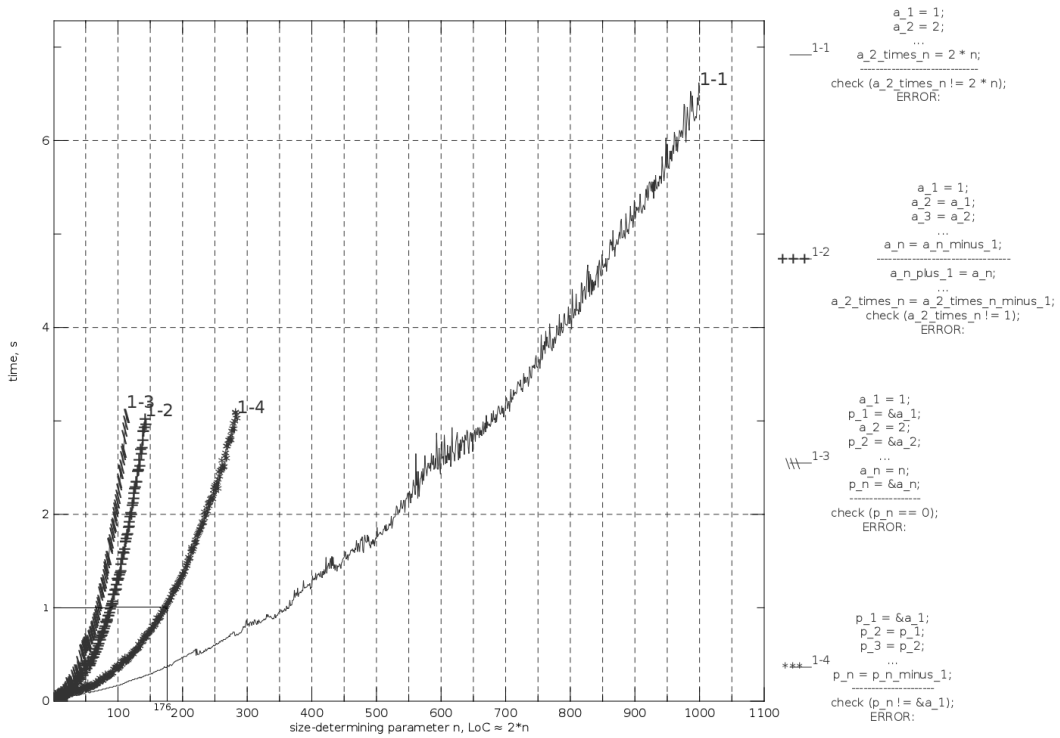
that corresponds to the predicate

$$V\Big(S\big(L(V(2)),0\big)\Big) = 2$$

that is `s_2->f_1 == 2`. The condition is indeed met in the specified program location and proves the infeasibility of the counterexample. The same predicate can be used to prove the infeasibility of another error path through the same program location, e.g.:

```
s_1->f_1 = 1;
s_1 = s_1->next;
s_1->f_1 = 2;
s_2 = s_1;
s_1 = s_1->next;
```

Fig. 2. Results on specially generated benchmarks (simplest cases).




```
----------------------
check(s_2->f_1 == 2); // 'else' branch
                      // in the same 'if'
                      // statement
s_1->f_1 = s_2->f_1 - 2;
check(s_1->f_1 != 0);
ERROR:
```

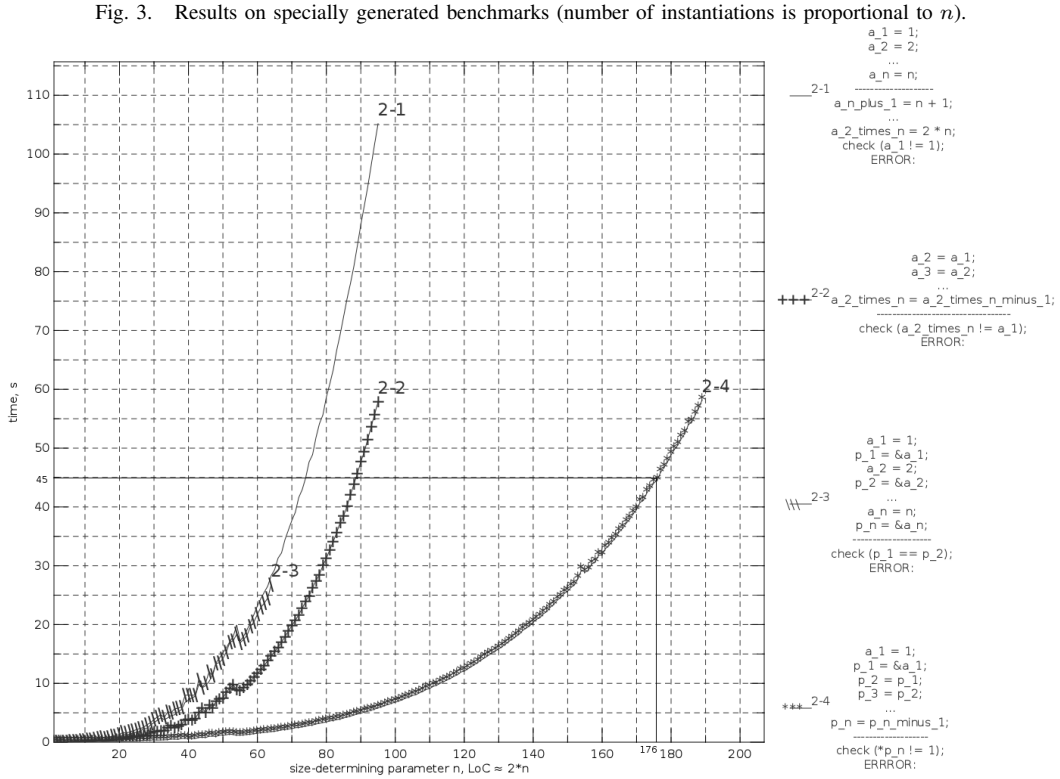
The results of benchmarking our tool on the specially generated example set are shown in figures 2 and 3. The legends of the plots contain code snippets corresponding to the path formulae used in each of the examples. The parameter n specifies the sizes of the test cases so that they contain approximately $2n$ lines of code. The cut-point location dividing the counterexample path into parts is marked with a dashed line and the `check` operator designates chosen branching condition the same way as in the previously considered example. The label `ERROR` is unreachable in the examples so the corresponding path formulae are unsatisfiable. They are built the using the uninterpreted functions presented above and are passed to the modified CSIsat tool with options `-extrInsts` and `-simplInsts` (the options enable our instantiation extraction and simplification heuristics in CVC3). All the benchmarks were launched with the time limit of 600 seconds and memory limit of 1 GiB. The launching for each of the examples was aborted after five abrupt terminations of the tool (including time, memory limit exceeding and uncaught exceptions).

Figure 2 presents the results for very simple cases without manipulating any structures or arrays. They also require no quantifier instantiations of memory update expressions (each variable value is used just after the assignment). Figure 3 presents the results for the very similar test cases. But these ones require the number of quantifier instantiations that is linear in proportion to the number of lines of code in the example. The plots show exponential growth of interpolation time in all the tests. But the tests requiring many quantifier instantiations took up to 45 times and even more greater amount of time for interpolation in comparison with the similar tests requiring very few instantiations (see the plots in figures 2 and 3 for $n \approx 176$).

For the benchmarking results on less trivial cases see the figures in the complete version of this paper ([21]).

Overall the results have shown that our modified tool can produce interpolants for quantified formulae with relatively small number of quantifier instantiations. But the size of real-life counterexamples, e.g. in verifying Linux kernel drivers, is about 1000 to 10000 lines of code and the number of interpolator calls is about 10 to 30 per driver (see the paper [17] for details). This supposes that to achieve a suitable verification time of about 15 minutes per a driver, the tool should produce interpolants for the formulae for about 5000 LoC in about 45 seconds. As the presented results have shown, the tool can process formulae for not more than about 350 (176×2) LoC in that time even in the very simple cases (see figure 3). The modified tool thus turned out to be not efficient enough to be used in a scalable model checker implementing

Fig. 3. Results on specially generated benchmarks (number of instantiations is proportional to n).



the logical memory model considered. This inefficiency can be explained with relative inefficiency of CSIsat internal decision procedure as well as the use of some heuristics for necessary instantiation extraction. The heuristics rather frequently extract unnecessary instantiations. They sometimes also fail to extract the necessary ones causing the tool to terminate abruptly.

VI. Conclusions

In this paper we proposed a relatively easy approach to implement an instantiation-based interpolating decision procedure for quantified formulae based on an existing interpolating solver and a modern SMT solver with quantifier support. We have implemented our approach in the modified version of the CSIsat interpolation tool using CVC3 as an external SMT solver. The modified tool implements Craig interpolation for quantified formulae with respect to the theories of linear real arithmetic and uninterpreted functions with equality.

We have also performed a preliminary benchmarking of our modified tool on two distinct benchmark sets. The first was obtained form the SMT-LIB benchmark set while the second contained the specially generated benchmarks simulating the interpolation tasks that a real model checker could give to our tool in case it implemented a location-based logical memory model for the discovery of pointer predicates.

The tool benchmarking results on our both test sets have shown that our proposed approach is generally functional, but lacks enough efficiency to be used for predicate derivation in a real industrial model checker.

The CSIsat internal decision procedure used in the tool to derive the unsatisfiability proof tree for interpolation is a way less efficient than that of CVC3 and other state-of-the-art SMT solvers. Hence one of the interesting further research areas here is more comprehensive analysis of CVC3 unsatisfiability proof trees in order to use them for reducing the overhead of proving the formula unsatisfiable once again in the interpolator itself. This way we can avoid the most significant burden bounding the efficiency of our tool.

References

[1] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. SLAM2: Static driver verification with under 4% false alarms. In *Formal Methods in Computer Aided Design*, 2010.

[2] Thomas Ball, Ella Bounimova, Vladimir Levin, and Leonardo De Moura. Efficient evaluation of pointer predicates with z3 smt solver in slam2. Technical report, 2010.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[4] Clark Barrett, Aaron Stump, Cesare Tinelli, Sascha Boehme, David Cok, David Deharbe, Bruno Dutertre, Pascal Fontaine, Vijay Ganesh, Alberto Griggio, Jim Grundy, Paul Jackson, Albert Oliveras, Sava Krstić, Michal Moskal, Leonardo De Moura, Roberto Sebastiani, To David Cok, and Jochen Hoenicke. C.: The smt-lib standard: Version 2.0. Technical report, 2010.

[5] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[6] D. Beyer, T.A. Henzinger, and G. Théoduloz. Lazy shape analysis. *Proc. CAV, LNCS*, 4144:532–546, 2006.

[7] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, 2007.

[8] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. Technical report, School of Computing Science, Simon Fraser University, 2009.

[9] Dirk Beyer, Damien Zufferey, and Rupak Majumdar. Csisat: Interpolation for la+euf. In *CAV*, pages 304–308, 2008.

[10] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The mathsat 4 smt solver. In *CAV*, pages 299–303, 2008.

[11] Juergen Christ and Jochen Hoenicke. Instantiation-based interpolation for quantified formulae. In Nikolaj Bjorner, Robert Nieuwenhuis, Helmut Veith, and Andrei Voronkov, editors, *Decision Procedures in Software, Hardware and Bioware*, number 10161 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[12] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Emerson and A. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin / Heidelberg, 2000. 10.1007/10722167_15.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 4963:337–340, 2008.

[14] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel*, pages 72–83, 1997.

[15] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. *SIGPLAN Not.*, 39(1):232–244, 2004.

[16] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, October 2009.

[17] Alexey Khoroshilov, Vadim Mutilin, Eugene Novikov, Pavel Shved, , and Alexander Strakh. Towards an open framework for C verification tools benchmarking. In *Proceedings of PSI*, pages 82–91, Akademgorodok, Novosibirsk, Russia, 2011.

[18] Kenneth L. McMillan. An interpolating theorem prover. In *TACAS*, pages 16–30, 2004.

[19] Kenneth L. McMillan. Interpolants from z3 proofs. Technical report, Microsoft Research, 2011.

[20] Leonardo Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *Proceedings of the 21st international conference on Automated Deduction: Automated Deduction*, CADE-21, pages 183–198, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] Vadim Mutilin and Mikhail Mandrykin. Instantiation-based interpolation for quantified formulae in csisat. In *Proceedings of the Institute for System Programming of the Russian Academy of Sciences (in Russian)*, 2012.

[22] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[23] Pavel Shved, Vadim Mutilin, and Mikhail Mandrykin. Static verfication "under the hood": Implementation details and improvements of BLAST. In *Proceedings of SYRCoSE*, volume 1, pages 54–60, 2011.