Predicate Abstraction of Programs With Non-linear Computation

Songtao Xia[1] Ben Di Vito[2] Cesar Munoz[3]

[1] NASA Postdoc at NASA Langley Research Center, Hampton, VA
[2] NASA Langley Research Center, Hampton, VA
[3] National Institute of Aerospace, Hampton, VA

Abstract. Verification of programs relies on reasoning about the computations they perform. In engineering programs, many of these computations are non-linear. Although predicate abstraction enables model checking of programs with large state spaces, the decision procedures that currently support predicate abstraction are not able to handle such non-linear computations. In this paper, we propose an approach to model checking a class of data-flow properties for engineering programs that contain non-linear products and transcendental functions. The novelty of our approach is the integration of interval constraint solving techniques into the automated predicate discovery/predicate abstraction process, which extends the expressive power of predicate abstraction-based model checking. Using this approach, we construct a prototype model checker for C programs called VISA (Verification of Industrial-Strength Applications). VISA is built on top of Berkeley's BLAST and University of Nantes' Realpaver. We successfully apply VISA to scientific computation libraries and avionics applications to verify the absence of certain runtime arithmetic errors.

1 Introduction

Software systems are notoriously bug-ridden. Formal techniques have become increasingly popular in verification, bug-hunting, and automatic test case generation. In this paper, we are interested in safety properties of a particular set of engineering programs from the avionics industry. These programs have two distinct features. First, their state space consists of hundreds, or thousands of, inputs to the system. Second, these programs may perform non-linear computations. More specifically, in avionics systems, input variables participate in the computation of control signals to be sent to actuators. The laws of electronics, dynamics, and geometry on which these computations are based constantly involve mathematical expressions that include non-linear products and transcendental functions.

An example of the domain of interest is given in Figure 1, taken from KB3D, an aircraft conflict detection and resolution program [18]. We are interested in the ability to prove that the variable `a` is non-zero at Line 5 (We ignore the issues caused by floating point arithmetic for now.). A brief argument for the property

is as follows: If `a` is zero, then both `vx` and `vy` have to be zero. Therefore, the first two terms in the assignment to `d` are zeros. The right hand side of the assignment is a subtraction from zero the sum of products of square numbers, which means `d` must be less than or equal to zero, which contradicts with the test in Line 4.

```
1: d = 2*sx*vx*sy*vy + sq(D)*(sq(vx)+sq(vy))
2:       - (sq(sx)*sq(vy) + sq(sy)*sq(vx));
3: a = sq(vx) + sq(vy);
4: if (d>0) {
5:       theta1 = 1/a;
6: }
```

Fig. 1. Code snippet from a conflict detection program

The challenge is to verify the kinds of properties automatically (with little or no human interaction) and accurately (with few or no false alarms). Among available techniques, some of which will be discussed in the related Section 7, we will focus on software model checking because it offers a high degree of automation. A major obstacle to software model checking is the large (infinite in most cases) state spaces. Predicate abstraction [7, 10, 15, 19, 21, 36, 40] has been successful in reducing the state explosion problem in model checking. This technique is particularly appealing when combined with counter-example driven predicate discovery techniques [4, 6, 12, 16, 25], because together they provide a (nearly) push-button process that, given a program and a property, will either verify the property or report a counter-example. The method is incomplete, but in practice, it has a high rate of success, especially when the cause of the bug (or the absence thereof) puts a virtual limit on the state space to be searched. Unfortunately, the decision procedures [9, 17, 20] used by predicate abstraction tools are not able to decide the satisfiability of formulas that contain non-linear computations.

This paper proposes a solution to predicate abstraction of programs with non-linear computations: instead of using a traditional cooperative decision procedure to answer the queries that occur during predicate abstraction and predicate discovery, we use constraint solvers based on interval analysis. Modern constraint solvers [24, 28, 34, 42] adopt a branch and prune strategy to search the solution space and apply interval computations [1,2,31] to reveal possible inconsistencies. Although incompleteness and slow convergence are intrinsic to these solvers, they are accurate, making them good candidates to be used in predicate abstraction.

Based on this approach, we extend Berkeley's BLAST [26] to construct a model checker for C programs with non-linear products and transcendental functions. Among possible applications, we present a prototype tool called VISA (Verification of Industrial-Strength Applications) that detects potential run-time violations such as division by zero and verifies the absence of these violations

under certain conditions. Another application of this approach is in automated test case generation [43]. We have applied VISA to software (often of tens of thousands of lines of code) taken from the scientific computing community and from the avionics industry, including legacy code from a Boeing 737 autopilot simulator and KB3D. Our model checker is able to verify (or find a counter-example of) properties that involves non-linear computation fully automatically. This automation is not accomplished by any other tool to the best of our knowledge.

The rest of the paper is organized as follows. Technical background is introduced in Section 2, where we focus on software model checking based on predicate abstraction and predicate discovery based on counter-example analysis. The challenge posed by programs with non-linear computation and our resorting to numerical approaches are discussed in Section 3. The overview of our solution is presented in Section 4. This section also discusses the soundness and incompleteness issues relevant to different types and configurations of the numerical constraint solvers. The implementation of VISA is described in Section 5. Section 6 reports experimental results. We discuss related work in Section 7.

2 Background

We present predicate abstraction and counter-example based predicate discovery in a general framework that is not tied to a particular programming language. Most of the material presented in this section is a review of well-known concepts.

2.1 Definitions

A (concrete) *state* of a program is a type preserving value assignment to program variables, which includes artificial ones such as *pc*. We denote by $E[s]$ the value of expression E evaluated at state s. We also write $s \models P$ if the predicate P holds at state s.

A (concrete and later, abstract) program can be organized as a control flow graph (CFG) (N, E, M, A), where N is a set of nodes that correspond to program locations, E is a set of edges $N \times N$, M is a set of *moves*, and A is a mapping of edges to moves. A move, concrete or abstract, is an abstraction of one semantic step in the program that changes (a model checker's) knowledge of the current (abstract) state. For a program without function calls, there are two kinds of moves: *assignments* and *assumptions*. An assignment move represents one or more assignment statements in a program. Assumptions model branch conditions of an if statement. One assumption is represented by a predicate showing the result of testing the if- condition; it labels the edge from the testing to the corresponding branch in the CFG. Given a state and a move, executing the move will result in the next state. We write $\hookrightarrow(m, s)$ for the state after the move m is executed in state s.

The CFG for the code above is illustrated in Figure 2 with edges labeled with moves. Concrete (directly corresponding to C statements) and corresponding abstract moves (explained in the next section) are listed alongside.

Move	Concrete	Abstract
E1:	d= 2*sx*vx*sy*vy	b1= 1
	+sq(D)*(sq(vx)+sq(vy))	
	-(sq(sx)*sq(vy)+ sq(sy)*sq(vx))	
E2:	a = sq(vx) + sq(vy)	b2 = 1
E3:	d<=0	not b3
E4:	d>0	(b1 ∧ b2)?
		b3 ∧ ¬ b4 : b3
E5:	theta = 1/a	nop
E6, E7	nop	nop

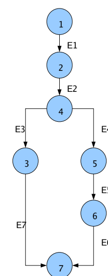

Fig. 2.

To reason about moves, weakest preconditions are used. We write $\mathcal{WP}(m, P)$ for the weakest precondition of P with respect to move m. We write $\mathcal{WP}(\bar{m}, P)$ as the weakest precondition with respect to the sequence of moves $\bar{m}$. A *counter-example* $\bar{m}$ is a sequence of moves. A counter-example $\bar{m}$ is feasible if $\mathcal{WP}(\bar{m}, P)$ is *satisfiable*. A formula is satisfiable if there is a value assignment to the variables so that the formula is true under a certain interpretation.

2.2 Predicate Abstraction

We give an operational definition of predicate abstraction partially following that of Ball's [4]. Predicate abstraction accepts as input a move m and a set Φ of predicates, and outputs a function (called *abstraction transition*) that maps one abstract state to another. An abstract state is represented as a bit vector. Every bit[4] in the vector represents the truth value (plus another value $*$ representing a non-deterministic choice) of a predicate in Φ. We denote by s_Φ the abstract state of s with respect to the set of predicates Φ. We overload the operator $\hookrightarrow (m, s_\Phi)$ to denote the next abstract state of s_Φ after move m is followed (by a model checker). We extend this operator to sets of states (concrete and abstract) in the natural way.

The computation of the abstraction transition relation is performed for each move. Informally, the effect of a move m over a predicate $P_i \in \Phi$ can be written

[4] Strictly speaking not a bit, but a variable ranging over values from a free lattice over $\{\text{true}, \text{false}\}$.

as an assignment [5] :

$$b_i = \mathcal{WP}'(m, P_i)$$

where we use b_i for the bit corresponding to P_i, $\mathcal{WP}'(m, P) = \mathcal{WP}(m, P)$ if m is an assignment, or $Q \Rightarrow P$ if m is assume(Q). Standard computation of predicate abstraction computes an approximation of $\mathcal{WP}(m, P)$ as $\mathcal{WP}_{\Phi}(m, P)$, which is implemented by calling a theorem prover to check the unsatisfiability of $Q \wedge \neg\mathcal{WP}(m, P)$.

The abstraction of our example with respect to four predicates (listed below) is shown on the right hand side of Figure 2.

$$\begin{aligned}
&b_1 : d = 2 * \texttt{sx} * \texttt{vx} * \texttt{sy} * \texttt{vy} + \texttt{sq(D)} * (\texttt{sq(vx)} + \texttt{sq(vy)}) - \\
&\qquad (\texttt{sq(sx} * \texttt{sq(vy)} + \texttt{sq(sy)} * \texttt{sq(vx))} \\
&b_2 : a = \texttt{sq(vx)} + \texttt{sq(vy)} \\
&b_3 : d > 0 \\
&b_4 : a = 0
\end{aligned}$$

Note that the branch of $d > 0$ is computed this way: $\mathcal{WP}'(d > 0, d > 0)$ is $d > 0 \Rightarrow d > 0$, so b_3 will always be true. $\mathcal{WP}(d > 0, \neg a = 0)$ is $d > 0 \Rightarrow \neg a = 0$, which is implied by:

$$\begin{aligned}
&a = \texttt{sq(vx)} + \texttt{sq(vy)} \wedge \\
&d = 2 * \texttt{sx} * \texttt{vx} * \texttt{sy} * \texttt{vy} + \texttt{sq(D)} * (\texttt{sq(vx)} + \texttt{sq(vy)}) - \\
&\qquad (\texttt{sq(sx} * \texttt{sq(vy)} + \texttt{sq(sy)} * \texttt{sq(vx))}
\end{aligned}$$

Note that, because non-linear computation is involved, the implication above cannot be proven by the cooperative decision procedures used in previous predicate abstraction methods. We will return to this issue in Section 3.

2.3 Predicate Discovery

Based on counter-example feasibility testing, counter-example driven predicate discovery allows the model checker to incrementally discover a suitable set of predicates, starting with an initial value of Φ. This procedure is known as *predicate refinement* and is in general incomplete (c.f., [4]). Let $\bar{m} = m_1, \ldots, m_n$ be a counter-example. Iteratively, we compute the weakest preconditions $P_1, \ldots, P_n$:

$$\begin{aligned}
P_1 &= \mathcal{WP}(m_n, \phi) \\
P_{i+1} &= \mathcal{WP}(m_{n-i+1}, P_i)
\end{aligned}$$

[5] Conventionally, an assumption is represented by a predicate. But as far as model checking is concerned, it is equivalent to this assignment form.

We check whether P_i is satisfiable. If, for some j, P_j is not satisfiable, we attempt to find new predicates from the path from m_j to m_n. One way to find new predicates is to collect all the predicates involved or use certain heuristics to select the new predicates. A better approach is to use Craig interpolation [25,29].

Again, in our example, we will need to check the satisfiability of non-linear formulas. The challenge and possible solutions are discussed in the next section.

3 Reasoning About Non-linear Computation

As revealed by the example in Section 2, reasoning about non-linear computation is an integral part of the abstraction and model checking mechanism. Unfortunately, the decision procedures used in counter-example driven predicate abstraction have trouble deciding the satisfiability of such formulas. They tend to work in a weaker theory of arithmetic. For example, in bug-hunting applications such as SLAM [7] and BLAST [26], the forms of the constraints are limited to propositional logic and quantifier free predicate logic with uninterpreted functions. When verifying hybrid systems, stronger decision procedures that accept linear equations and inequalities are used. For example, d/dt [3] uses the Lp_solve software package. Verification of non-linear programs in general is hard because non-linear arithmetic is not decidable over mixed (integer and real) variables and the satisfiability problem for formulas involving transcendental functions is not decidable even for reals.

3.1 Existing Tools

Existing decision procedures, such as ICS and CVC-lite [9, 20], also attempt to decide the satisfiability of non-linear products. Due to the nature of these cooperative decision procedures, such an attempt is made only during an early phase of an arithmetic sub-theory to rule out simple unsatisfiable cases. From our experience, the current versions of these tools cannot solve constraints that appear in our predicate abstraction.

Based on a variation of the simplex method [35] and computation of Gröbner basis [41], Tiwari's non-linear decision package [39] can solve many non-linear constraints very efficiently. Still the current version cannot handle unsatisfiable constraints that involve perfect squares.

None of these procedures mentioned above solves constraints that involve transcendental functions; in the best case, they can solve such constraints without interpreting these transcendental functions (for example, they can decide that formula $sin(x) = sin(x) + 1$ is not satisfiable).

3.2 Numeric Decision Procedures

Modern constraint solvers, pioneered by Numerica [24], adopt a branch-and-prune technique to either find a set of intervals that contain a solution or report that no solution is possible. In constraint solvers, the set of ranges where a

variable is defined is called a *box*. The band-and-prune algorithm takes as arguments a set of constraints, an initial set of boxes for each variable appearing in the constraints, and a precision. If during the search, all the boxes contain intervals that are smaller than the precision, then the search stops. The internal loop of the algorithm consists of two stages: prune and branch. Prune removes boxes that are not in the solution space and branch splits one box into two or more boxes. The prune stage enforces local consistency conditions by reducing intervals associated with the variables. Typically, the constraints are evaluated using interval arithmetic [31].

A group of local constraint satisfaction techniques with polynomial time worst case complexity are also used. They can be applied to non-linear, non-square, and heterogeneous systems. Furthermore, numerical methods are adopted to process either a sub-problem or a sub-class of problems. For example, a Newton method can be used for an equation of the form $f(x) = 0$, where function f is square and differentiable [1]. Moreover, systems of inequalities can be handled by a version of the Simplex method [27].

4 Approach

The goal of VISA is to detect potential runtime safety bugs for C programs. Like BLAST, it allows a user to specify the property that she wants to check. The property specification is instrumented with the source code (at the CFG level) to form a new CFG where a violation will be reported when a special error node is reached during model checking.

The model checker will take this instrumented CFG as input. The model checking is based on the procedures described in Section 2. First, predicate abstraction is performed using an enhanced theorem prover, which will behave just like a traditional theorem prover if the candidate theorem (constraints) does not contain non-linear computation, and will behave like a wrapper of a constraint solver when attempting to prove a non-linear candidate theorem. Then, model checking is performed over the abstract model. When the model checker concludes the (artificial) error label is not reachable, VISA will report that the code is safe. Otherwise, if a counter example is discovered by the model checker, the same enhanced theorem prover will be used to determine its feasibility. If the error path is feasible, VISA will report an error. Otherwise, VISA attempts to refine the error trace to find a new predicate to repeat the abstraction/model checking process. Figure 3 illustrates the architecture of VISA.

4.1 Instrumentation

In VISA, a source program is first instrumented with respect to a property specification. In the instrumented program, an error node (in the CFG) is reachable if and only if the specified error condition is true in the source program.

We have designed a specification language that is similar to that of BLAST's [11]. A *cut-point* is a program location (strictly speaking, not a program location,

but a node in CFG, see Section 5) where we may want to insert a check for a certain operation where we may insert a check; all cut-points that are pertaining to the operation are called (a not-entirely-misuse of term) *aspect*. A *pattern* is associated with a cut-point, which will match the actual expressions that participate in the operation of concern.

In VISA, the checking of division by zero is instrumented by first querying the patterns associated with division operations that we want to check. Such a pattern includes a divident and a divisor. We assert that this divisor must not be zero before division takes place. There is practically no restriction on the form of the formula being asserted. For example, a user may also choose to check whether the divisor's abstract value is less than a small positive constant.

Next, an instrumentor of VISA will scan the internal representation of the syntax tree (the CFG in BLAST), add an artificial test at an appropriate place per the specification, for example, before the division of interest. The assertion that specified by the user will be tested; if it is not true, an artificial error node is reached.

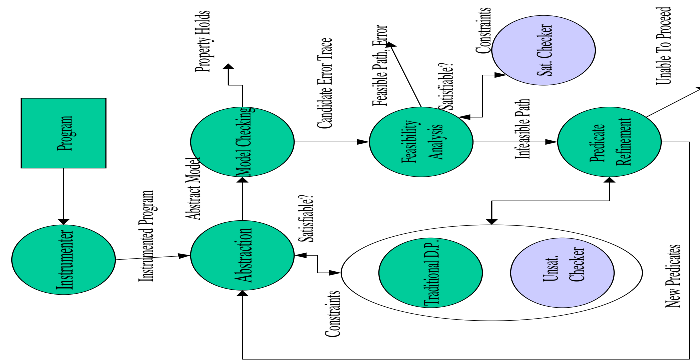

Fig. 3. Architecture of VISA

Once the code is instrumented, the model checker will check to see if the error node is reachable. The impact of using constraint solvers in such a model checker is discussed in the next subsection.

4.2 Using Constraint Solvers In Model Checking

In the approaches described above, the model checker uses the constraint solvers to process non-linear constraints at two different places (computing the abstract transition relation and testing/refining a candidate error trace). There are various configurations/types of numerical constraint solvers. Each solver behaves differently with regard to soundness, completeness or performance and is suitable only for certain applications.

Satisfiability vs. Unsatisfiability Ideally, a constraint solver may return three possible answers: Satisfiable, Unsatisfiable, and Don't Know. In practice, a tool may return two answers (Satisfiable/Don't Know, or Unsatisfiable/Don't Know). For example, Realpaver, which is used in our prototype implementation, will either declare that a set of constraints is not satisfiable or give a set of boxes that might contain a solution. The latter should be considered as Don't Know. We will call a constraint solver that returns Satisfiable/Don't Know as a *satisfiability checker* while one that returns Unsatisfiable/Don't Know as an *unsatisfiability checker*.

- When testing whether an error path is feasible, an unsatisfiability checker may return a Don't know. Then we cannot detect an unfeasible path and can only raise a false alarm. Conversely, a satisfiability checker may return Don't know on a feasible path, which further contributes to the incompleteness of the system.
- When computing predicate abstraction, we should always use an unsatisfiability checker. As long as the Unsatisfiable answer is trusted, the soundness of predicate abstraction is preserved. Of course, Don't know answers further contributes to the imprecision that already exists in predicate abstraction.

It is also worth mentioning that the precision of a tool is adjustable. The more precise the tool is, the slower it is.

Floating Point vs. Real Numbers It is also important to know exactly what satisfiability problem a particular solver aims to solve. In particular, whether the problem domain is real numbers or floating point numbers has a profound impact on the soundness and completeness of the system.

In theory, interval-based techniques can solve satisfiability problems for both floating point numbers and real numbers. If the satisfiability is interpreted as the existence of a floating point solution, because the domain is finite, the procedure always terminates; if the satisfiability is interpreted on reals, then the procedure may not terminate. But the unsatisfiability check can be highly accurate [22,34]. So far, the majority of the tools have been focused on solving real constraints. As a result, our experience has focused on these solvers.

When using a solver for real numbers, the result of both verification and bug-hunting must be treated with care. If the model checker signals that there is no error, then if we do not consider rounding errors, this answer is sound provided

that the over-approximation condition is (as expected) satisfied. On the other hand, if the model checker finds a violation based on the fact that there is a real solution to a constraint, then this real solution may not correspond to a floating point number solution, in which case we have a false alarm.

It is hard for constraint solvers to maintain a semantics that exactly matches that of the floating point arithmetics of the machine; in reality, they rarely do. When an unsatisfiability checker decides that a particular set of boxes does not contain a solution, due to rounding error, there could still be a solution that causes the constraints to be satisfied. Here we must not confuse the interval arithmetic used in determining the unsatisfiability with the interval arithmetic used in controlling rounding error. In practice, a constraint solver extensively uses the former but seldom uses the latter.

The implication of this problem with rounding error is that we cannot claim full verification without the assumption of absence of rounding errors.

4.3 Example of VISA Approach

In Figure 2, suppose we are interested in Line 5, where a division takes place. To decide whether a could be zero, a counter-example driven approach will start the initial value of Φ to be $\{a = 0.0\}$. Line 1 does not affect this predicate. Line 3 assigns the sum of two square numbers to a. The precondition of $a = 0.0$, for example, will be:

$$0 = \texttt{sq(vx)} + \texttt{sq(vy)}$$

The predicate abstractor will attempt to decide whether combinations of predicates imply this precondition. This decision is made by calling an unsatisfiability checker. If the combination is $a = 0.0$, then we decide whether constraint:

$$\begin{array}{c}(a = 0.0)\ \wedge \\ \neg(0 = \texttt{sq(vx)} + \texttt{sq(vy)})\end{array}$$

is satisfiable to see if $a = 0.0$ should be included in the approximation. The constraint is acquired as a conjunction of $a = 0.0$ and the negation of the precondition (the conclusion of the implication). This constraint is satisfiable, which means the implication does not hold. Repeating this for $\neg(a = 0.0)$, Line 3 will be translated into $b = *$, where b is the Boolean variable corresponding to $a = 0.0$.

Suppose that we have a test that checks whether a is 0 between Line 4 and Line 5. Because a is $*$, there will be a path in which $a = 0.0$ could be true. This way we have a counter-example. By analyzing this trace, we will find out that the following constraint, which is computed using weakest preconditions, is not satisfiable (note that the counter-example contains a test of a at the end, which is not reflected in the figure):

$$
\begin{aligned}
&\mathtt{a} = 0.0 \wedge \\
&0.0 = \mathtt{sq(vx)} + \mathtt{sq(vy)} \wedge \\
&0.0 < \mathtt{d} \wedge \\
&0.0 < 2 * \mathtt{sx} * \mathtt{vx} * \mathtt{sy} * \mathtt{vy} + \mathtt{sq(D)} * (\mathtt{sq(vx)} + \mathtt{sq(vy)}) - \\
&\qquad (\mathtt{sq}(\mathtt{sx} * \mathtt{sq(vy)} + \mathtt{sq(sy)} * \mathtt{sq}(Idvx))
\end{aligned}
$$

A satisfiability constraint solver will decide that the constraint is not satisfiable. Thus the error trace is not feasible. Then related predicates (the four predicates described earlier) are added into Φ. This time, when the model checker reaches Line 4, in addition to $d > 0$, the predicate abstraction will also notice that $\neg(0.0 = a)$ must be true because the constraint below is not satisfiable. This constraint is the conjunction of the negation of formula above, that is, $a = 0.0$, the combination of predicates being tested and $d > 0$, which is introduced by computation of the precondition (details on how the constraints is computed are described earlier in Section 2.2).

$$
\begin{aligned}
d = 2 * \mathtt{sx} * \mathtt{vx} * \mathtt{sy} * \mathtt{vy} + \mathtt{sq(D)} * (\mathtt{sq(vx)} + \mathtt{sq(vy)}) - \\
(\mathtt{sq}(\mathtt{sx} * \mathtt{sq(vy)} + \mathtt{sq(sy)} * \mathtt{sq}(Idvx)) \wedge \\
a = \mathtt{sq(vx)} + \mathtt{sq(vy)} \wedge d > 0 \wedge a = 0.0
\end{aligned}
$$

Then, between Line 4 and Line 5 the predicate abstractor will recognize that $a = 0.0$ must be false. Therefore the program will not cause division by zero.

5 Implementation of VISA

We implement our model checker based on two existing systems, BLAST from Berkeley [26] and Realpaver from University of Nantes [28]. BLAST provides a reachability test framework for a C program; Realpaver can be used to determine the unsatisfiability of a set of non-linear constraints. We extend Realpaver to a decision procedure of the Nelson and Oppen flavor. We then plug the new decision procedure into BLAST, replacing the decision procedures used there.

5.1 Extending BLAST

With C programs, many problems must be addressed before or during model checking. Function calls, pointers, various data types and a programmer's tendency to explore their flexibility are just some of them. Using CIL (C Intermediate Language) [30], BLAST handles full C syntax and represents different C syntax structure in a uniform and less ambiguous way. BLAST also provides context free analysis and alias analysis in a best-effort manner. Therefore, we do not focus on the issues mentioned above and rely on BLAST to cope with these for us, albeit pointers remain a major source of problems.

One extension (besides the interface to theorem provers, which will be covered in the next subsection) to BLAST is an instrumentation API that allows a user to specify the properties that they want to check. The specification language of VISA can be viewed as a front end using this API. A user can query about a cut-point and use the matching pattern to define a check using both the specfication language (as we have shown earlier) or equivalently, the API. Besides division, VISA supports the following aspects: addition, function call/return, array access, and assignment.

A cut-point in VISA is a finer-grained event that is of interest to a property comparing that in BLAST. For example, in BLAST, all function calls, are possible cut-points to verify the correct use of APIs. But this specification language cannot be used in VISA because the cut-points do not include arithmetic (and other) operations.

Other parts of BLAST that are particularly useful (some of which require re-programming) in VISA include:

- Lazy abstraction. The cost of predicate abstraction is exponential in the number of predicates. One of the important features of BLAST is lazy abstraction, where a predicate is included in the computation only in a part of the model checking tree where it is necessary. This is especially useful when clusters of predicates are locally relevant to the property only in certain parts of the search path. Such a pattern is found frequently in the programs of interest.
- Soundness issue. BLAST provides several options to compute the predicate abstraction. For speed considerations, the most commonly used ones are not conservative with respecct to aliases because alias-safe predicate abstraction is too expensive to be practical and alias analysis in general is not precise enough. Although the abstraction may be unsound and not suitable for strict verification, it is still good for bug-hunting.
- Trace Slicing. This is arguably one of the most useful features of BLAST. The idea is to slice the candidate error path to a portion of it that maintains same feasibility characteristics. Because the constraints generated in this stage are normal large and become a bottleneck for the constraint solver. Slicing the path will often generate a surprisingly small constraint. When combined with Craig interpolation, this feature is useful for dealing with counter-based loops (such as a for loop), which are sometimes used in initialization parts of the code and a major cause of unfeasible error trace.

5.2 Realpaver

We use Realpaver [28] as the numerical constraint solver. Based on interval computations, Realpaver solves non-linear formulas over the real numbers. The inputs to Realpaver are

- a finite list of real variables $\mathcal{V} = \{x_1, \ldots, x_n\}$,
- a list of constraints (that can contain nonlinear products and transcendental functions), and

– an initial set of of interval domains $\mathbf{x}_1, \ldots \mathbf{x}_n$ (called a *box*) for the variables in $\mathcal{V}$.

Under the assumption $x_i \in \mathbf{x}_i$, for $1 \leq i \leq m$, Realpaver returns either a *no solution in the initial box* message or a list of boxes, included in the initial box, that contain solutions to the conjunction of the constraints. Realpaver also returns a flag that indicates whether the resolution process was reliable or not. A non-reliable output means that some solutions may be lost during the process.

Realpaver is claimed to satisfy the following property [28].

Proposition 1 (Reliability). *Realpaver computes a union of boxes that contains all the solutions of the original constraint satisfaction problem. Therefore, if no box is computed by Realpaver, the constraint satisfaction problem has no solutions.*

This property means that Realpaver can be used as the basis for an unsatisfiability decision procedure.

We use an ad hoc method to integrate Realpaver into a cooperative decision procedure (cvc-lite [9]). Specifically, the core engine of a cooperative decision procedure uses variable abstraction to divide input formulas (of multiple theories) into formulas of different sub-theories. That is, these formulas do not contain sub-formulas that involve functions or predicates of a different theory. For each sub-theory, a combination of so-called solver and canonizer will find equalities and dis-equalities; this information is propagated to the core engine to find either a solution or inconsistency. Realpaver cannot be a solver as needed by such cooperative decision procedures because it cannot discover equality or dis-equality in an easy way. However, because it can discover inconsistencies, it is possible to put Realpaver into an arithmetic sub-theory before the solvers of this sub-theory is called and signal the core engine only when inconsistency is found. Modern cooperative decision procedures already use different heuristics to find simple inconsistency early at that stage (for speed considerations) [8]. In this sense, Realpaver can be considered as another heuristic.

6 Experience

In a preliminary case study, we apply VISA to a set of public domain scientific computation libraries. We choose these programs because 1) as scientific computation applications, they resemble the engineering programs in aviation industry, the domain of our interest; 2) these programs are actively maintained public domain program and are considered programs with reasonable quality; 3) they might not be as good in quality as the programs in the aviation industry, which makes them a good target to improve. We primarily look for division by zero violations. Table 4 below lists a few representative programs, their sizes, the number of divisions, model checking time, and the number of runs when the model checker fail to terminate (failure runs column in the table). The size of the program is measured by the numbers of lines of syntactically reachable

functions with comments removed. The model checking time is the mean time in seconds for all terminating runs (we configure VISA to run once per division). The data reported here is on executing VISA on a commodity laptop (Pentium M 1.73GHz, 512Mb).

Program	Size	No. of Div.	MC Time	Failure Runs
anneal.c	12602	27	210	3
conjdir.c	24134	20	288	1
cube.c	1834	10	65	0
spmat.c	18517	11	60	0

Fig. 4. Representative Runs of VISA

We found division-by-zero traces for three of these programs (anneal.c, conjdir.c and cube.c). Human inspection of the error trace proves that these are all not false alarms. When there are no alarms, through reasoning about the source code (and the model checking trace produced by VISA) manually, we are able to double-justify the absence of division-by-zero.

Also, we apply VISA to KB3D. The correctness of this program, including the safety with regard to division-by-zero, has been previously verified using the theorem prover PVS [32]; thus the program is considered of high quality. KB3D is a small program of a few thousand lines and contains predominantly geometric computations. KB3D contains a number of good examples that demonstrate the capability of VISA. We are able to verify that this program is free of division-by-zero. The computation time is usually within a minute. We conjecture that other tools either are not be able to handle KB3D due to large number of non-linear computations or report false alarms.

VISA and its test suites are available on line at http://www.nianet.org/m̃unoz/VISA.

7 Related Work and Conclusion

7.1 Program Analysis

In computer science folklore, data flow analysis has been treated as model checking over abstract domains [37]. Yet predicate abstraction can be viewed as a systematic way of designing abstract interpretation, and the counter-example driven approach is strongly connected with the widening operator. Abstraction based on interval analysis has been studied by Cousot's group to reduce runtime errors in C programs. Their tool, ASTREE [14], is based on such abstraction domains as octagon, ellipsoid and decision trees. ASTREE has been successfully applied to large embedded, command and control, safety critical real-time software. Differences between VISA and ASTREE are: First, VISA essentially provides an abstraction mechnism for a non-linear domain, which is a substantial (and practically useful) gain of expressive power; second, VISA does not handle

the rounding errors, while ASTREE does; third, VISA is fully automatic (for all programs) while ASTREE needs to be trained to work on a family of programs; and fourth, VISA inherits unsound factors (such as pointers) and incompleteness from BLAST, which is not an issue with ASTREE because of its selected application domain.

Combining different aspects from VISA and ASTREE is a promising research direction. ASTREE researchers have pointed out that certain abstractions cannot be achieved using a counter-example based approaches; the problem that we had with counters is another example where other forms of abstract interpretation (different from predicate abstraction) are more efficient.

7.2 Decision Procedures

The decidability issue of real arithmetic dates back to the 1930's. Tarski [38] shows the first order theory of real numbers with addition and multiplication is decidable through quantifier elimination. Collins shows that quantifier elimination can be done through Cylindrical Algebraic Decomposition [13]. Adding different functions to the theory is different case by case. For example, adding periodic functions such as sin will cause the theory to be undecidable, while adding exp is decidable conditionally (if Schanuel's conjecture holds). Numerical decision procedures (for so-called stable formulas) are studied by Ratschan [33].

Cooperative decision procedures are mostly based on proposals by Shostak and by Nelson and Oppen. Various systems are used in practice, such as ICS [20], CVC-Lite [9], Simplify [17], Euclid, etc. Microsoft's Zapato [5] is designed specifically to solve formulas for predicate abstraction and is used in Microsoft's SLAM. Zapato uses Nelson and Oppen's method to combine a theory of uninterpreted functions with a solver for conjoined (linear) integer constraints based on Harvey and Stuckey's method [23], which is complete and linear in time. Zapato also takes advantage of fast propositional SAT solvers to first try an abstracted version of the original constraints.

8 Conclusion

This paper extends the current practice of automated software model checking to checking data-flow properties for real, engineering programs that contain non-linear products and transcendental functions. We propose the adoption of interval constraint solvers as the (un)satisfiability checkers used in predicate abstraction and predicate discovery. The soundness and completeness issues are discussed under both theoretical and practical settings. Factors that affect these issues are identified. Based on our proposed approach, a practical system is built for bug-hunting/verification. This prototype shows the potential applications of our model checking framework. The effectiveness of the prototype system is demonstrated on real programs from the avionics industry.

We feel that the framework of our method and the initial success of our prototypes constitute a reasonable contribution to state-of-the art in predicate

abstraction research. Our preliminary case studies demonstrated the expressive power of this approach in verifying arithmetic safety. The prototypes that we implemented are valuable complements to the existing tools in the respective communities. We expect the approach to be integrated with other approaches as part of a collective method to prove or disprove run-time errors in an accurate and static way.

References

1. A. Neumaier. *Interval Methods for System of Equations.* Cambridge University Press, 1990.
2. G. Alefeld and J. Herzberger. *Introduction to Interval Computations.* Academic Press, 1983.
3. E. Asarin, T. Dang, and O. Maler. The d/dt tool for verification of hybrid systems. In *CAV*, pages 365–370, 2002.
4. T. Ball. Formalizing counter-example driven predicate refinement with weakest preconditions. Technical Report MSR-TR-2004-134, Microsoft Research, 2004.
5. T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *CAV*, pages 457–461, 2004.
6. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic Predicate Abstraction of C Programs. In *Proceedings of Programming Languages Design and Implementation (PLDI) 2001*, pages 268–283. ACM, 2001.
7. T. Ball and S. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN2001, Lecture Notes in Computer Science 2057*, pages 103–122. Springer-Verlag, May 2001.
8. C. Barret and C. Tinelli. Theory and practice of decision procedures for combinations of theories. Slides of Talk Given at CAV 2005.
9. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *CAV*, Lecture Notes in Computer Science. Springer, 2004.
10. S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *Proceedings of Conference on Computer Aided Verification (CAV) 98, Lecture Notes in Computer Science 1427*, pages 319–331, June 1998.
11. D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *Proceedings of the 11th International Static Analysis Symposium (SAS 2004), LNCS 3148*, pages 2–18. Springer-Verlag, 2004.
12. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of Conference on Computer Aided Verification (CAV) 00*. Springer-Verlag, 2000.
13. G. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proceedings of the Second GI Conference on Automata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183. Springer-Verlag, 1975.
14. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREE analyser. In *Proceedings of The European Symposium on Programming*, pages 21–30, 2005.

15. S. Das, D. Dill, and S. J. Park. Experience with Predicate Abstraction. In *Proceedings of Conference on Computer Aided Verification(CAV) 99, Lecture Notes in Computer Science 1633*, pages 160–171, Trento, Italy, July 1999.
16. S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Proceedings of Conference on Formal Methods in Computer-Aided Design*, Portland, Oregon, November 2002.
17. D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking, 2003.
18. G. Dowek, A. Geser, and C. Muñoz. Tactical conflict detection and resolution in a 3-D airspace. In *Proceedings of the 4th USA/Europe Air Traffic Management R&DSeminar, ATM 2001*, Santa Fe, New Mexico, 2001. A long version appears as report NASA/CR-2001-210853 ICASE Report No. 2001-7.
19. M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, R. Visser, and H. Zheng. Tool-supported Program Abstraction for Finite-state Verification.
20. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonizer and Solver. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (Paris, France)*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer-Verlag, July 2001.
21. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of Conference on Computer Aided Verification (CAV) 97, Lecture Notes in Computer Science 1254*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.
22. L. Granvilliers. On the combination of interval constraint solvers. *Reliable Computing*, 7(6):467–483, 2001.
23. W. Harvey and P. J. Stuckey. Constraint representation for propagation. *Lecture Notes in Computer Science*, 1520:235–245, 1998.
24. P. V. Hentenryck, L. Michel, and Y. Deville. *Numerica, A Modeling Language for Global Optimization*. The MIT Press, 1997.
25. T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstraction from Proofs. In *Proceedings of ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages (POPL)*, pages 232–244, 2004.
26. T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-Safety Proofs for Systems Code. In *Proceedings of Conference on Computer-Aided Verification (CAV)*, pages 526–538, 2002.
27. K. Yamamura, H. Kawata and A. Tokue. Interval analysis using linear programming. *Proceedings of BIT 38*, pages 188–201, 1998.
28. L. Granvilliers and F. Benhamou. Realpaver: An interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*. Accepted for publication.
29. K. L. McMillan. Craig interpolation and reachability analysis. In *SAS*, page 336, 2003.
30. S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer. CIL: Intermediate Languages and Tools for C Program Analysis and Transformation. In *Proceedings of Conference on Compiler Construction (CC'02)*, March 2002.
31. R. Moore. *Interval Analysis*. Prentice-Hall, 1966.
32. S. Owre, J. Rushby, and N. Shankar. Pvs: A prototype verification system, 1992.
33. S. Ratschan. Slides, available at `http://www.mpi-sb.mpg.de/\~ratschan/decproc1.pdf`.
34. S. Ratschan. Continuous first-order constraint satisfaction. In *Proceedings of Artificial Intelligence and Symbolic Computation*, LNCS. Springer, 2002.

35. W. Rudin. *Principles of Mathematical Analysis (Third Edition)*. McGraw-Hill, 1976. Chapter 10.
36. H. Saidi and N. Shankar. Abstract and Model-check While You Prove. In *Proceedings of Conference on Computer Aided Verification (CAV) 99, Lecture Notes in Computer Science 1633*, pages 443–454. Springer-Verlag, July 1999.
37. D. Schmidt. Data Flow Analysis is Model Checking of Abstract Interpretation. In *Proceedings of SIGPLAN Symposium on Principles of Programming Languages (POPL) 98*, 1998.
38. A. Tarski. *Logic, Semantics, Metamathematics, papers from 1923 to 1938*. Hackett Publishing Company, 1983. English Version, original in Polish.
39. A. Tiwari. An algebraic approach for the unsatisfiability of nonlinear constraints. In L. Ong, editor, *Computer Science Logic, 14th Annual Conf., CSL 2005*, volume 3634 of *LNCS*, pages 248–262. Springer, Aug. 2005.
40. W. Visser, S. Park, and J. Penix. Applying Predicate Abstraction to Model Check Object-oriented Programs. In *Proceedings of the 33rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*.
41. W. Adams and P. Loustaunau. *An Introduction to Gröbner Bases*. American Mathematical Society, 1994.
42. M. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming, 1997.
43. S. Xia, B. D. Vito, and C. Muñoz. Automated test case generations for non-linear engineering programs. In *Proceedings of the Nineenth International Conference on Automated Software Engineering*, 2005.