A Constraint-Based Approach to Verification of Programs with Floating-Point Numbers

Carlos Acosta, Martine Ceberio, and Christian Servin
Computer Science Department
University of Texas at El Paso
El Paso, Texas 79968-0518
Email: ceacosta@gmail.com, mceberio@utep.edu, christians@miners.utep.edu
Telephone: (915) 747–6950
Fax: (915) 747–5030
SERP'08

***Abstract*—Software plays an important role in our daily lives. It can even be of extreme importance, e.g., software included in our cars such as the anti-lock brake systems, software applications controlling airport traffic, applications used in hospitals to deliver radioactive treatment to patients, programs used in nuclear stations. Ideally, we want programs to always work right and as specified. That is, we want reliable software. One of the things we can do to achieve reliable software is to verify it and validate it. Validation and Verification (V&V) are two terms that are similar and both terms have been used to refer to all the activities we perform to check that software does what it is supposed to do.**

Constraint Programming has been successfully used in solving scheduling problems and circuit design, among others. Its use in verification of software is still being researched and it is being applied to automatically generating test cases and to showing the conformity of software to its specification.

In this work, we propose a different approach, based on constraints, to translate code constructs where existing approaches made use of guarded constraints. We discuss the challenges and advantages of our approach, and we describe a process for solving constraints of the form $\neg A \wedge B$, which arise in our approach.

Keywords. Software verification, constraint solving, floating-point programs, inner/outer approximations.

Contact author: Martine Ceberio

I. Introduction

Software plays an important role in our daily lives. There is software in our cell phones, in our workplaces and in our homes just to mention a few examples. There is also software that is even more important; for example, software included in our cars such as the anti-lock brake systems, software applications controlling airport traffic and software in the airplane itself, applications used in hospitals to deliver radioactive treatment to patients, programs used in nuclear stations. These kinds of applications are critical: human-lives depend on their functionalities. Ideally, we want programs to always work right and as specified [12]. Programmers and users want specially the latter kind of software to work right. That is, we want reliable software. One of the things we can do to achieve reliable software is to verify it and validate it. Validation and Verification (V&V) are two terms that are similar and both terms have been used to refer to all the activities we perform to check that software does what it is supposed to do. We adopt the following meaning for validation and verification. Validation refers to checking that a system satisfies its specification (usually checking that the design specification satisfies the users requirements) whereas verification refers to proving that a system satisfies its specifications (usually proving that the code satisfies the design specifications) [12], [13].

Constraint Programming has been successfully used in solving scheduling problems and circuit design, among others. Its use in verification of software is still being researched and it is being applied to automatically generating test cases and to showing the conformity of software to its specification. In this work, we survey constrained-based verification techniques [3], [4], [11] and we propose a different approach, translating code construct where existing approaches made use of guarded constraints. We describe the challenges and advantages of our approach. In particular, we describe a process to solve constraints of the form $\neg A \vee B$, which arise in our approach.

II. Common verification techniques

In this section, we present common verification techniques as listed by Balci in [1], [2], and shown on Figure 1. Balci classifies these techniques as follows:

- **Informal verification techniques**
  These techniques usually involve human participation and rely mainly on human reasoning. Some of the techniques that fit in this category are the following: audits, inspections, walkthroughs, and reviews.
- **Static verification**
  These techniques do not require machine execution of the model. Some of the techniques that fit in this category are the following: data flow analysis, syntax analysis, graph-based analysis, and structural analysis.
- **Dynamic verification**
  These techniques require model execution. Some of the techniques that fit in this category are the following: testing, debugging, and execution tracing.

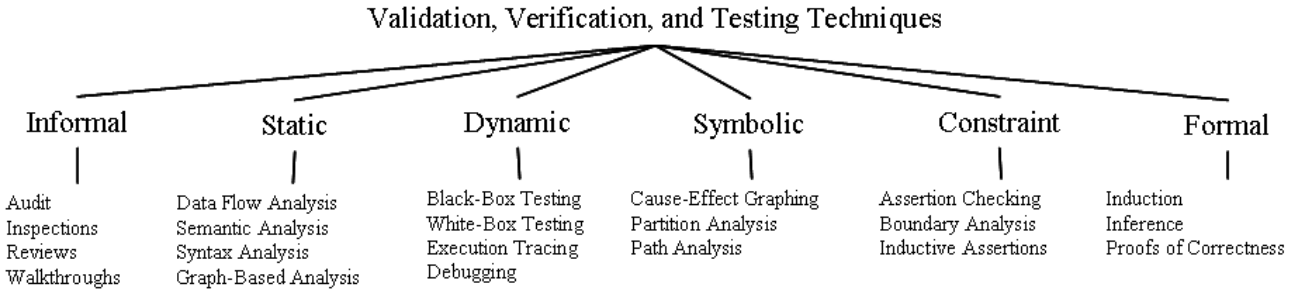

Fig. 1. Common V & V techniques

- **Symbolic verification**
  On these techniques symbolic inputs are given to a model and the output consists of expressions that result from the transformation of the symbolic inputs after going through execution paths in the model. Some of the techniques that fit in this category are the following: path analysis, symbolic execution, cause-effect graphing, and partition analysis.
- **Constraint verification**
  Constraint verification techniques use assertion checking, boundary analysis, and inductive assertions. Some of the techniques that fit in this category are the following: assertion checking, boundary analysis, and inductive assertions.
- **Formal verification**
  These techniques are based on formal mathematical proof of correctness. Some of the techniques that fit in this category are the following: lambda calculus, logical deduction, and proof of correctness.

III. Related work on constraint-based verification techniques

A. Automatic test case generation

In this section, we briefly describe the work of Gotlieb et al. [9]. In their approach, the authors mainly discuss their work for computations with integers.

The approach taken by Gotlieb et al. [9] consists of automatically generating test data that will execute a selected point in the code. In this approach, they transform the code into Static Single Assignment (SSA) form and analyze control-dependencies. They build a constraint system with this information and then they solve this system. When the constraint system is solved, test data is generated such that the selected point executes (if there is a feasible path that leads to the selected point).

The main steps of Gotliebs et al. can be outlined as follows:

1) Translate the program into a constraint system from its SSA form and control-dependencies.
2) The result of this step is a set of constraints (Kset). This set consists of the constraints generated for the program and the constraints that are generated for the selected point.
3) Solve Kset to generate test data for the selected point if there is at least one feasible path to the selected point.

B. Conformity of specifications and code

In this section, we briefly describe the work of Rueher et al. [3], [5] Their approach handles only operations with integers, i.e., they work on discrete domains. In this section, we first present an overview of the work in [5], which is the base of our work. Then, we describe the steps followed by Rueher et al.

The approach taken by Collavizza and Rueher consists in transforming the program and its specification into a constraint system. In this work, a program is verified if the union of the constraints derived from the program and the negation of the constraints derived from its specification is inconsistent (i.e., it does not have a solution). Consider that we have specification B and its implementation A: we would then try to solve $A \wedge \neg B$. In this sense, Rueher's and Collavizza's approach is similar to the process of resolution in logic. That is, we want to show that the implementation models the specification. That is,

$$A \models B \quad \text{which is equivalent to} \quad A \wedge \neg B \models \perp$$

The main steps of Collavizza's and Rueher's approach can be outlined as follows:

1) Translate the program into a constraint system. (A)
2) Translate the negation of the specifications into a constraint system. ($\neg B$)
3) Consider the conjunction of these two constraint systems as a CSP (possibly involving guarded constraints): ($A \wedge \neg B$)
   a) If a solution is found, it means that the program does not meet its specification and the solutions to the CSP are the test cases that would fail to meet the specifications.
   b) If a solution is not found, it means that the program meets its specification.

IV. Our contribution

Our approach is based on the work of Rueher and Collavizza [5]. In this work, the CSPs generated from the code and the specifications may contain guarded constraints. Collavizza

and Rueher point out that standard CSP solvers may not be able to prune the system, and only after a costly enumeration process, the CSP solver is able to detect an inconsistency on the CSP. To deal with this disadvantage of standard CSP solvers, they propose using a SAT solver first. They introduce a Boolean variable for modeling conditional statements such as $i < j$. Once the transformation is done, standard CSP solvers are able to detect the inconsistency.

In our work, we want to be able to handle programs involving floating points, i.e., representing real values. Since SAT solvers are not efficient for such problems, we want to consider an alternative approach so that there is no need for a SAT solver and it can be extended to handle domains of real numbers. The alternative we consider is that in the case of guarded constraints, we translate them by using the equivalence of logical implication with a disjunction. In the following subsection, we describe the main steps of our proposed approach.

A. Algorithm of our proposed approach

Our proposed approach can be outlined as follows:

1) Translate if-then-else statements as specified by Collavizza and Rueher [5]. This step generates guarded constraints of the form $A \rightarrow B$.
2) Transform guarded constraints of the form $A \rightarrow B$ into their equivalent $\neg A \vee B$. After Step 1, the CSP is in conjunctive normal form (CNF).
3) Transform the CSP in the form of a CNF into a disjunctive normal form (DNF), which is a disjunction of CSPs.
4) Solve the CSPs and consider the final solution to be:

$$\bigcup_i \text{Solution}(C_i)$$

In the following subsection, we present an example describing the steps outlined above.

B. Example: Our proposed approach

Consider the example shown in Table 1 (also summarized in Figure 2. After translating the code and the negation of the specification we get the constraints shown in Table 2.

```
// @ensures \results ≥ 0
public int absolute(int i, int j) {
    if (i < j)
        return (j − i)
    else return (i − j)
}
```

TABLE I
EXAMPLE OF AN IF-THEN-ELSE STATEMENT

If the code is correct with respect to its specifications, then we are aiming at the following CSP:

$$\{(c_1 \rightarrow c_2) \wedge (c_3 \rightarrow c_4) \wedge c_5,\ D_i = D_j = D_r = \{0, ..., 65635\}\} \quad (1)$$

which is expected not to have solutions.

c_1 :	$i < j$
c_2 :	$r = j - i$
c_3 :	$i \geq j$
c_4 :	$r = i - j$
c_5 :	$r < 0$

TABLE II
CONSTRAINTS FROM TABLE 1

However, we want to translate the guarded constraints into their equivalent (plain) constraints. After the transformation, we obtain the following new (equivalent) constraint system:

$$\{(\neg c_1 \vee c_2) \wedge (\neg c_3 \vee c_4) \wedge c_5, D_i = D_j = D_r = \{0, ..., 65635\}\} \quad (2)$$

Unfortunately, disjunctions of CSPs are difficult to deal with, thus we need further transformation.

Currently, we have a CSP in CNF, which involves disjunctions of constraints. We want to translate the CSP into a DNF because dealing with disjunction of CSPs is easier than dealing with disjunctions of constraints within a CSP. After this transformation on the CSP shown in Equation 1, we obtain the disjunction of the following CSPs:

$$\begin{aligned}
CSP_1: &\ \{(\neg c_1 \wedge \neg c_3 \wedge c_5), \\
&\ D_i = D_j = D_r = \{0, ..., 65635\}\} \\
CSP_2: &\ \{(\neg c1 \wedge c4 \wedge c5), \\
&\ D_i = D_j = D_r = \{0, ..., 65635\}\} \\
CSP_3: &\ \{(c_2 \wedge \neg c_3 \wedge c5), \\
&\ D_i = D_j = D_r = \{0, ..., 65635\}\} \\
CSP_4: &\ \{(c2 \wedge c4 \wedge c5), \\
&\ D_i = D_j = D_r = \{0, ..., 65635\}\}
\end{aligned}$$

Now, we would have to solve these 4 CSPs and what we are aiming at is:

$$\bigcup_i \text{solution}(CSP_i) = \emptyset$$

However, in order to limit the computational complexity of our approach, we apply rules to reduce the number of CSPs to be solved. We now present these elimination rules.

Challenges of our proposed approach In our approach we face the following two challenges:

- Solving a union of CSPs.
- Solving conjunction of CSPs of the type $\neg A \wedge B$.

In the following, we describe how to address both challenges.

C. Elimination rules

Solving a union of CSPs means solving each of them one by one. Therefore our objective is to control the number of CSPs to be solved as much as possible. In our approach for each CSP involving guarded constraints, we generate 4 CSPs. However, we can eliminate two CSPs by applying the following rules:

- **Rule 1:** Eliminate CSPs containing $C \wedge \neg C$. We apply this rule such CSPs have no solution. Moreover, if we were dealing with real domains, considering solving this

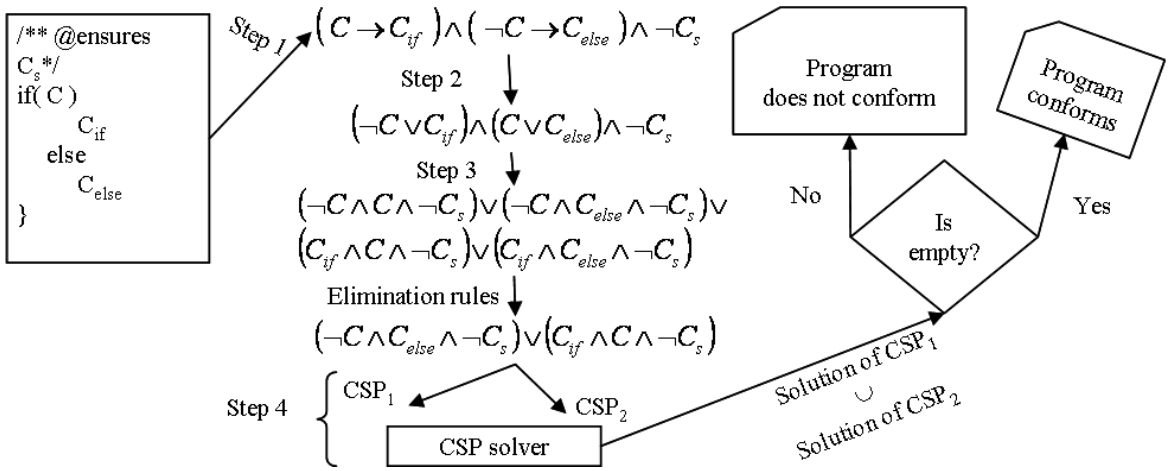

Fig. 2. Solution set of $\neg A \wedge B$

kind of CSPs could become a source of false positives and we want to reduce the number of false positives.

- **Rule 2:** Eliminate CSPs containing $C_{if} \wedge C_{else_1}$. We can apply this rule because of the semantics of the if-then-else statement.

By doing so, we reduce the explosion of CSPs to be solved.

D. Solving constraints of the form $A \wedge \neg B$

Consider that we have two constraints, A and B, defining a relation on x and y. A and B are represented as shown in Figure 3.

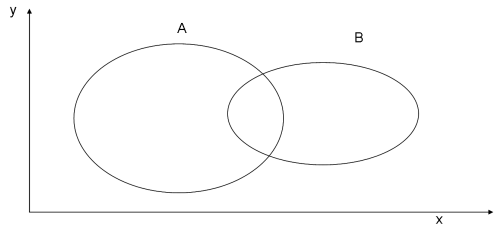

Fig. 3. Solving $\neg A \wedge B$

We want to solve $\neg A \wedge B$: the solution set is shown in Figure 4.

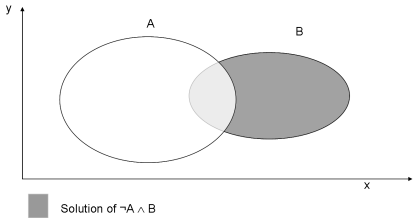

Fig. 4. Solution set of $\neg A \wedge B$

In the following, we propose two approaches to solving $\neg A \wedge B$.

First approach

We consider solving A by an outer approximation. When performing such an outer approximation of A, we keep the discarded parts. The discarded parts constitute an inner approximation of $\neg A$. However, by doing so, there are two risks that we may run into:

1) we may miss some solutions; and
2) we may get false positives.

In the following, we identify and describe these risks.

Risk of missing solutions

Let us examine the risk of missing solutions by focusing on the region where this problem may arise, as indicated in Figure 5. After we solved A by an outer approximation to get an inner approximation of $\neg A$, we solve B on the inner approximation of $\neg A$. When solving B, we get an outer approximation of B, which is delimited in part by the inner approximation of $\neg A$. Note that we initially considered an outer approximation of

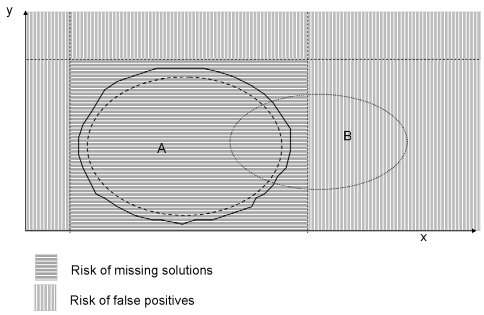

Fig. 5. Identifying risks

A. Therefore, our approximation for $\neg A$ may not contain all solutions that in fact belong to $\neg A$. Since we may lose some

solutions of $\neg A$, and B will be delimited by the approximation of $\neg A$, we have the risk of missing some solutions of $\neg A \wedge B$, colored in dark gray, as shown in Figure 6.

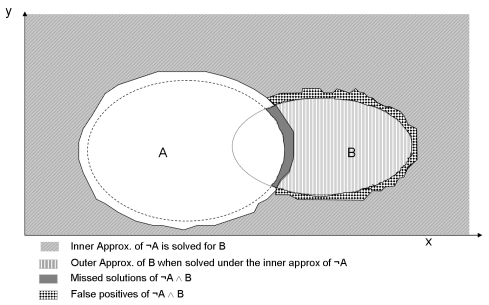

Fig. 6. Risk in the first approach

Risk of false positives
Now, let us examine the risk of having false positives. Note that when solving B, we computed an outer approximation of B. Therefore, we may get some solutions that originally did not belong to B. When solving for $\neg A \wedge B$, since we considered an outer approximation of B, we may have solutions that are not solutions, i.e., we may get false positives. These false positives are shown on Figure 6 colored in a black and white check board pattern.

Conclusion
One of our goals is to achieve completeness, i.e., not to miss any solution. Indeed, missing solutions may yield to the conclusion that the specification and the code are conform, while they are not, and could generate catastrophic situations. In this approach, correctness (i.e., no false positives) was not achieved, nor was completeness (i.e., no missed solutions).

A second approach, to address this problem, is presented hereafter.

Second approach

Let us recall that we want to solve $\neg A \wedge B$. In our second approach, instead of solving A using an outer approximation and considering the discarded parts as solutions for $\neg A$, making it an inner approximation of $\neg A$, we now consider solving A using an inner approximation and considering the discarded parts as solutions for $\neg A$,making it an outer approximation of $\neg A$.

Risk of false positives
Let us examine the risk of having false positives. Note that, in this approach, we only changed the way we solve $\neg A$. Therefore, we still have the same risk of false positives as in the first approach because when solving B, we still compute its outer approximation.

Furthermore, in the second approach, we may introduce more false positives. When solving $\neg A \wedge B$, we now consider an outer approximation of $\neg A$. As a result, we may get solutions that do not satisfy $\neg A$, and therefore, not $\neg A \wedge B$ either. The false positives that we may get on this example are shown on Figure 7.

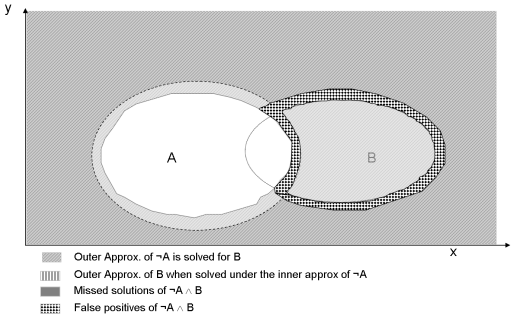

Fig. 7. Risk in the second approach

Conclusion
On one hand, when we consider the second approach, we achieve completeness, that is, we do not miss solutions when solving for A B. On the other hand, we do not achieve correctness, that is, we may introduce more false positives than we had when considering the first approach. This is shown in Figure 9.

Discussion of the two approaches

The two approaches we just described can be considered at different levels of risk: 1. The first approach involves the risk of missing solutions, and therefore to draw inappropriate / dangerous conclusions from the solving process. 2. The second approach is rather conservative, and there will not be any dangerous conclusion that the specifications and code conform together. On the other hand, there may be too many times when the code is rejected / not verified (at least in a first attempt), while it should not be.

Depending on the kind of targeted applications we are dealing with, one or the other approach may make more sense that the other one. In any case, the risk should be further quantified, in order to point out the gain / drawback and risk of using one approach or the other.

V. Conclusion

In this work, we considered the problem of verification of software. In particular,we were interested in proving that a program conforms to its specifications, using constraint programming techniques.

Our work extends and generalizes that of Rueher and Collavizza on the conformity of specifications and code. While their approach was handling programs dealing with computations on integers, our work aimed at also handling programs with floating-points computations. Besides, while they were making use of guarded constraints and of a SAT solver to detect inconsistencies early in the solving process, to avoid finding these inconsistencies after the costly enumeration process in normal CSP solvers, the use of SAT solvers is not efficient for handling programs dealing with computations on

floating-point numbers. In our work, we proposed an approach that does not include guarded constraints nor use a SAT solver. We pointed out the main two challenges related to our approach, and have extensively described algorithms to address one of them. In particular, two strategies to solve CSPs of the form $\neg A \wedge B$ were proposed and their properties pointed out. We also defined rules that prevent the computational explosion of our approach, hence addressing the second challenge as well.

As part of our future work, we will generalize our elimination rules to more complex combinations of statements, and show how we can control the growth of the problem size. We also plan to carry out a study of the gain observed in the number of false positives / false negatives between using our method with and without the elimination rules, and with the first approach or the second one. We anticipate that this study will not only allow us to draw conclusions regarding the level of risks, but will also help determine appropriate post-processing techniques to filter out false-positive results.

REFERENCES

[1] Osman Balci. *Validation, Verification, and Testing Techniques throughout the Life Cycle of a Simulation Study*. In Annals of Operations Research, pages 121173, 1994.

[2] Osman Balci. *Principles and Techniques of Simulation Validation, Verification, and Testing*. In WSC 95: Proceedings of the 27th conference on Winter simulation, pages 147154, 1995.

[3] Benjamin Blanc, Fabrice Bouquet, Arnaud Gotlieb, Bertrand Jeannet, Thierry Jéron, Bruno Legeard, Bruno Marre, Claude Michel, and Michel Rueher. *The V3F project*. In B. Blanc, A. Gotlieb, and C. Michel, editors, Workshop on Constraints in Software Testing, Verification and Analysis (CSTVA06), Nantes, 2006.

[4] Ashok K. Chandra and Vijay S. Iyengar. *Constraint Solving for Test Case Generation*. In ICCD 92: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors, pages 245248, Washington, DC, USA, 1992. IEEE Computer Society.

[5] Hélène Collavizza and Michel Rueher. *Exploration of the Capabilities of Constraint Programming for Software Verification*. In Hermanns and Palsberg [15], pages 182196.

[6] Frédéric Dadeau. *Évaluation symbolique contraintes pour la validation*. PhD thesis, Université de Franche-Comté, July 2006.

[7] Richard A. DeMillo and A. Jefferson Offutt. *Constraint-based Automatic Test Data Generation*. IEEE Trans. Softw. Eng., 17(9):900910, 1991.

[8] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2nd edition edition, September 2002.

[9] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. *Automatic Test Data Generation using Constraint Solving Techniques*. In ISSTA 98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis, pages 5362, New York, NY, USA, 1998. ACM Press.

[10] NASA. *Verification and Validation*. NASA website.

[11] Alexander Pretschner and Heiko Ltzbeyer. *Model-Based Testing with Constraint Logic Programming: First Results and Challenges*. In Proceedings of the 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verification (WAPATV01), Toronto, 2001.

[12] Allan M. Stavely. *Toward Zero-Defect Programming*. Addison-Wesley, 1st edition edition, September 1998.

[13] Richard C. Waters. *System Validation via Constraint Modeling*. SIGPLAN Notices, 26(8):2736, 1991.

[14] Jun Yuan, Carl Pixley, and Adnan Aziz. *Constraint-Based Verification*. Springer, 1 edition, January 2006.