A New Attempt to Use the Compiler Technology  
Supporting the Development of High-Reliable Software

Wei Huo, Hong-Tao Yu, Zhao-Qing Zhang, Ru-Liang Qiao, Xiao-Bing Feng

Advanced Compiler Technology Laboratory, Institute of Computing Technology, 
Chinese Academy of Sciences, Beijing 100080, P.R.China
{huowei, htyu, zqzhang, qrl, fxb}@ict.ac.cn

Abstract. The reliability of software has attracted more and more attention nowadays, and as we know, verification is an effective way to improve it. Most of the verification tools are, however, not so practical as expected. In this paper, an attempt is made to verify programs written with a subset of C language, in the help of program analysis techniques built-in compiler. A new approach to verify programs which are composed of normal assignments, if-branch and while-loop statements is presented. The new approach adopts a coarse path-sensitive use-define analysis based on GSA to generate the verification conditions. An automatic theorem prover is fed by the verification conditions to finish the verification. Then according to Hoare Logic, we argue that our approach is reasonable. Besides, some experiments using the compiler integrated with the automatic theorem prover are also reported, which confirm the feasibility of our approach.

Keywords: Software, High-Reliable, Verification, Theorem Proving, Symbolic Execution, SSA, GSA, Path-Sensitive Use-Define Analysis, Compiler, Compiler Technology, Hoare Logic

1. Introduction

The software crisis emerged in late of 1960’s. Many software projects ran over budget and schedule. Some of them caused economic losses, and some even consumed people’s life. For many long years, how to tackle the problems emerged in crisis was paramount to researchers and companies producing software tools. Unfortunately, not all of the problems had been well settled. Today, the increase of the scale and complexity of the network systems by leaps and bounds, the appearance of the multi-core architecture and the method of developing software distributed all make the software system larger, more complex and buggy. So the requirement of information safety and software verification techniques increases day by day. A proposal which is advocated by Tony Hoare named “Verified software—theories, tools and experiments” is presented. It suggests that the achievement of the project should be accelerated by a major international research initiative, modeled on a Grand
The Compiler Technology Supporting Development of High-Reliable Software

Challenge, with specific measurable goals.

To help programmers developing high-reliable software, verification tools which bridge the division between theory and praxis are indispensable. Some tools use symbolic analysis, path analysis and flow analysis etc. most of which have been implemented in modern compilers as mature analysis techniques. Generally speaking, most of verification tools adopt two main methods, theorem proving and model checking both of which are proved to be helpful in practice. But neither of them can meet the practical requirements totally. The former lacks of automatization and the latter faces the state explosion problem.

Modern compilers usually perform the translation from source code to target code through the operations of static and dynamic analysis and optimization. The classic analysis techniques such as control-flow analysis and data-flow analysis which are used by some of verification assistant tools, verification condition generator for instance, have been implemented in the compilers as mature and fundamental components. And all of the compilers are used to deal with practical software. So the techniques implemented in compilers are able to analyze large software and to provide enough information of programs for the verification tools.

So it is entirely possible that we apply these compiler techniques during the verification process and a verification-assistant compiler is developed. However, the analysis in verification aims at verifying the correctness of the software, while it in compilers helps to support optimization. The difference results in that we must adjust some analysis in compilers to fit in with needs of verification. Though one of our goals is to make full use of compiler technologies implemented in modern compilers, we should adjust some of them and/or add new ones.

In this paper an attempt is made to verify programs written with a subset of C language, in the help of program analysis techniques built-in ORC. We present a new approach which can assist to verify the normal assignments, as well as if-branch statements. In the approach, a coarse path-sensitive use-define analysis based on GSA is performed, then the compiler generates verification conditions based on the results of this analysis, and an integrated automatic theorem prover named CVC lite is called whenever it is needed to finish the verification. Our approach can also deal with loops in case that loop invariants are provided by programmers. Due to compiler technologies, our approach can generate the verification conditions very efficiently. For some simple cases, compiler alone can finish the verification directly, absent the cooperation of CVC Lite.

The rest of this paper is organized as follows: Section 2 covers related works. Section 3 expands on the approach of using the compiler technologies to support the verification of programs. Section 4 is about some implementation issues. Section 5 includes some cases. At last, we will proceed to conclusions and expectations in future work in Section 6.

---

1 Open Research Compiler, jointly developed by Advanced Compiler Technology Lab in Institute of Computing Technology and Inter Inc.
2 An automatic theorem prover developed by Stanford University, formal verification group.
2. Related Work

There are two main methods used in verification: theorem proving and model checking.

Specifications of a program are expressed in some logic form, based on the chosen logic system. The logic system is used to prove that the program satisfies the specifications. This is the main mechanism of theorem proving. The most popular logic system is Hoare logic. The famous verification tools based on this method are CVC lite, Boyer-Moore theorem prover, Simplify, PVS, Coq, HOL theorem prover etc.

Model checking is an automatic technique for verifying finite-state reactive systems. Specifications are expressed in temporal logic, and the reactive system is modeled as a state-transition graph. An efficient procedure is used to determine whether or not the state-transition graph satisfies the specifications. Some widely used model checkers are SPIN, Cadence SMV, CWB, SAL, SLAM, etc.

Compilers with the ability of verification for some programming language have been developed. The Spec# programming system developed by Microsoft Inc. is an example. It is a new attempt at a more cost effective way to develop and maintain high-quality software. The Spec# system consists of the Spec# programming language which is an extension of the object-oriented language C# with some features to support verification, the compiler which is integrated into the Microsoft Visual Studio development environment for the .NET platform and prepares meta data for verifier and the Spec# static program verifier.

Our approach is also inspired from symbolic execution. One of the steps in proof method of verification, the verification conditions generation can be done simply by symbolic execution. Instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may the symbolic formulas over the input symbols.

3. Verification of Program Correctness

It is our final goal to constructing a developing environment for high-reliable software. But now we focus on the use of compiler technologies supporting the verification of C language or its practical subset, in hope of making the verification tools practical and reducing the work of theorem proving tools which would be embedded in compiler. As an attempt, we only deal with a simple subset of C language including just integer variables and three types of statements, i.e. assignment, if-the-else and while statements right now.

3.1 Principle Description

The validity of the results of a program will depend on the values taken by the variables before that program is initiated. These initial preconditions of successful use
can be specified by the same type of general assertion as is used to describe the results obtained on termination. To state that, the form of \( p \parallel S \parallel q \), so called asserted programs where \( p, q \) are assertions to specify the initial state and final state of the program (or part of program) \( S \), is introduced by Hoare Logic system. The intuitive meaning of \( p \parallel S \parallel q \) is as follows: whenever \( p \) holds before the execution of \( S \) and \( S \) terminates, the \( q \) holds after the execution of \( S \). From the Hoare Logic’s perspective, a logic system which is composed of axioms and several rules is used to reason about the asserted programs. However, from the compiler’s perspective, we want to check whether the result after the execution of the programs is the same as expected. Inspired from this perspective, we think of using the compiler techniques to transform the whole construct of \( p \parallel S \parallel q \) to the form of \( p' \parallel S' \parallel q' \). We believe the transformation is semantically equivalent if the correct compiler techniques are preformed. So the correctness of \( S \) specified by \( p \) and \( q \) is equivalent to the correctness of \( S' \) specified by \( p' \) and \( q' \). If \( S' \) is empty, the verification reduces to prove

\[
p' \vdash q'.
\]

If not, we hope the compiler digs the information of the programs as much as possible to make the transformation simple enough. Consequently, the work of the verification tools is reduced.

To put the idea into practice, we treat the \( p \) and \( q \) as normal expression in C, and they are assigned to some special boolean variables respectively. As a result, we get the new program \( N \_ S \) that is \( \{ p ; S ; q \} \). The compiler processes \( N \_ S \), and then what we get is the transformed \( N \_ S' \) that is \( \{ p' ; S' ; q' \} \). We will also show that the approach is based on Hoare Logic, so it’s reasonable.

### 3.2 Process of Assignment Statement

Assignment is undoubtedly the most characteristic feature of programming a digital computer. Axiom of assignment is the fundamental of Hoare Logic, with which our approach must accord. The axiom of assignment is

\[
\vdash \{ p[t/x] \} x := t \{ p \}. \tag{1}
\]

Now the use-define analysis is performed to process

\[
N \_ S = \{ x := t ; p \}. \tag{2}
\]

We get the transformation of

\[
N \_ S' = \{ p[t/x] \}. \tag{3}
\]

From (2) and (3), if we can prove

\[
\{ p[t/x] \} \vdash \{ p[t/x] \}. \tag{4}
\]

then (1) is accorded with. Obviously, (4) is true. So our approach is reasonable for

---

\(^3\) As usual, \( p[t/x] \) stands for the result of substituting \( t \) for the free occurrences of \( x \) in \( p \).

\(^4\) Note that left hand is from (1) and right hand from (4).
assignment statement.

Our use-define analysis is based on an intermediate representation called Single Static Assignment (SSA) form. A procedure is in SSA form if every variable assigned a value in it occurs as the target of only one assignment. In SSA form use-define chains are explicit in the representation of a procedure: a use of a variable may use the value produced by a particular definition if and only if the definition and use have exactly the same name for the variable in the SSA form of the procedure. To distinguish the different definitions of same variable, a unique version is given to each definition. Meanwhile, when several definitions of a variable reach a control confluent point of the control-flow graph of the program, \( \Phi \)-function is introduced to keep the single assignment property. An example is show in Fig. 1.

After the use-define propagation, we can easily get the \( \{ y_1 = 4 \} \) is true in example.

![Fig. 1. SSA form of sequence statements](image)

### 3.3 Process of If-Then-Else Statement

For the if-branch statement \( \{ p \} \) if e then \( S_1 \) else \( S_2 \) \( \{ q \} \), the normal use-define analysis is not applicable. The execution of the program depends on path conditions. Whether they are taken is determined only at run time. Therefore, we must verify the correctness of program on each possible execution path. We exploit the path-sensitive use-define analysis to deal with branch statement. If a variable demands the value through a merge node, we should first process the predicate that determines the path to follow. After this analysis, we propagate the symbolic value of each variables with the path predicates taken into account. Then we replace the variables appearing in the post-condition with their path-sensitive values and several transformed post-conditions which can be proved easier than before are generated. The number of transformed post-conditions is in direct ratio to the number of execution paths. The whole process could divide into four main steps described below.

#### 3.3.1 Construct GSA

The path-sensitive use-define analysis is based on an intermediate representation
The Compiler Technology Supporting Development of High-Reliable Software

called Gated Single Assignment (GSA), which is an extension SSA form. The $\Phi$-function in SSA form is not directly interpretable; there is no mechanism to discriminate among the various definitions that reach a given $\Phi$-function. Thus, we have no information about which path may or may not be taken. Augmentation of the $\Phi$-function is needed to include this additional information. So the gating functions are introduced to replace $\Phi$-functions in SSA form. They can capture the control conditions that determine which of the definitions reaching a given $\Phi$-function will provide the value for the function. One such gating function, $\gamma(p,x_1,x_2)$, which can be read as a simple if-then-else, is introduced. An example is show in Fig. 2.

![Fig. 2. SSA and GSA form of program fragment with branch](image-url)
3.3.2 Path-Sensitive Use-Define Analysis and Value Propagation

After constructing the GSA form of the program, all of path predicates which have the effect on use of variables are represented in $\gamma$-functions. Thus, we can combine the basic blocks included by an if-branch statement to form an abstract node. We call this kind of node if-region. Fig. 3 shows this process. This is an application of structural analysis about if-then-else or if-then constructs. The order of the (1) and (2) in the

![Diagram](image)

Fig. 3. Structural Analysis for program fragment (2)

Fig. 3 (b) is not important regardless of the side effect of two statements, because there are different versions to denote the two different definitions and what we concern is the symbolic values of the variables, not the execution result of if-region. Then we can propagate the symbolic values of variables from their definitions to the uses directly except the two conditions:

1. The definition reaching the use of a variable in a normal statement is $\gamma$-defined.
   
   $$x_3 = \gamma(P, t_1, t_2);$$
   
   ...  
   
   $$y_1 = (x_3 + 1 \leq 3)$$

2. The definition reaching the use of a variable in $\gamma$-function is $\gamma$-defined.
   
   $$x_3 = \gamma(P, t_1, t_2);$$
   
   ...  
   
   $$y_1 = \gamma(P_1, t_3, t_4);$$

For the first case, we generate a new $\gamma$-function

$$y_1 = \gamma(P, t_1 + 1 \leq 3, t_2 + 1 \leq 3)$$
The Compiler Technology Supporting Development of High-Reliable Software

for y1. For the second case we generate a new $\gamma$-function

$$y_1 = \gamma(P1 [ \gamma(P, t_1, t_2)/x3], \gamma(P, t_1, t_2), x4)$$

for y1.

3.3.3 Generate Transformed Post-Conditions

After all propagations shown above have been performed, we get the final transformation of post-condition $q'$ which is defined by $\gamma$-function whose arguments may also include $\gamma$-function. We would expand the $\gamma$-function to get the provable post-conditions.

1. If the predicate of the $\gamma$-function includes another $\gamma$-function, we can expand it as follows:

$$q' = \gamma(\gamma(P1, V1, V2), x1, x2) \Rightarrow q'1 = \gamma(P1 \land V1, x1, x2)$$

$$q'2 = \gamma(\neg P1 \land V2, x1, x2)$$

2. If the value arguments of the $\gamma$-function include another $\gamma$-function, we can expand it as follows:

$$q' = \gamma(P, \gamma(P1, V1, V2), x2) \Rightarrow q'1 = \gamma(P \land P1, V1, x2)$$

$$q'2 = \gamma(P \land \neg P1, V2, x2)$$

In this process, a simplifier for boolean expressions can be used for the predicates in $\gamma$-function. The unreachable paths could be gotten rid of when the predicates are simplified to false. While the predicates are simplified to true, only one path is taken and the $\gamma$-function is transformed back to normal assignment. This is an effective way to avoid path explosion.

3.3.4 Generate Verification Conditions

For the if-branch statement $\{p\} if e then S1 else S2 \{q\}$, after the transformation above, we could get the form $\{p\} if e then S1' else S2' \{q' = \gamma(e, t1, t2)\}$. Then we generate the verification conditions:

1. $\{p \land e\} S1' \{t1\}$
2. $\{p \land \neg e\} S2' \{t2\}$

It’s easy to see that, the verification conditions are equivalent to the if-then-else rules in Hoare Logic.

$$\{p \land e\} S_1 \{q\}, \{p \land \neg e\} S_2 \{q\} \quad \text{if-then-else Rule (1)}$$

From the definition of $\gamma$-function and the transformation, $\{p \land e\} S1' \{t1\}$ is equivalent to $\{p \land e\} S1 \{q\}$. And also $\{p \land \neg e\} S2' \{t2\}$ is equivalent to $\{p \land \neg e\} S2 \{q\}$ for the same reason. So our approach accords with Hoare Logic.

We then pass both conditions to some theorem proving tool to finish the verification.

Normally, S1' and S2' are both empty. The two conditions can be reduced to

$$\{p \land e\} \models \{t1\} \quad \text{(2)}$$

$$\{p \land \neg e\} \models \{t2\} \quad \text{(3)}$$

Each of them can further transform to
respectively by the logic rules

\[ \{p\} \vdash \neg e \lor \{t1\} \quad (4) \]
\[ \{p\} \vdash e \lor \{t2\} \quad (5) \]

For the example given in Fig. 3 (b), the two conditions that we get after path-sensitive use-define analysis and value propagation are \{2+1 \leq 3\} and \{1+2 \leq 3\}, both of which are true obviously even without the help of third part verification tools.

### 3.4 Process of Loop Statement

In our approach, the programmer must provide invariants about the loop statements. With the assumption of truth of the invariants, we generate the necessary assertions about the invariant according to the rule of loop in Hoare Logic. The example is shown in Fig. 4.

The four assertions generated are denoted by (1) … (4). But they have different properties. The (1) and (3) in Fig. 4(b) are regarded as post-conditions which need to prove, however the (2) and (4) are regarded as preconditions which are assumed to be true. We then pass the programs generated with assertions to the compiler. After kinds of compiler optimization processes, especially path-sensitive use-define analysis and value propagation, loop invariant code motion and induction-variable recognition, the program can be transformed to the form shown in Fig. 5.

Then we have three pairs of verification conditions to feed the verification tools, there are

1. \{p’\} S1’; _CV_invariant’
2. _CV_invariant_entry’; S’; _CV_invariant_exit’
3. _CV_invariant_post’; S2’; \{q’\}

The first pair is used to prove the establishment of loop invariant. The second pair says the invariant holds and the third pair is used to prove the post-condition with the additional information provided by the exit of loop. If the first two pairs could be proved, we conclude that the loop is correct with respect to the invariant given by programmer. If the third is verified to be true, the whole program is correct with the specification of \{p\}, invariant and \{q\}. There is problem if the value of a variable defined in S’ is related to the number of iterations of loop and that variable is used in S2’ or q’. We can’t get the exact symbolic value about the variable, so the proving of the third pair may fail. This is a very difficult problem, and we will try to solve it in the future work.

Because the assertions about loop invariant are generated following the rule of loop of Hoare Logic, it accords with the Hoare Logic obviously.
The Compiler Technology Supporting Development of High-Reliable Software

3.5 Integration with Automatic Verification Tools

The compiler can dig information about programs as much as possible, but it’s not good at boolean expression’s simplification and logic reasoning. For one hand, the ability of logic reasoning is a must for verification. And for the other, we will benefit by the logic reasoning for the further optimization in compiler. Therefore, we integrate compiler with an automatic verification tool. We make a clear and seamless interface between the compiler and the tool, and we can call the tool to interact with the compiler as long as the compiler meets the need of boolean expression simplification and logic reasoning. With the help the tools, we can finish verification after the processes described above.

Fig. 4. Generating the assertions about invariant for program fragment with while loop

(a) fragment (3)

(b) Generating assertions for (3)
4. Implementation Issues

Some detail issues about implementation are discussed here.

4.1 Language Issue

Now, our compiler framework for verification can only support a simple subset of C language. The data type can be integer, float-point and character. The statement can only be assignment, if-branch and while-loop statements. The programmer must provide preconditions, post-conditions and invariants if loops exist in programs. Our implementation about assertions just supports proposition logic, and we use arithmetic, logic and relation operations in C language to express the assertions.

4.2 Compiler Reconstruction

We build our experiment platform on the basis of Open Research Compiler, which is a high-performance open source advanced compiler jointly developed by Advanced Compiler Technology Lab in Institute of Computing Technology and Intel Inc. We extend the parser so as that the compiler could recognize the assertion specified by the keywords “requires”, “assures” and “invariant”, which designate the precondition, the post-condition and loop invariant respectively. Our parser recognizes the assertions, treats them as the normal C statements with the additional assertion attributes and passes the parsing results to the next components of compiler. Fig. 6 gives an example.
The Compiler Technology Supporting Development of High-Reliable Software

In the middle end of the compiler, we implement the path-sensitive use-define analysis and value propagation based on GSA form, which can be further used in compiler optimization besides verification. Based on the result of the analysis, we write a procedure to generate the verification conditions for the program.

Also, we integrate the compiler in its middle end with an automatic theorem prover named CVC lite. CVC Lite is a fully automated theorem prover (or validity checker) for a many-sorted (i.e. typed) first-order logic with interpreted theories, including some support for quantifiers, partial functions, and predicate subtypes. The currently implemented interpreted theories are real and integer arithmetic (linear and some support for non-linear), uninterpreted functions, arrays, records, and bitvectors.

```
void division( int x, int y, int q, int r )
//@ requires x >= 0 && y >= 0 ;
//@ ensures x == q*y+r && r >= 0 && r<y ;
{
    q  = 0;
    r = x;
    while (r >= y)
    //@ invariant x == q*y+r && r >= 0;
    {
        r = r - y;
        q = q + 1;
    }
}
```

Fig. 6. Program with specifications

5. Experiment Results

We only do experiment using little programs now. Experiments about complicated and large programs are being done. Fig. 7 and Fig. 8 show two examples.

For the first case of fragment of Max shown in Fig. 7 (a), there are four transformed post-conditions generated by our approach. Then they are passed to the automatic theorem prover to finish verification. For the second example of division, the three pairs of assertions have been generated by our compiler. Then we would pass the three pairs \( (1) \Rightarrow (2), (3) \Rightarrow (4) \text{ and } (5) \Rightarrow (6) \) shown in Fig. 8 (b) to CVC lite to check their validities. The result is they are all proved.

From our experiments, our approach has several advantages:

1. We use the mature compiler techniques and their implementations have been validated for long time, so the verification condition generations are efficient and not prone to bugs. And all these implementations aim to complex and large programs,
which enhances the practicability of our approach.

2. For some cases, the correctness can be proved directly after the programs processed by compiler.

3. For the cases which can be proved directly by tools, the compiler still has effect on them. The compiler can reduce the work of tools by use of use-define links of the variables, dividing the assertions about branches down to individuals one for each execution path and digging deeper relations between variables. Some cases which can’t be proved directly by verification tools can be proved in the help of our approach. For example, the example in Fig 8 (a) can’t be verified by Spec# system directly. But after the process of compiler, the three pairs of assertion of it could be checked, and the program can be proved. And also we find that the compiler can get a lot of deeper information about loops’ properties and the induction variables. The information is very useful in verifying loops.

```plaintext
//@requires true;
if (a >= b)
    Max = a;
else
    Max = b;
if(c > Max)
    Max = c;
//@ensures {(Max == a && a>= b && a>=c) || (Max == b && b >= a && b >= c) || (Max == c && c>= a && c>= b)};
```

(a) fragment (4) of Max
The Compiler Technology Supporting Development of High-Reliable Software

(b) The result of fragment (4) after compiler processing

Fig. 7. Example: Max of three numbers
//@requires x>=0 && y>=0;
//@ensures x==q*y+r && r>=0 && r<y;
{
  q=0; r=x;
  while (r>=y)
  //@invariant x==q*y+r && r>=0;
  {
    r=r-y; q=q+1;
  } return q; }

(a) fragment (5) of Division

{ _CV_precondition = (x >= 0 && y >= 0);    (1)   
  _CV_invariant0 = x >= 0;         (2)   
  for(r0 = 0; y <= (x - (y * r0)); r0 = r0 + 1 ) {
    _CV_invariant_entry0 = ((x - (y * r0)) >= 0 )
    && (y <= (x - (y * r0)));(3)
    _CV_invariant_exit0 = (x == ((y * (r0 + 1 )) +
    ((x - (y * r0)) - y)) && (((x -
    (y * r0)) - y) >= 0 )); (4)
  }
  _CV_invariant_post0 = ((x - (y * r0)) >= 0) && (y
  > (x - (y * r0)));        (5)
  _CV_postcondition = (((x - (y * r0)) >= 0 ) && (y
  > (x - (y * r0))));
  (6)
  } return r0; }

(b) The result of fragment (5) after compiler processing

Fig. 8. Example: division
6. Conclusions

In this paper, an attempt is made to verify programs written with a subset of C language, in the help of program analysis techniques built-in compiler. A new approach to verify the programs which are composed of normal assignments, if-branch and while-loop statements is presented. The new approach adopts a coarse path-sensitive use-define analysis based on GSA to generate the verification conditions. An automatic theorem prover is fed by the verification conditions to finish the verification. Then according to Hoare Logic, we argue that our approach is reasonable. Besides, some experiments using the compiler integrated with the automatic theorem prover are also reported, which confirm the feasibility of our approach.

In the future work, specification expressions with quantifiers will be supported and the expressing ability of assertions will be extended to first-order or higher-order logic. The language we can deal with in our compiler framework will be a practical subset of C language, which may include array, pointer, user-defined data types and switch, for-loop, call statements etc. The recursive function will also be supported. We want to add accurate pointer analysis to the compiler, which will be used to verify pointers. Also, we will automatically generate some kinds of loop invariants, which can enhance the practicability of our approach. Meanwhile, we want to design a software developing methodology which is based on “Design by Contracts” (DBC) to support the verification about function calls. Our ultimate object is to construct a developing environment with verifying compiler for high-reliable software.

Acknowledgement We would like to thank Ping-Yu Zhang and Feng Li for their contributions to the work described in this paper.

References

22. CVC lite: http://www.cs.nyu.edu/acsys/cvc/