Exploring Abstraction Techniques for Scalable Bit-Precise Verification of Embedded Software

Nannan He

Dissertation submitted to the Faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Michael S. Hsiao, Chair
Amos L. Abbott
Yaling Yang
Allen B. MacKenzie
James D. Arthur
David Yang Gao

May 1, 2009

Bradley Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

Keywords: SAT, Bounded Model Checking, Program Analysis, Abstraction and Refinement

Exploring Abstraction Techniques for Scalable Bit-Precise Verification of Embedded Software

Nannan He

(ABSTRACT)

Conventional testing has become inadequate to satisfy rigorous reliability requirements of embedded software that is playing an increasingly important role in many safety critical applications. Automatic formal verification is a viable avenue for ensuring the reliability of such software. Recently, more and more formal verification techniques have begun modeling a non-Boolean data variable as a bit-vector with bounded width (i.e. a vector of multiple bits like 32- or 64- bits) to implement bit-precise verification. One major challenge in the scalable application of such bit-precise verification on real-world embedded software is that the state space for verification can be intractably large.

In this dissertation, several abstraction techniques are explored to deal with this scalability challenge in the bit-precise verification of embedded software. First, we propose a tight integration of program slicing, which is an important static program analysis technique, with bounded model checking (BMC). While many software verification tools apply program slicing as a separate preprocessing step, we integrate slicing operations into our model construction and reduction process and enhance them with compilation optimization techniques to compute accurate program slices. We also apply a proof-based abstraction-refinement framework to further remove those program segments irrelevant to the property being verified. Next, we present a method of using symbolic simulation for scalable formal verification. The simulation involves distinguishing X as symbolic values to abstract concrete variables' values. Also, the method embeds this symbolic simulation in a counterexample-guided abstraction-refinement framework to automatically construct and verify an abstract model, which has a smaller state space than that of the original concrete program.

This dissertation also presents our efforts on using two common testability metrics — *controllability metric* (CM) and *observability metric* (OM) — as the high-level structural guidance for scalable bit-precise verification. A new abstraction approach is proposed based on the concept of under- and

over-approximation to efficiently solve bit-vector formulas generated from embedded software verification instances. These instances include both complicated arithmetic computations and intensive control structures. Our approach applies CM and OM to assist the abstraction refinement procedure in two ways: (1) it uses CM and OM to guide the construction of a simple under-approximate model, which includes only a subset of execution paths in a verification instance, so that a counterexample that refutes the instance can be obtained with reduced effort, and (2) in order to reduce the cost of using proof-based refinement alone, it uses OM heuristics to guide the restoration of additional verification-relevant formula constraints with low computational cost for refinement. Experiments show a significant reduction of the solving time compared to state-of-the-art solvers for the bit-vector arithmetic.

This dissertation finally proposes an efficient algorithm to discover non-uniform encoding widths W_e of individual variables in the verification model, which may be smaller than their original modeling width but sufficient for the verification. Our algorithm distinguishes itself from existing approaches in that it is path-oriented; it takes advantage of CM and OM values to guide the computation of the initial, non-uniform encoding widths, and the effective adjustment of these widths along different paths, until the property is verified. It can restrict the search from those paths that are deemed less favorable or have been searched in previous steps, thus simplifying the problem. Experiments demonstrate that our algorithm can significantly speed up the verification especially in searching for a counterexample that violates the property under verification.

To my grannies.

Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Michael S. Hsiao, for his timely guidance, continuous inspiration and support to my research work. His encouragement and help are very important throughout my Ph.D. study.

I would like to thank Dr. Amos Lynn Abbott, Dr. Yaling Yang, Dr. Allen B. MacKenzie, Dr. James D. Arthur and Dr. David Yang Gao to serve on my final defense committee and spend precious time on the dissertation review and in attending the oral presentation. I am also very grateful to Dr. Sandeep Shukla, Dr. Shawn Bohner and Dr. Layne T. Watson for their precious suggestions during the formation of this dissertation.

I would like to express my deep thanks to all professors, alumni and current students in the Center for Wireless Telecommunications. I feel very lucky in joining such wonderful projects, working with such a wonderful team and obtaining the great opportunity to develop my Ph.D. research in the software verification.

I would like to give my sincere gratitude to all Proactive alumni and current members for their valuable comments on my research and dissertation. Each week, I much enjoyed the group meeting time and afterwards discussions with them in the lab.

I am also very grateful to the people from Blacskburg Chinese Bible Study, International Christian Fellowship and One Accord group for their invaluable care and support in my both spiritual life and worldly life, especially at the hard times that we experienced together.

Last but not the least, I would like to give my special thanks to my parents for their love and support to

me during the pursuit of my graduate study.

Nannan He

April, 2009

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded software is playing an increasingly important role in many safety-critical or mission-critical applications. Since an error may potentially cause disastrous results or severe economic consequences in these applications, the correctness of the software must be rigorously tested before it is put into operation. Statistics show that a large fraction of resources is devoted to guarantee the software correctness in most software development practices.

Software verification and validation (V & V) aims to assure that the software correctly implements all the requirements. Technically speaking, software verification checks that the software product produced after each development step satisfies the given design specifications. It has two main aspects: (1) the software does what it is supposed to do, and (2) the software does not do what it is not supposed to do. Both of these aspects are important and have become great concerns in the software development. Software validation checks that the design specification meets the intended user requirements. Operating under the assumption that the specification already matches the user requirements, we focus on the software verification issues in this dissertation.

1.1 Software Verification

1.1.1 Overview

Testing and formal verification are two fundamental software verification approaches. Generally speaking, testing is an attempt to show the presence of bugs in the software, while formal verification primarily aims to prove the absence of bugs with respect to specified properties. As testing and formal verification each has its own benefits and drawbacks, the selection of approaches in practice depends on many factors, such as the required degree of correctness and the available verification resources in the software development.

Testing is still the mainstream approach in software verification practices to detect bugs, and demonstrate confidence in the software quality before the product is delivered, mainly due to its automation capability and simplicity. Many techniques have been established for software testing [12]. Based on whether or not the internal information of the software is used, they can be basically classified into two groups: black-box testing and white-box testing. Black-box testing only takes the external information such as design requirements of the software to generate test cases without considering any internal details. It is primarily applied in the large test scope; for example, testing the functional or non-functional requirements of the entire software system. White-box testing derives tests mainly from the internal perspective of the software, such as the program structure in the implementation. It is typically used to test small-scale program units at the early testing phases, such as unit testing.

One of the main advantages of testing techniques is the ease of use. A test practice is done simply to apply an input stimulus to initiate the program execution, and compare whether the actual outputs equal to the expected ones. If so, we say the software passes the test; otherwise, the errors are reported for debugging. In addition, testing can be independent from the complex implementation details as in black-box testing. This feature also makes testing the only verification choice when the source code is not available. Furthermore, when encountering verification resource constraints such as memory or execution time, testing can still provide some coverage, while formal verification may not be successfully conducted to report meaningful coverage numbers. However, automatic generation of high quality test cases and the accuracy of coverage metrics for measuring testing quality are still two very challenging

issues of software testing. Moreover, since it is rarely able to achieve one hundred percent input coverage, software testing is inadequate to guarantee program correctness in the reliability-critical embedded applications, even if the program passes all the available tests.

1.1.2 Formal Verification

Formal verification conducts an exploration of all the possible behaviors based on the formal models of the program and the formal specification of the intended requirements. The main advantage of formal verification is the completeness it offers in terms of the specific properties, which can eliminate the notion of inadequate coverage that conventional testing faces. This is a very attractive feature in reliability-critical embedded applications. Furthermore, some automatic formal verification methods like model checking can be used to detect hard corner-case bugs, which are very difficult to be detected by testing alone. With the significant advances in automatic reasoning and the computing capability of modern computers, formal verification is no longer only of academic interest. However, the limited scalability is still the major problem of most formal methods when dealing with practical applications, and so they are often used to verify the small portions of code, but cannot directly handle the large scale and complex programs in the real-world.

Several formal verification techniques have been proposed that can be basically classified into two main categories. One is deductive verification such as theorem proving; the other is model checking. The basic idea of deductive verification is to use a set of axioms or mathematical rules to prove program correctness. Although some tools have been developed to aid the correct use of axioms and proof rules, deductive verification is still hard to be automated and is an extremely time-consuming process that can be performed only by experts with considerable experience in logical reasoning and mathematics. This makes deductive approaches less attractive in practice.

On the other hand, model checking is used to verify the specified property of the finite state model defined by the software, through an explicit or implicit enumeration of all the reachable states and behaviors. It attempts to assure the correctness of the software, as it has shown the capability of finding subtle bugs, and formally verifying the correctness of complex hardware designs, with respect to a target reliability property. Model checking can be fully automatic without requiring the users to be experts.

Furthermore, when the design being verified fails to satisfy a given specification, counterexamples are generated, which show the erroneous behaviors of the design. This information can be very valuable for debugging. Model checking has been successfully applied to the formal verification of the real-world hardware designs in industry.

Figure 1.1 compares the verification efforts and assurance capabilities among several verification techniques: testing, model checking, and deductive verification. From the aspect of verification efforts, testing is the easiest approach, while deductive verification is the hardest one as it can not be fully automated. Model checking lies in between, mainly because some model construction processes still need the manual assistance, although the checking can be automatic. From the aspect of assurance capabilities, deductive verification and model checking as formal methods can provide the complete guarantee of program correctness. Figure 1.1 shows that deductive verification has the highest assurance capabilities since it can deal with both finite and infinite state systems, while model checking is restricted to the finite ones. In general, testing can only provide the partial assurance of program correctness, since the application of exhaustive stimuli is often prohibitive.

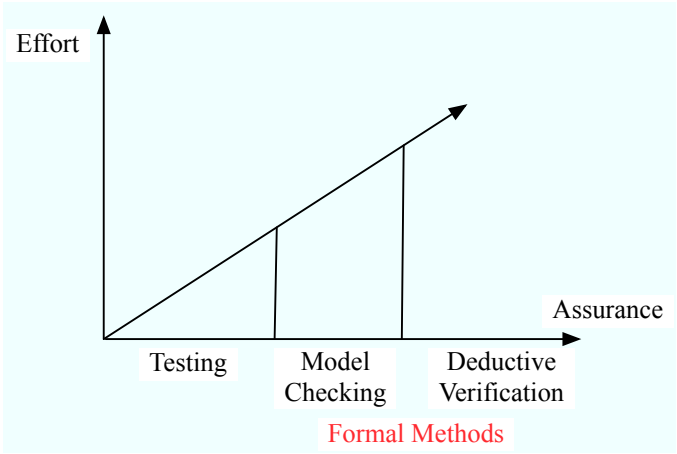


Figure 1.1: Effort versus assurance capability

Data modeling, as a very important issue in formal verification [19], can be conducted in different manners. One data model is the bit-level modeling where each bit is represented individually. It is the basis of most computer aided verification. Although this bit-level modeling is precise, it may make it hard to extract the functionality, due to the excessive details that come with individual bits. And such modeling has very limited scalability. Another form of data modeling is using symbolic words, where

each word can have an arbitrary value like unbounded integers. In the past, this approach was used in the software verification to model program variables. But this data modeling is not precise, as it does not allow for the detection of arithmetic underflow and overflow bugs, which frequently occur in embedded software. Moreover, the existing theorem provers based on the logics with unbounded word can only provide the limited reasoning for real program properties, since they mainly target on mathematical theories.

Recently, modeling data variables as *bit-vectors* with bounded width has shown some unique benefits. Bounded data modeling is capable of capturing precisely the semantics of the verification instances constrained by a physical word-size on a computer. With the advances in propositional and bit-vector arithmetic reasoning, the formal verification with this data modeling has the potential to deal with large problems. Many existing software model checking tools (e.g., CBMC [35], SATABS [36], Saturn [115], F-SOFT [70]) and hardware design validation techniques (e.g., [66, 82]) have taken the bit-vector modeling of program variables. We also adopt it in our software verification work. In this dissertation, we define the term *bit-precise verification*, which **specifically** represents verification techniques using such bit-vectors based data modeling.

1.2 Contributions of this Dissertation

With the successful application of model checking to hardware designs, there is a growing interest in applying automatic formal techniques to verify the embedded software programs, whose safety and reliability are critical. However, the limited scalability is still the major problem of formal techniques, and a lot of research has been invested in alleviating this problem. In this dissertation, several efficient abstraction techniques are explored for scalable bit-precise verification of embedded software.

1.2.1 Scalable BMC via Integration of Program Slicing and Proof-based Localization Abstraction Refinement

Since the properties under verification usually depend on a small portion of a program, we propose an approach to accurately and efficiently find this portion with low computational cost toward the goal

of enhancing the scalability of software BMC. The proposed approach tightly integrates an aggressive static program slicing approach in the software verification model construction and reduction process. Program slicing is an important program analysis technique that can reduce the entire program to segments relevant to a particular computation. This allows for effective removal of those program segments that are *computationally irrelevant* to the property, so as to significantly reduce the model complexity. Moreover, our slicing operations naturally combine the compilation optimization techniques, such as constant propagation, to compute the accurate program slice.

We further explore a proof-based localization abstraction-refinement strategy, using under- and over- approximation of our software verification model to abstract the program segments, which are *verification relevant* to the property for scalable BMC. A heuristics method by program analysis is also proposed to effectively refine the under-approximation in each iteration.

1.2.2 BMC via Symbolic Simulation

We also explore the potential of combining symbolic simulation with localization abstraction [78] for scalable formal verification. The proposed approach applies distinguishing X as symbolic values to abstract the concrete variables' values, so that a limited number of identified symbolic input vectors can cover the complete input space. In order to reduce the verification cost and ease the identification of symbolic input vectors, symbolic simulation is used in a counterexample-guided abstraction-refinement framework to automatically construct an abstract model. This model includes a subset of property-relevant program constraints, which can be more easily verified than the original program. The property-relevance of program constraints is automatically determined during the iterative abstraction refinement procedure.

1.2.3 Testability-guided Abstraction to Solving Bit-vector Arithmetic

We investigate two testability metrics — controllability metric (CM) and observability metric (OM) — as the high-level structural guidance for the scalable bit-precise verification. To speed up solving bit-vector formulas generated from control-dominated embedded software verification instances, A new

abstraction approach is proposed based on the concept of under- and over-approximation. We also design a new CM and OM computation method and apply CM and OM to efficiently guide the proposed abstraction procedure. The under-approximate model is built by enforcing constant constraints to a small set of single-bit variables that control the branch selection of If-Then-Else (ITE) nodes. This restricts the search space to only a subset of formula constraints. With the guidance of CM and OM, the proposed approach can quickly find a satisfying solution on the easily controllable portion of the formula if a solution exists.

Our approach computes the over-approximate abstraction via the learning from an UNSAT proof of the under-approximate model. It also restores additional verification relevant constraints according to the OM metric to reduce the refinement cost by only using the UNSAT proof. As a result, an abstract model that is accurate enough to the verification can be built quickly, long before all partitions are enumerated. To start a new refinement iteration, the satisfiable assignment (solution) of the current over-approximate abstraction is used to guide the construction of a new under-approximate model with an unvisited portion in the formula. With the proposed approach, the verification can be conducted incrementally due to its partition-based feature.

1.2.4 A Reduced Bit-vector Encoding Width Computation Algorithm

We finally propose an efficient algorithm to iteratively discover non-uniform encoding widths W_e of variables in the verification model, which may be smaller than their original modeling widths, but sufficient to the verification. Different from existing approaches [20], the proposed algorithm is path-oriented in that it takes advantage of the CM and OM values to guide the computation of the encoding widths in three ways: (1) it computes initial non-uniform W_e of variables on different paths; (2) it enlarges the W_e of the easily-controllable variables first in the encoding width adjustment steps (if necessary); (3) it sets W_e to zero for some single-bit variables that determine the path(s) selection so as to concentrate searching only on a subset of paths. Our approach is capable of restricting the search from those paths that are deemed less favorable or have been checked in earlier steps, thus greatly simplifying the verification problem.

1.3 Organization of the Dissertation

The remainder of the dissertation is organized as follows. The next chapter describes the background of program representation, SAT problem, software model checking, especially SAT-based bounded model checking and abstraction techniques. The details behind the proposed verification approach of integrating program slicing with the proof-based abstraction for scalable BMC are presented in Chapter 3. Chapter 4 discusses our proposed verification approach of combining symbolic simulation with counterexample-based weakening abstraction for formally verifying the embedded software. Chapter 5 describes our new testability guided abstraction technique for speeding up solving bit-vector arithmetic instances derived from the verification of the embedded software. Our path-oriented bit-vector encoding width computation algorithm is presented in Chapter 6. Finally, the conclusion and future work are presented in Chapter 7.

Chapter 2

Background

In this chapter, we provide the necessary background knowledge related to our work. We first give several definitions about the program representation used throughout this dissertation. Then, in Section 2.2, we introduce some basics about the satisfiability problem. In Section 2.3, we review the SAT-based bounded model checking technique, in particular, its application to software formal verification. Finally, we introduce some automatic abstraction techniques in Section 2.4.

2.1 Definitions of Program Representation

Definition 1. *(digraph).* A directed graph, or digraph, G consists of a nonempty set of nodes N, and a set of ordered pairs between distinct nodes E ($E \subseteq N \times N$). Each ordered pair is called a directed edge or an arc.

Definition 2. *(basic block).* A Basic Block (BB) contains one or more program statements in the straight-line sequence with a conditional or unconditional *goto* statement only at the end.

Definition 3. *(guard).* The guard of a BB is the conjunction of a set of conditional predicates, which must be satisfied in order to enter this BB.

Definition 4. *(control-flow graph).* A Control Flow Graph G_{CF} is a digraph representation of the program, where each node represents a BB, and each directed edge represents the execution flow from one BB to another.

Definition 5. *(branch-merge).* In a G_{CF}, a node with more than one successor is called Branch; a node with more than one predecessor in G_{CF} is called Merge.

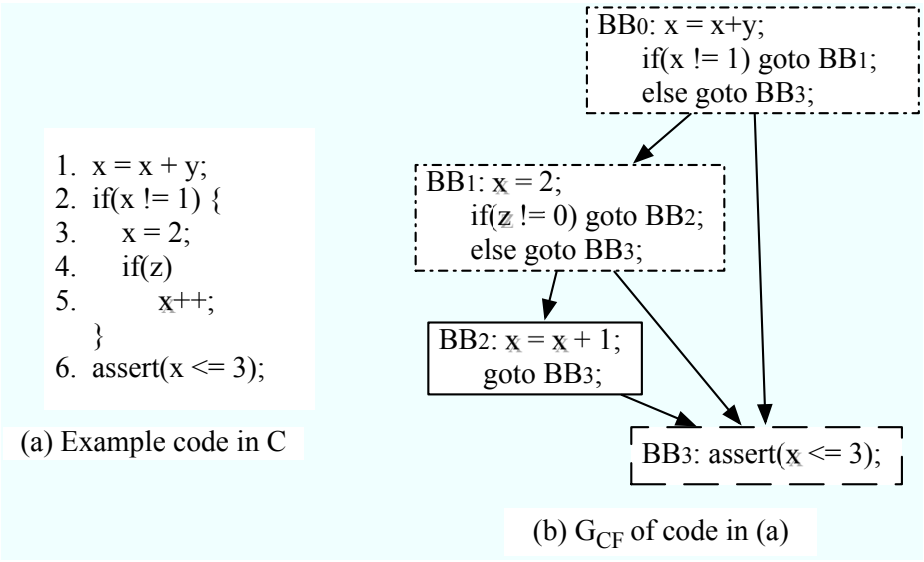


Figure 2.1: An example of G_{CF}

Figure 2.1 shows a G_{CF} example, where each node in it represents a labeled BB. The nodes bounded by the dot-dash border lines (e.g., BB_0 and BB_1) are Branches; for those bounded by the dashed border lines (e.g., BB_3) are Merges. All nodes are connected by the directed edges, which represent the control flow relationships among the corresponding BBs. The guards of BB_0, BB_1 and BB_2 are respectively $g(BB_0) = TRUE$, which means BB_0 is always executed; $g(BB_1) = (x! = 1)$, which is actually the conditional predicate in line 2; $g(BB_2) = ((x! = 1) \wedge (z! = 0))$, which is the conjunction of conditional predicates in line 2 and line 4.

Definition 6. *(SSA form).* A Static Single Assignment (SSA) form [39] is an intermediate representation of a program, in which every variable is replicated in different versions, and each variable version is statically defined exactly once. (One **static** definition may be in a loop resulting in the variable being dynamically defined many times during execution.)

As an intermediate representation of the program widely used in the modern compiler designs, the SSA form enhances various compiler optimizations such as dead code elimination and partial redundancy elimination. This is mainly because the use-definition chain of each variable becomes explicit in this form. Figure 2.2 presents a small example code and its SSA form. In Figure 2.2(b), to find all uses

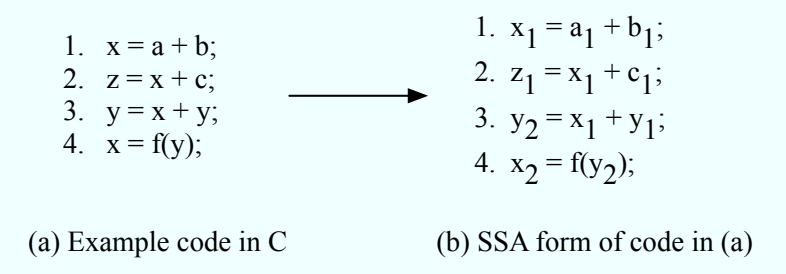


Figure 2.2: An example of SSA form

of the variable x defined at line 1 before it is re-defined, we simply need to identify the occurrences of the variable version x_1, which are in line 2 and 3. Furthermore, since the value of each variable version never changes (not considering those in loops), reasoning about variables in this form is significantly simplified compared to that in the original program. This feature also greatly facilitates the verification task that we will explain in later chapters.

Definition 7. *(Phi function).* The basic operation of a *Phi* function of x formulated as $\phi(x_1,\ldots,x_n)$, is to select one of all reachable definition versions $(x_1,\ldots,x_n)$ of x at the location of the *Phi* function.

Before Merge nodes in an SSA form, special statements using *Phi* functions are added to define new versions of variables, for example the statement $x_{n+1} = \phi(x_1,\ldots,x_n)$ defining a new version x_{n+1} of a variable x.

Definition 8. *(SSA-transformed CFG).* An SSA-transformed Control Flow Graph G_{CF}^{SSA} is a G_{CF} augmented with *Phi* function nodes in front of Merges. Each of these nodes includes one or more special statements using *Phi* functions, if more than one definition version of the variables can reach the node. In addition, all variables in each BB are in the SSA form.

In Figure 2.3, the subscripts distinguish different variable versions in the SSA form. In Figure 2.3(a), the version x_5 is defined by a *Phi* function parameterized with all three reachable versions x_i $(i \in I = \{2,3,4\})$ at that location, which are listed by their definition order. The number in the parentheses beside each of three versions indicates the index of the BB, in which the version is defined. Each x_i has a different definition condition (Def-C). Its definition is executed only if its Def-C is satisfied. x_5 is assigned to the last version x_k ($k \in I$) whose Def-C is $TRUE$ during the actual execution. Correspondingly, the *Phi* node with the dotted border line, which includes this *Phi* function is added before the Merge

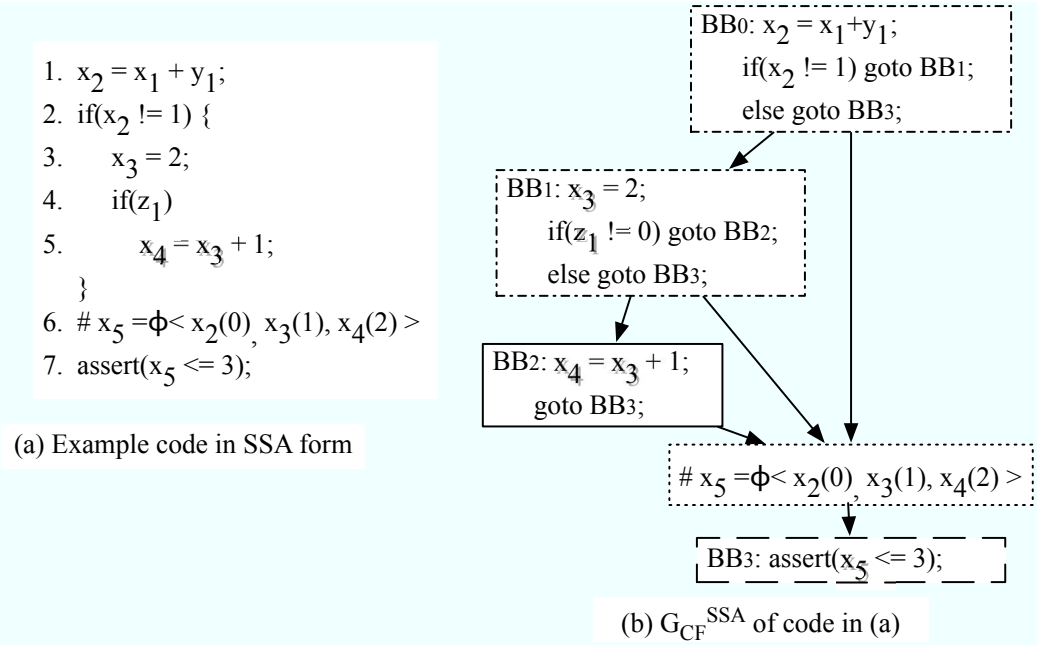


Figure 2.3: An example of G_{CF}^{SSA}

node BB_3 in the G_{CF}^{SSA} of Figure 2.3(b).

2.2 Satisfiability Problem

The satisfiability problem (SAT) is one of the most studied NP-Complete problems because of its significance in theoretical computer science and practical applications. In this section, we begin with some basic definitions related to SAT, before diving into a discussion of solving this problem.

Definition 9. *(First-order logic).* First-order logic [62, 68, 77] is defined by a formal mathematical language, whose basic elements include:

1. Logical symbols:
   - Parentheses: (,)
   - Quantifiers: $\forall$ (for all), $\exists$ (there exists)
   - Boolean connectives: $\wedge$ (and), $\vee$ (or),$\neg$ (not)

- Constant Booleans: $TRUE$, *FALSE*
- Equality: $=$

2. Non-logic symbols:
   - Functions symbols: a function symbol has an associated arity, a non-negative integer that indicates the number of arguments required by the function.
   - Constants: a special case of function whose arity is zero.
   - Relation symbols (Predicates): $\leq, \geq, <, >$, etc.
3. Variables

Terms and formulas consisting of these elements can be constructed by following some fixed, *well-formedness* rules, which are referred to [62, 68].

Definition 10. *(term).* A term is recursively defined as

- a variable
- application of function symbol over terms

Definition 11. *(formula).* A formula is recursively defined as

- $TRUE$,*FALSE*
- equality between terms or application of relation symbol to terms
- if θ is a formula, so is $\neg\theta$
- if θ and ϕ are formulas, so are $\theta \wedge \phi$ and $\theta \vee \phi$

Definition 12. *(atomic formula).* It is a formula constructed by applying a relation symbol over terms. The formula $TRUE$ and *FALSE* are also atomic.

Definition 13. *(literal).* A literal is an atomic formula in either its positive or negated form.

Definition 14. *(quantifier-free formula).* It is a formula constructed by only literals and Boolean connectives.

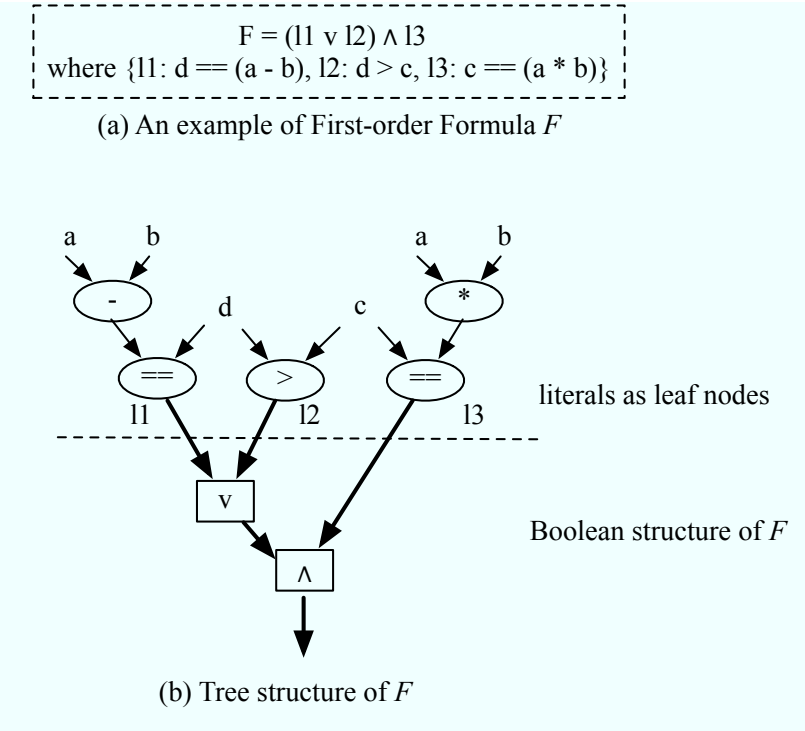


Figure 2.4: An example of FOL formula

The formulas used in this dissertation are the quantifier-free first-order formulas. Every such formula can be represented as a tree structure, whose leaves are literals, and whose internal nodes are the Boolean connectives. An example quantifier-free formula F is given in Figure 2.4. F consists of three literals and two Boolean connectives as shown in Figure 2.4(a). The three literals are the leaves of the tree structure of F as in Figure 2.4(b).

Definition 15. *(assignment).* For a formula F, an assignment of F from some domain D is a value mapping F's variables to elements in D.

Definition 16. *(satisfiability).* A formula is satisfiable (SAT) if there exists an assignment under which the formula evaluates to $TRUE$. If no such assignment exists, the formula is unsatisfiable (UNSAT).

Definition 17. *(validity).* A formula is valid if it evaluates to $TRUE$ under all assignments.

Definition 18. *(decision problem).* It is a question with yes-or-no answer, which is the validity of a given first-order formula of theory T.

Definition 19. *(decision procedure).* A function for solving the decision problem with respect to every formula of theory T is defined as a decision procedure for the theory T.

Technically speaking, several theories can be considered as first-order logic theories [77], such as propositional logic theory, equality theory, theory of bit-vectors, theory of arrays, etc. In the following two subsections, we mainly introduce the SAT problem in the theory of propositional logic and fixed-width bit-vector arithmetic, which are mostly related in our work.

2.2.1 Propositional Logic

Propositional logic is the basis of automated reasoning. It is widely used in various fields that include planning problems in artificial intelligence, circuit design and verification, etc.

Definition 20. *(Propositional formula).* The formula in propositional logic is also called the Boolean formula, which is defined by the following grammars:

- formula : formula $\wedge$ formula | formula $\vee$ formula | $\neg$ formula | (formula) | atom
- atom: Boolean variable | TRUE | FALSE

Definition 21. *(Conjunction).* Conjunction is a AND logic operation between literals (or atoms) that results in a value of $TRUE$ if and only if all the literals (or atoms) in the conjunction are $TRUE$.

Definition 22. *(Disjunction).* Disjunction is a OR logic operation between literals (or atoms) that results in a value of $TRUE$ if and only if any literal (or atom) in the disjunction is $TRUE$.

Definition 23. *(CNF).* Conjunctive Normal Form (CNF) is a conjunction of clauses, where each clause is a disjunction of literals.

Here is an example of a Boolean formula in the CNF format: $f = (\bar{a} \vee b) \wedge (a \vee b \vee c \vee \bar{d}) \wedge (b \vee c \vee d)$. In this simple formula, f is satisfiable. One possible satisfying assignment is: $a = TRUE, b = FALSE, c = TRUE, d = FALSE$. Various verification problems are represented as or transformed to the circuits that are built up with logic gates. The circuits must be translated to the CNF formula first, in order to utilize SAT solvers for verification. SAT solvers are the tools for solving the SAT problem in the propositional logic.

Figure 2.5 gives the CNF translation of four common logic gates.

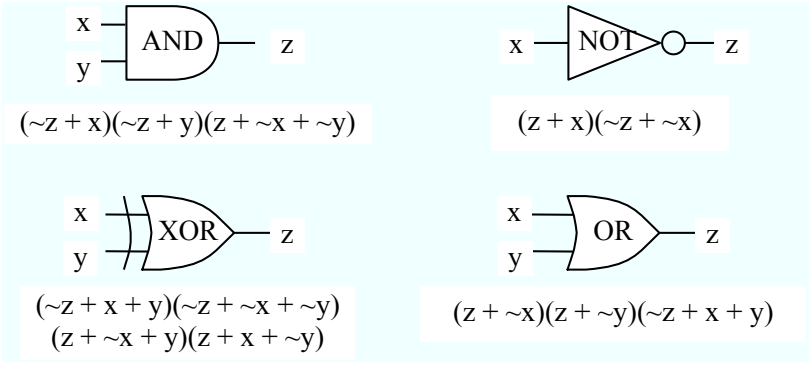


Figure 2.5: CNF translation of basic gates.

One popular approach to systematically solve a SAT problem in propositional logic is based on a branch-and-bound search algorithm called the Davis Putnam Logemann Loveland (DPLL) algorithm [41]. DPLL is the basis for most state-of-the-art SAT solvers. It is a complete, backtracking-based algorithm, whose memory cost is linear to the number of Boolean variables in the formula, but the solving time can still be exponential. The basic DPLL algorithm is composed of three major steps:

1. Decide the branching variable and assign a value (either $TRUE$ or *FALSE*) to it.
2. Propagate the impact of the decision variable being assigned by invoking more implied assignments. For instance, in the previous example formula f, after a is decided to be $TRUE$, b is implied *FALSE* by the first clause.
3. If a conflict occurs in which at least one clause has all its literals assigned *FALSE*, backtrack to undo the decisions made so far to resolve the conflict.

This algorithm was augmented with the conflict-driven-learning proposed in [83] to prune the search space. The basic idea of the conflict-driven learning is to record the variable value combination in the decision history, which is responsible for the conflict. A *conflict clause* is added to the formula to avoid the unnecessary value assignments that will definitely cause a similar conflict in the future. Figure 2.6 gives the pseudo code of the DPLL algorithm with the conflict-driven learning.

In the following, we briefly introduce some basic concepts about the unsatisfiable core extraction from an unsatisfiable formula, since we use this core in our verification work.

```
1.  while(True) do
        /* choose a branching variable */
2.    if (Decide_Next_Branch_Variable()) then
          /* propagate the constraints of the chosen branching variable*/
3.      while (Deduce() == Conflict) do
            /* learn conflict reason and return a backtracking level to resolve conflict*/
4.          level = Conflict_Analysis();
5.          if (level == 0) then /* show conflict exists even without any branching*/
6.              return UNSATISFIABLE;
7.          else
8.              Back_Track(level);   /* Backtrack to resolve conflict */
9.          end if
10.     end while
        /* no branching variable that all variables got assigned */
11.   else
12.     return SATISFIABLE;
13.   end if
14. end while
```

Figure 2.6: DPLL algorithm with learning.

Definition 24. *(resolution rule).* The resolution rule in the Boolean logic takes two clauses containing complementary literals (i.e., x and $\bar{x}$), and produces a new clause with all literals from both except for the complementary one. The clause produced by the resolution rule is called the resolvent of the two clauses.

Definition 25. *(unsat proof).* A proof of unsatisfiability P for a set of clauses C is a directed acyclic graph (V_P, E_P), where each node $v \in V_P$ represents a clause. Each node in P can be classified into one of the following three categories:

1. The Root nodes (without predecessors), which are clauses belonging to the set C;
2. The unique Leaf node, which is the empty clause;
3. The internal node v, which has exactly two predecessors v_1 and v_2 such that v is the resolvent of v_1 and v_2.

A simple example of deriving an empty clause is given in Figure 2.7. Many DPLL-style SAT solvers have been extended to provide this proof, such as [89, 90].

Given an unsatisfiable CNF formula $F = C_1 \wedge C_2 \wedge \ldots \wedge C_n$, there exists a subset of clauses $\lambda \subseteq \{C_i \mid i =$

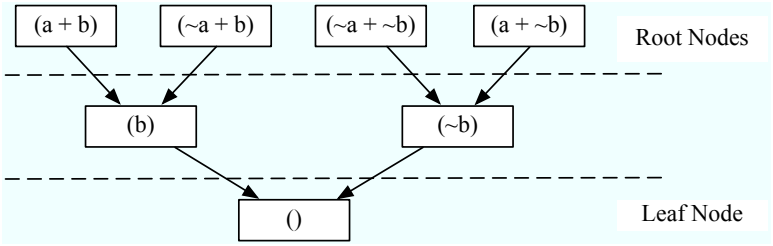


Figure 2.7: An example of unsatisfiability proof by resolution

$1 \ldots n\}$ such that a formula F_c formed by the conjunction of the clauses in λ is also unsatisfiable. We call F_c an *unsatisfiable core* of the original formula. F_c may contain a much smaller number of clauses than the original formula but it may not be unique. All root nodes used in the proof of unsatisfiability actually form an unsatisfiable core. A number of efficient procedures have been proposed to extract a small unsatisfiable core from the unsatisfiability proof provided by a SAT-solver [14, 118].

2.2.2 Bit-vector Arithmetic

In the last few years, the solvers for Satisfiability Modulo Theories (SMT) [80] have experienced the amazingly fast applications in formal verification, compiler optimization and scheduling, program analysis, etc. SMT solvers are constructed to determine the satisfiability of first-order formulas with respect to decidable background theories, using a combination of SAT solving and theory-specific decision procedures. Here, we only discuss the SAT problem related to the theory of fixed-width bit vectors. The SAT problem for the theory of arbitrary-width bit vectors is undecidable and is beyond the scope of this dissertation.

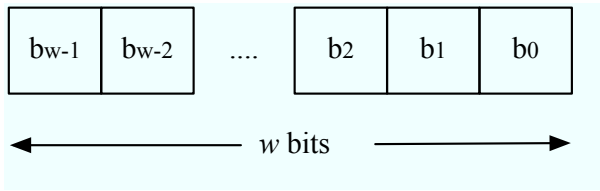


Figure 2.8: An example of bit-vector with w bits. The i^{th} bit is denoted as b_i.

Definition 26. *(bit vector).* A bit vector B as shown in Figure 2.8 is a vector of bits with a given modeling width w:

$$B : \{b_0, .., b_i, .., b_{w-1}\} \quad \textit{where } 0 \leq i < w,\ b_i \in \{0,1\}$$

The *type* associated with a bit vector B in the bit-vector arithmetic formula depends on two main factors:

1. Modeling width w of B;
2. Whether B is signed or not.

Definition 27. *(binary encoding).* Let x be an unsigned integer, and B be a bit vector with modeling width w. We say that B is the binary encoding of x ($0 \leq x < 2^w$) if and only if

$$x = \sum_{i=0}^{w-1} b_i \times 2^i$$

Bit b_0 is the *least significant* bit, and bit b_{w-1} is the *most significant* bit.

Definition 28. *(two's complement encoding).* Let x be a natural number (signed integer) and B be a bit vector with modeling width w. We say that B is the two's complement encoding of x ($-2^{w-1} \leq x < 2^{w-1}$) if and only if

$$x = -2^{w-1} \times b_{w-1} + \sum_{i=0}^{w-2} b_i \times 2^i$$

Bit b_{w-1} is the *sign* bit.

Definition 29. *(bit-vector arithmetic formula).* The bit-vector arithmetic formula is one kind of first-order logic formula [77] which is defined based on the following elements:

1. Constants: e.g., $\langle 0b00011101 \rangle_8$;
2. Bit vector variables with fixed width: implicit restriction to finite domain.
3. Function symbols:
   - Type cast operators: Concatenation, Extraction, Extension
   - Arithmetic operators: $+, -, \times$; $\div, \%$ (signed or unsigned)
   - Bitwise operators: $\wedge$ (and), $\vee$ (or),$\neg$(not), $\otimes$(xor)

- Shifting operators: $<<, >>$ *a* (arithmetic shift right), $>>$ *l* (logic shift right),

4. Predicate symbols: $=, \neq$; $<, >, \leq, \geq$ (signed or unsigned)

5. If-then-else operator: *ite*

Here, we introduce four general categories of decision procedures (DP) for solving the SAT problem in the bit-vector arithmetic, which are widely used in the software formal verification [48]:

1. SAT translation-based decision procedures (Bit-blasting) and its variants

   Through the SAT translation or bit-blasting, a quantifier-free formula in the theory of fixed-width bit vectors can be equivalently transformed into a quantifier-free formula over Boolean variables. Many existing DPs (E.g., Cogent [37], CVC-Lite [47], STP [25], Yices [116]), share this basic idea to reduce the input SAT problem over bit-vectors to the SAT problem over Boolean variables, then apply a SAT solver to solve the resultant Boolean SAT instance. This method is popular mainly due to the ever-increasing efficiency of SAT solvers. However, the naive translation of all bit-vector operators to SAT usually can not make use of the inherent structural information of the input formula. So, most DP tools in this category apply the pre-processing step before bit-blasting to SAT to improve the performance. Some effective pre-processing methods use solvers for linear arithmetic, algebraic transformations and simplifications, etc. For example, CVC-Lite [47] includes a normalization step followed by equality rewriting to pre-process the input formula; STP [25] pre-processes the bit-vector formulae using several array optimizations, as well as arithmetic and Boolean simplifications.

2. Shostak-style based procedures

   The DP for a particular theory T using the Shostak-style approach needs to have a canonizer and solver for T, so as to be soundly and completely combined with other DPs also using this Shostak-style [99]. More information about canonizer and solver can be found in [102]. Stanford Validity Checker or SVC [10] belongs to this category. But so far, the DPs developed with this method are restricted to a subset of bit-vector arithmetic with concatenation, extraction and bitwise Boolean operations. They also have not shown as competitive as the translation to SAT based DPs or abstraction-refinement based methods.

3. Procedures for modular or bounded arithmetic

   A large amount of work has been done for modular arithmetic to decide both linear and non-linear bit-vector arithmetic. Since the methods developed in the decision procedures in this category vary a great deal, we refer the reader to [6, 16, 67, 95] for obtaining more details about individual methods.

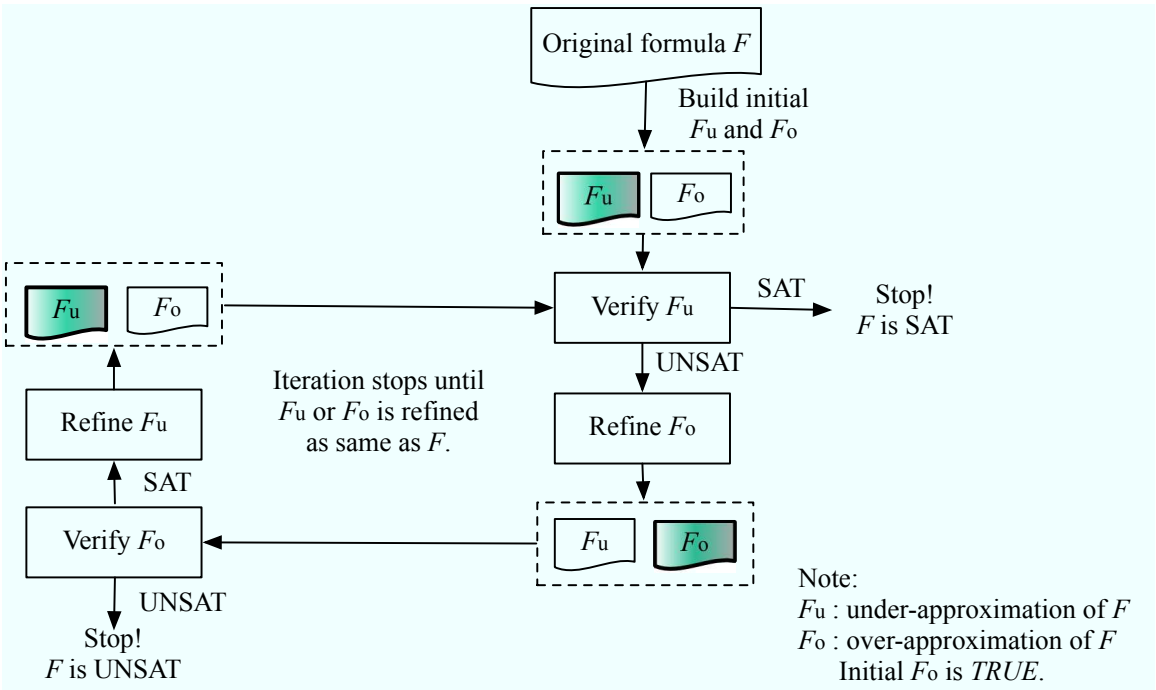


Figure 2.9: Overview of abstraction-refinement based decision procedure.

4. Abstraction-refinement based procedures.

   Recently, the abstraction-refinement paradigm was applied to solve the SAT problem of bit-vector arithmetic [20, 58, 59]. The basic idea is to alternatively compute an over- and under- approximation of the formula, until the SAT problem is solved, as illustrated in Figure 2.9. The over-approximation F_o owns a *superset* of behaviors of those in the original formula F, while the under-approximation F_u has a *subset* of behaviors compared to the original one. At the start, we build the initial F_u and F_o, where F_o can be initialized as $TRUE$. The iterative refinement procedure has four main operations. It first verifies F_u. If F_u is SAT, it can conclude that F is also SAT, and it can stop. Otherwise, F_u is refined via learning the UNSAT proof of F_o. Then, it verifies F_o. If F_o is UNSAT, it can conclude that F is also UNSAT and stop. Otherwise, F_u is refined via

learning the counterexample of F_o. Then, it goes back the first operation of verifying the refined F_u. The iteration definitely terminates when either F_o or F_u is refined as the same as F. A more detailed discussion of over-approximation and under-approximation will be presented in Section 2.4. It has shown efficiency when the approximations are easier to be solved than the original formula, and the number of iterations to compute the appropriate approximations is small.

Our techniques, which are proposed in Chapter 5 and 6 for solving the bit-vector arithmetic formula instances derived from the bit-precise verification of embedded software, are in the fourth category.

2.3 Software Model Checking

Model checking is a formal technique for automatically verifying the properties of finite state systems [33]. Given a specification and a model with finite states defined by the design, model checking systematically traverses the entire state space to completely verify whether the specification holds or not. If so, model checking proves the correctness of the design with respect to the specification; otherwise, it returns a counterexample to show an erroneous trace that violates the specification. Modeling, specification and verification are three major components in model checking. We will briefly introduce them in the following.

Generally speaking, the design is modeled as a *Finite State Machine* (FSM) that can be graphically represented as the State Transition Graph (STG).

Definition 30. A FSM is a six tuple $(I, S, \delta, S_0, O, \lambda)$, where

- I : a set of inputs.
- S : a finite set of states.
- $\delta : S \times I \rightarrow S$ (the next state function).
- S_0 : a set of initial states.
- O : a set of outputs.

- λ : the output function.

    1. $S \times I \rightarrow O$: Mealy machine that is widely used to represent sequential circuits.
    2. $S \rightarrow O$: Moore machine.

The specification is usually expressed in temporal logic, which is a formalism for describing sequences of transitions between states. Here, we introduce one kind of powerful temporal logic called Computation Tree Logic (CTL), which consists of two path quantifiers: *A* ("for all computation paths") and *E* ("for some computation path"), and five basic temporal operators:

1. X φ ("next time"): Property φ must hold in the second state of the path.
2. F φ ("in the future" or "eventually"): Property φ must hold at some time on the path.
3. G φ: ("always" or "globally"): Property φ must hold at every state on the path.
4. φ U ψ: ("until"): Property ψ holds at the current or a future state, and property φ must hold until that state. the path.
5. φ G ψ: ("release"): Property ψ is true until the first state in which φ is true.

With CTL or its variants, we can assert how the behavior of the design evolves over time. Safety is a class of properties widely used in the specification, which says some error f will never happen at any (time) instance ($AG \neg f$ in CTL). It is also the class of properties that we mainly focus on in our work.

A model checking example from the wireless connection system is given in Figure 2.10. The FSM of this example system has 5 states derived from three atomic propositions: scanned, classified and connected. In order to verify whether a property φ described in the temporal logic holds or not, a model checker explores the FSM to check if all states satisfy φ. For instance, a property $\varphi = AG(p = (connected => classified))$ claims that, all states in the design satisfy that if a connection is set up, the connection must use a classified waveform. The simple verification process of φ with 4 steps is given in the block under the circle of "model checker" in Figure 2.10. Since $AG(p) = \neg EF(\neg p)$, the verification problem is first transformed to check the FSM to see if there exists a path where a state satisfying $f = \neg p = (connected \wedge \neg classified)$ can eventually be reached. After traversing every state

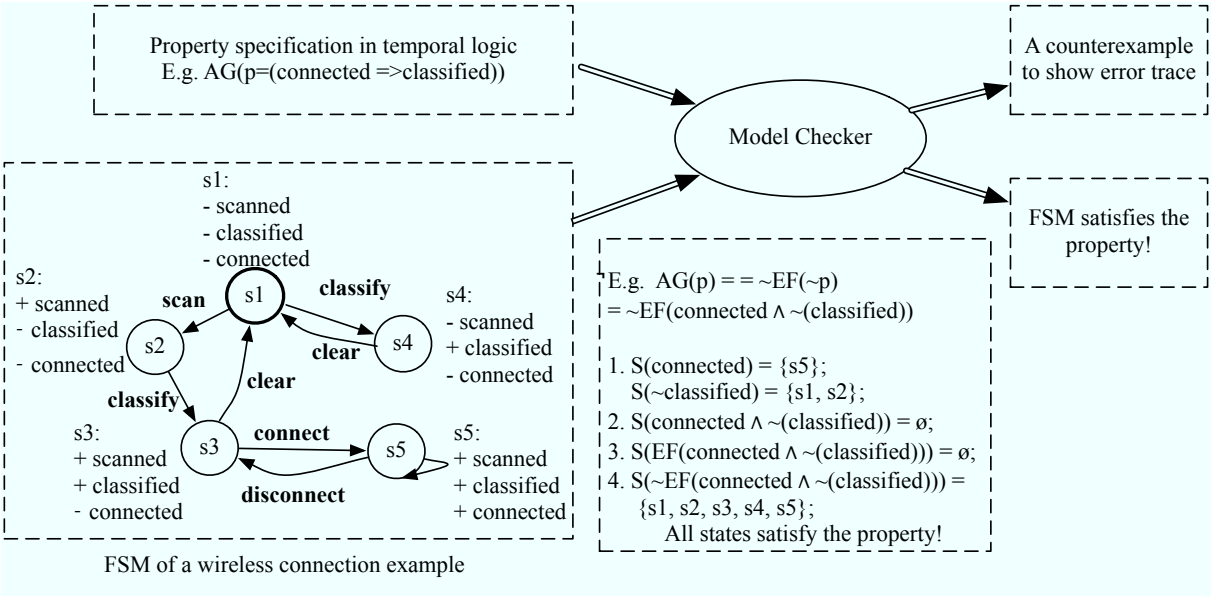


Figure 2.10: An example of model checking

in the FSM, the model checker finds that the set of states satisfying such query is empty, which means all states satisfy *p*. So, it can conclude that this FSM satisfies the property φ. For other verification instances of obtaining a non-empty set of states that dissatisfy the property, i.e., satisfy the negation of the property, the model checker can automatically analyze these states and the traversal history to return a counterexample that shows the error trace for debugging.

Due to the state explosion problem, i.e., the number of states of a design is exponential in size of the design description (i.e., the number of state variables), the explicit model checking that directly traverses every state in the STG of the design, is unable to handle large instances in practice. With the discovery of the concise representation of transition relations in FSM using *Ordered Binary Decision Diagrams (OBDDs)*, the capability of model checking was dramatically enhanced by symbolic model checking (SMC) [24, 85] based on this representation. Instead of explicitly traversing the STG, SMC implicitly traverses the FSM via OBDDs, where the number of nodes no longer depends on the actual number of states in FSM. With this breakthrough, some real-world designs with more than 10^{20} states have been successfully verified [22, 23]. However, for large design systems, SMC is still costly in terms of memory since the OBDDs built for model checking can still be very large. Selection of the right ordering of BDD variables is very important, but the generation of a variable ordering with small BDD

size is a time consuming procedure, and may need manual intervention.

With the recent dramatic advances in SAT-solver, Bounded Model Checking (BMC) is becoming increasingly popular [11, 119]. The basic idea is to use a SAT-solver to check the states within a bounded length of K transitions from a given initial state(s). When K reaches the diameter of FSM, BMC can prove the property completely. BMC has two main advantages. First, BMC can find the counterexample with the shortest path very fast. Second, it uses much less space than OBDD based SMC, and does not need manual variable ordering.

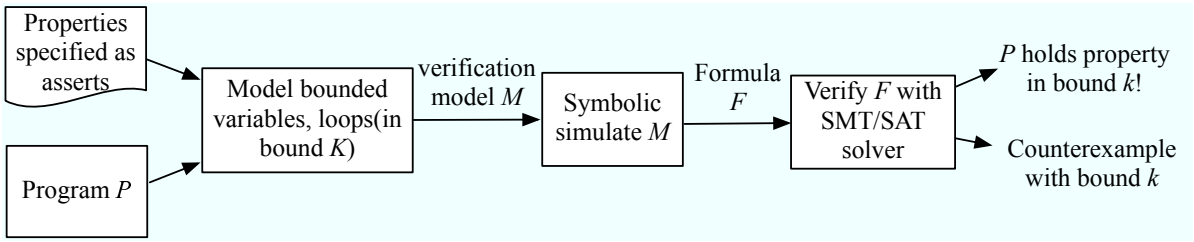


Figure 2.11: Basic steps of software bounded model checking

2.3.1 Software Bounded Model Checking

A typical SAT-based bounded model checking (BMC) formulation for software programs has the steps as shown in Figure 2.11.

1. Given a program P (assuming that all function calls have been inlined), and the properties specified as assertions in P, we build the bounded verification model M by unrolling the loop structures or recursions in bound K and modeling the data variables as bit-vectors with fixed width. The assertion is a special program statement where the value of the asserted expression must always be true; otherwise, the execution aborts. Figure 2.12 gives an example assertion, which claims the array index x must fall within the lower and upper bounds of the array *arr*.

   Figure 2.13 gives a simple example of bounded modeling the execution sequence of loop structures and program variables. In Figure 2.13(a), the code block A includes all statements in the loop. The maximum number of iterations of the *for* loop is five. In the bounded modeling shown in Figure 2.13(b), code block A is repeated five times, and every program variable is modeled as

```
void foo() {
  int x;
  char arr[4];

  assert((x+1) >= 0&&(x+1)<4);
  arr[x+1] = 'k';
}
```

Figure 2.12: An example of assertion

a vector of 32 bits. Any constant is given in its binary representation; for example, the constant integer 0 is modeled as a bit-vector [0..0000] with 32 bits.

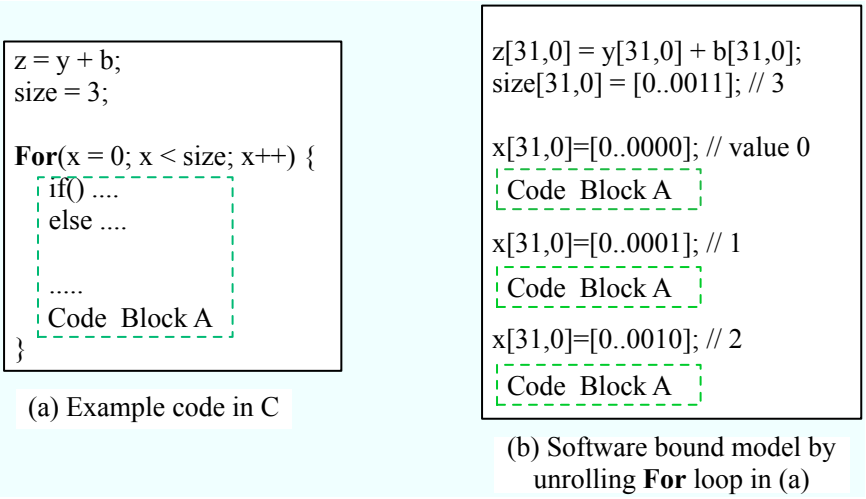


Figure 2.13: An example of software bounded model.

Modeling program variables as bounded bit-vectors rather than unbounded variables with infinite ranges (as in the pure mathematical sense) conforms to the actual computation; for example, the integer data type is set to 32 bits wide in most existing embedded computers.

2. All bounded program constraints, including the asserted property, are encoded as a first-order bit-vector arithmetic formula F.

3. The satisfiability of the formula F, which corresponds to the discovery of a counter-example that can violate the property, is then decided by a SMT/SAT solver. If F is UNSAT, we can conclude that the property assertion can never fail within a bound of K. If the depth of un-winding reaches

the finite upper bound of the loop iterations, we are able to prove the program property completely. On the other hand, if F is satisfiable, a satisfying solution is produced that can be mapped to the word-level to construct an error trace in the program.

2.4 Automatic Abstraction Techniques

Generally speaking, software verification models have very large numbers of states, which make the state-explosion problem of model checking much worse than in hardware design systems. Property-based automatic abstraction becomes necessary in the scalable hardware design and software formal verification [28, 40, 53, 78, 84, 120]. Abstraction reduces the verification effort by building a small abstract model only with the information relative to the property in the design and removing the irrelevant information. Furthermore, an abstract model sometimes will not keep all the relevant portions of the design, thereby aggressively pruning away much of the state space. In contrast with such a model, we call the original design under verification the concrete model. We also name the counterexample found in the abstract model and concrete model as the *abstract counterexample* and *concrete counterexample* respectively. To automate the abstraction process, a refinement process is usually conducted iteratively. Refinement is used to learn and restore any relevant information that was removed in a previous abstraction step, until the model is precise enough for the verification task.

According to the verification capability that the abstract model can render, we distinguish three important abstraction approaches for model checking, especially safety properties [69, 104]:

- Over-approximation based abstraction (Existential Abstraction): render the correct verification of *proving* the property. It may produce *false negative* errors, but not *false positive* errors.
- Under-approximation based abstraction (Universal Abstraction): render the correct verification of *refuting* the property. It may produce *false positive* errors, but not *false negative* errors.
- Ternary abstraction: render the sound verification of both proving and refuting the property, depending on the usage.

A false negative error f_n is an error of rejecting a design while it is actually correct. A false positive

error f_p is an error of failing to reject a design while it is actually false. In the case of conducting verification based on the abstraction, f_n occurs when the property holds in the concrete design, but a spurious counterexample violating the property can be found in the abstract model; f_p occurs when the property is violated in the concrete design, but no abstract counterexample can be found. In the following subsections, we will introduce several realistic techniques belonging to these three kinds of abstraction approaches and two systematic refinement methods (counterexample guided and proof-based refinement).

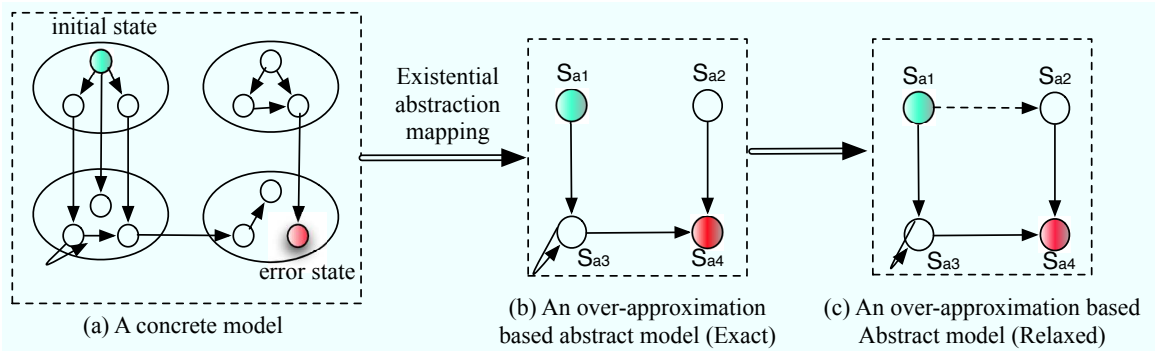


Figure 2.14: Over-approximation based abstraction

2.4.1 Over-approximation based Abstraction

In an exact over-approximation based abstract model, an abstract transition is made from an abstract state if there exists a transition from *at least* one corresponding concrete state. So, this abstraction is also called the *existential* abstraction. The example in Figure 2.14 illustrates constructing such an abstract model from a concrete one. All concrete states in each big circle of Figure 2.14(a) are mapped to a new abstract state in Figure 2.14(b). In the abstract model, the error state S_{a4} is reachable from the initial state S_{a1} through the abstract transitions $\{(S_{a1}, S_{a3}), (S_{a3}, S_{a4})\}$. But the error state in the concrete design is actually not reachable, so this trace of abstract transitions is spurious and incurs the false negative error. Since the precise computation of abstract transitions is expensive, most realistic abstract techniques relax the condition of making an abstract transition to reduce the computation cost, and, as a result, may add extra transitions, as in Figure 2.14(c). The abstract transition $\{(S_{a1}, S_{a2})\}$ is made, although there does not exist a transition from the corresponding concrete states. Predicate abstraction is an important example of such techniques.

Weakening abstraction represents a special class of over-approximation based abstraction techniques. The model with such abstraction can generate more behaviors with respect to the concrete one, either by directly removing the input computation constraints of some concrete variables, or by simply replacing the concrete transitions of the original model with the weakened ones in the same data domain. The distinguishing point is that no state space mapping is involved during this kind of abstraction. In a word, the over-approximation based abstraction techniques add additional behaviors (or transitions) into the abstract model that the concrete model does not have. In the following, we present the predicate abstraction technique and weakening abstraction with more details.

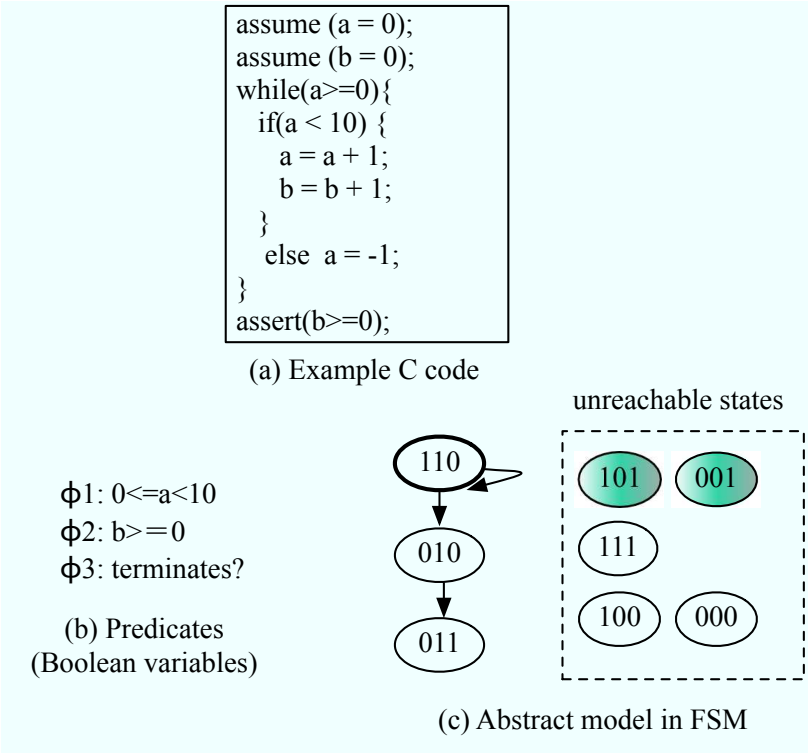


Figure 2.15: An example of predicate abstraction of program in C.

Predicate Abstraction

Predicate abstraction [40, 53] is a powerful abstraction technique that can transform an infinite state system design to a finite state model. This makes model checking applicable to some infinite state concurrent system designs. Its basic idea is to construct the abstraction by tracking only a small set

of predicates over the variables in the original concrete design. Each predicate is represented by a Boolean variable. In this way, a very large or even infinite state space of the concrete design can be mapped to a new, small state space, but all behaviors of the concrete design are still preserved. Predicate abstraction has been widely used in software model checking [8,26,36,60]. Figure 2.15 gives a predicate abstraction example of a simple C code. Three predicates (Boolean variables), $\{\phi_1, \phi_2, \phi_3\}$ as given in Figure 2.15(b), are used to track the computation on two integer variables in the code, whose number of states can be 2^{64} (assume that each integer has 32 bits). The abstract model is represented as a FSM, as in Figure 2.15(c). The initial abstract state corresponding to the first line of the code is $\{110\}$; it is true until the *while* loop execution finishes. The next abstract state $\{010\}$ is mapped from the concrete state when the code execution jumps out of the loop, since the value of a becomes -1 and the predicate ϕ_1 becomes $false$. Then the abstract state moves to $\{011\}$, as the predicate ϕ_3 becomes $TRUE$ when the execution terminates at the assertion statement. The two abstract states $\{101, 001\}$ filled with color represent the erroneous states with respect to the asserted property in the code. Since, all the reachable states, beginning with the initial state, are not erroneous, the assertion can be verified as valid.

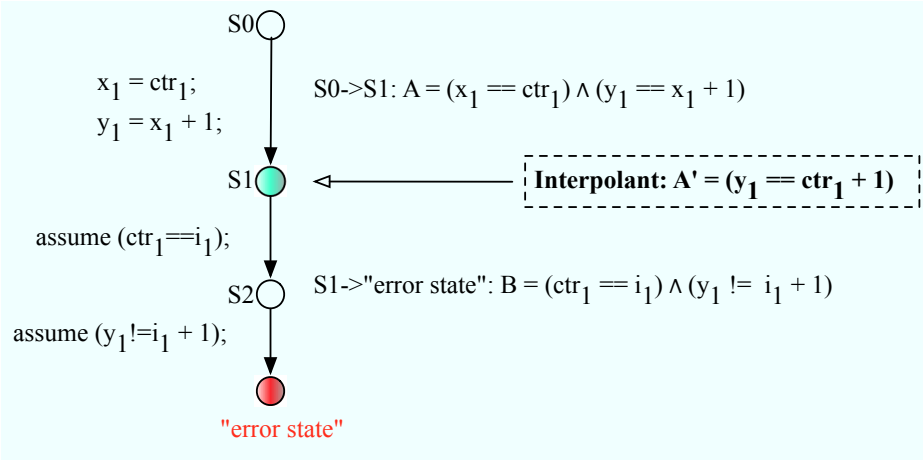


Figure 2.16: An example of interpolant.

The number and the quality of predicates closely impact the complexity and precision of the abstract model. The method of extracting predicates from interpolants [86], has been widely used in the predicates discovery/selection for the abstraction refinement, which is a critical step in the predicate abstraction. Given a pair of formulas (A, B) such that $A \wedge B$ is UNSAT, an interpolant [38] for (A, B) is a formula A' with three properties: (1) A implies A', (2) $A' \wedge B$ is UNSAT, and (3) A' refers only to

the common symbols of A and B. Figure 2.16 gives an example of an interpolant. Assume the execution trace from state $S0$ to "*error state*" presented on the left is obtained by analyzing the spurious abstract counterexample. The two "*assume*" statements placed at state $S1$ to state $S2$, and state $S2$ to "*error state*" respectively, mean that each transition between the states must satisfy the condition claimed in the corresponding "*assume*" statement. Formula A represents the constraints between state $S0$ to state $S1$; formula B corresponds the constraints between state $S1$ to "*error state*". Since $A \wedge B$ is UNSAT, the trace leading to the error state in Figure 2.16 is actually not feasible. We could obtain an interpolant A' at the state $S1$, such that $A' \wedge B$ is still UNSAT. In the counterexample guided abstraction refinement which we will introduce later, such interpolants are usually used as new predicates for refining the abstract model, so as to avoid generating such spurious abstract counterexamples again. This interpolation based predicate selection approach has one major advantage: at each program location, it uses only predicates that are relevant to that location. This can achieve a reduction in the number of abstract states, so as to increase the verification performance [61]. However, the operations of predicate discovery and abstract states mapping generally require a great deal of computation resources. So, although it is powerful, predicate abstraction is an expensive abstraction technique.

Weakening Abstraction

Weakening abstraction [84] represents a class of over-approximation based abstraction techniques, which weakens the transitions of the concrete model without mapping to a new data domain so that both the abstraction and the concrete designs are in the same domain. This is different from predicate abstraction, which involves the state space mapping between different domains; for instance, in predicate abstraction, variables in the Integer domain may be mapped into variables in the Boolean domain via a set of predicates.

Localization abstraction [78] is a weakening abstraction technique that weakens the transition relation by completely removing the input constraints of a set of non-input variables, and considers these variables as pseudo primary inputs. In other words, all variables removed are *existentially* quantified as inputs. This is shown in Figure 2.17: the removed variables, including their connections (drawn in pale black), are invisible in the abstract model; the variables (represented by the squares filled with color) at the boundary between visible and invisible variables become pseudo inputs. As a result, the abstract

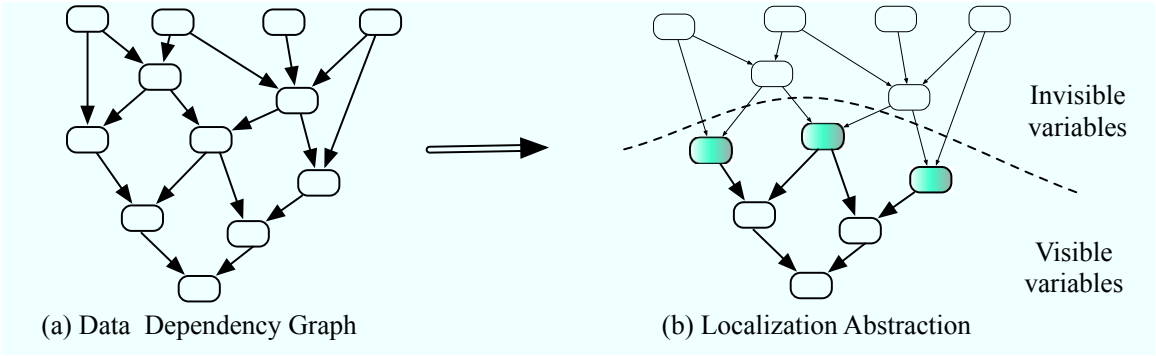


Figure 2.17: Over-approximation based abstraction without state space mapping

model contains only a subset of computation constraints in the concrete model represented by the visible variables.

Figure 2.18 gives a simple localization abstraction example of a small C code segment. Figure 2.18(b) gives a weakening abstraction ω of the code in Figure 2.18(a) by removing the program constraints in Line 1 and 2. The asserted property "$assert(z >= y)$" fails in ω since ω removes the concrete constraint between y and z and causes the false negative error. After restoring the constraints in Line 2, we get a refined abstraction ω′ in Figure 2.18(c). The property holds in ω′, so it also holds in the original program.

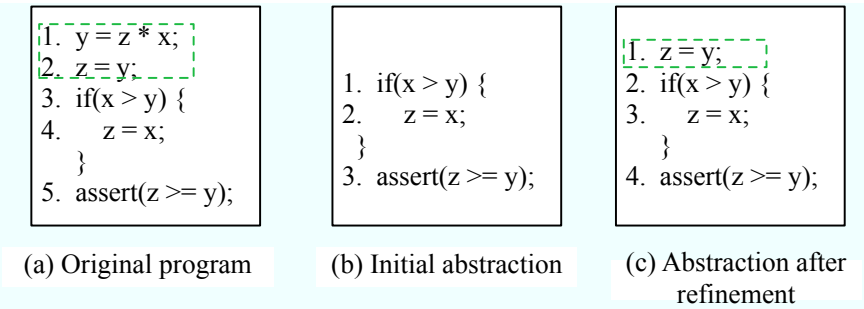


Figure 2.18: An example of localization abstraction of program in C

Un-interpreted function (UF) abstraction [21] is another weakening abstraction technique widely used in the high-level hardware design and software verification. It is mainly applied to abstract the arithmetic computations in the data-path. UF operator removes the constraints between the output and inputs of the individual arithmetic operator. But the constraint of functional consistency is enforced, which is

symbolically defined as follows:

$$(x_1 = y_1)\&\ldots\&(x_n = y_n) \longrightarrow f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n)$$

In practice, more arithmetic constraints between the inputs and outputs of the specific UF computation can be partially interpreted, as in [20]. For example, assuming x and y are integers, $y = x * 2$ is defined in the original code. We could use the UF operator to weaken the accurate multiplication computation constraint between x and y with a relational function as $y >_{(UF)} x$.

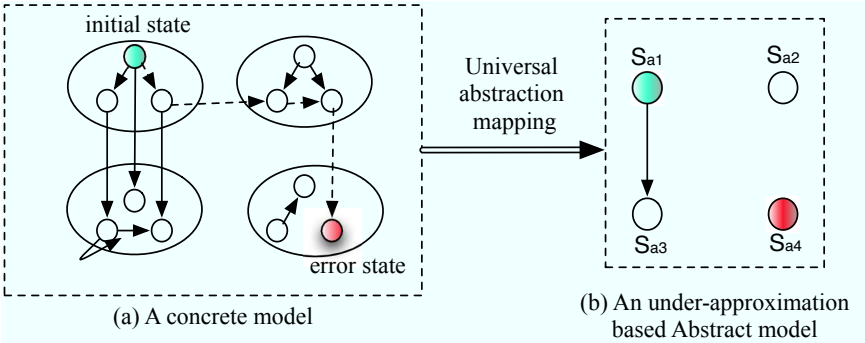


Figure 2.19: Under-approximation based abstraction

2.4.2 Under-approximation based Abstraction

In an under-approximation based abstract model, an abstract transition is made from an abstract state if there exists a tradition from *all* corresponding concrete state. This abstraction is also called the *universal* abstraction. Figure 2.19 gives an example of such an abstract model construction. In the abstract model of Figure 2.19(b), the error state S_{a4} is not reachable from the initial state S_{a1}. However, the error state in the concrete design is actually reachable along the transitions drawn with the dash line in Figure 2.19(a). So this abstract model causes the false positive error. Compared with the over-approximation based abstraction, the under-approximation based abstraction actually *removes* some behaviors of the concrete model.

Here we briefly introduce two realistic under-approximation based abstraction techniques for software verification, which are also used in our work. First, an abstract model can be built by constraining the

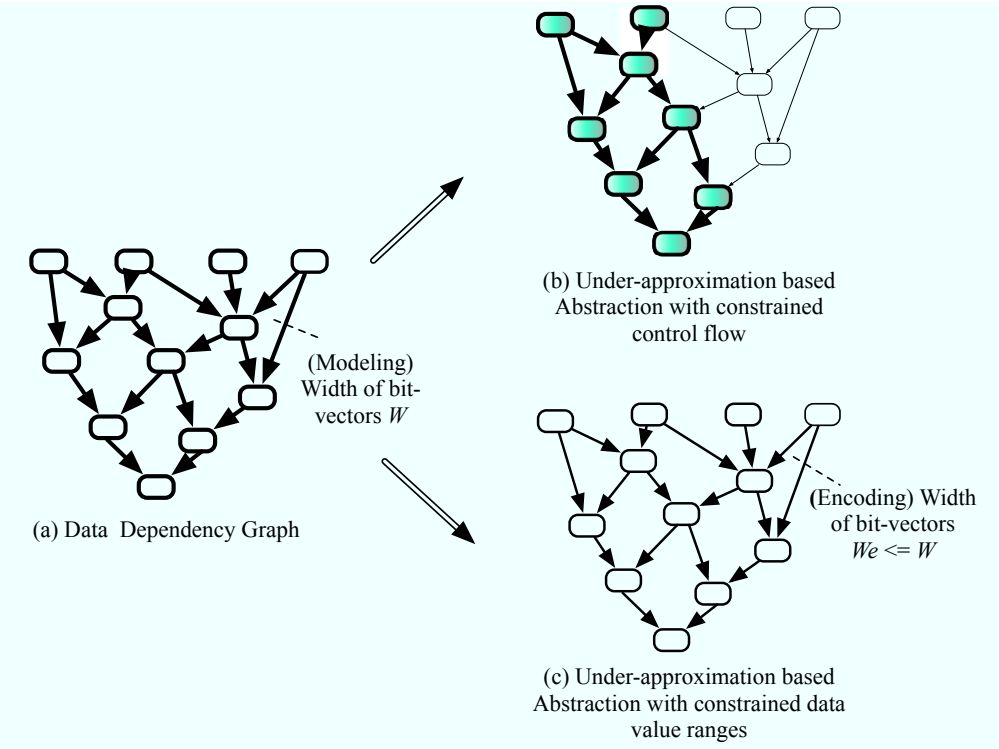


Figure 2.20: Two examples of under-approximation based abstraction

program execution on a certain subset of paths in the concrete model. For example, enforcing some control predicates in the program to some constant values so that only one branch is taken for each enforced predicate. This is shown in Figure 2.20(b). Second, another kind of abstract model can be built by constraining the value of variables to a smaller range compared with the original variable value ranges in the concrete model. This can be realized by setting the encoding width W_e of bit-vector variables smaller than their individual modeling width W as illustrated in Figure 2.20(c). Both of these abstract models do not need the state space mapping.

2.4.3 Ternary Abstraction

Ternary abstraction [15, 91] is based on the three-valued logic, which enhances the two-valued Boolean logic with a third value X denoting the unknown value. The traditional symbolic simulation [17, 101], which uses these three values $\{0, 1, X\}$ as symbolic inputs to abstract the inputs space, is considered an application of ternary abstraction. It can cover multiple system executions using concrete value

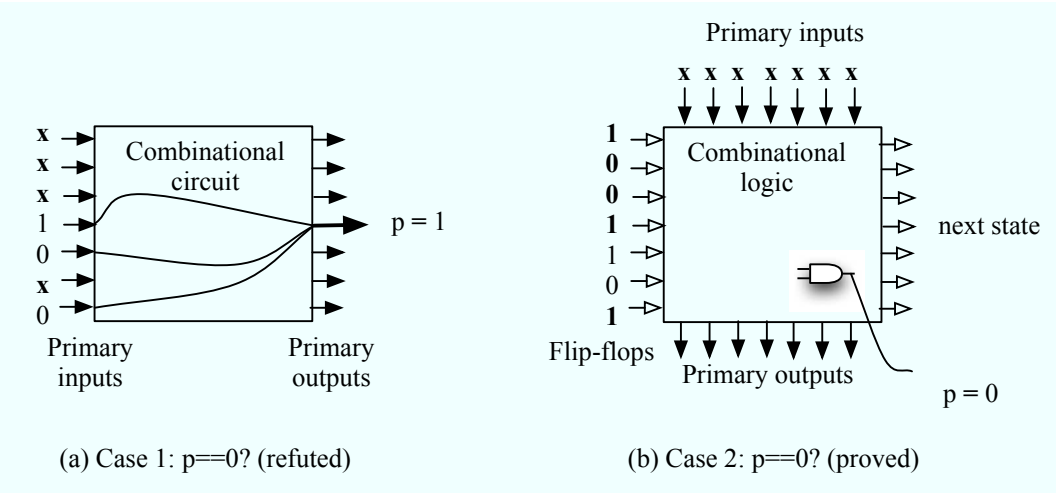


Figure 2.21: Two verification examples using ternary abstraction

inputs in a single symbolic run, thus it has the potential of greatly reducing the size of verification problems. Through dual rail encoding, such three-valued variables and operators have a propositional representation, which can be processed by a Boolean SAT solver. Ternary abstraction uses the value X to abstract the computation in the design that does not affect the property being checked. It is capable of both proving and refuting the correctness of the design, depending on the concrete applications. As in Figure 2.21(a), we aim to verify the property that the value of one output p in the combinational circuit always be 0. We could find a counterexample which sets $XXX10X0$ at the primary inputs (PI) makes p to 1, thus refuting the property. The computations related to the PIs set by X are abstracted away. For another example in Figure 2.21(b), the property to be verified is that some internal gate p is always 0 starting from the state 1001101 in the sequential circuit. If p is 0 while setting 1001101 to the corresponding flip-flop, and setting all PIs to X, we could conclude the property holds.

However, it is a challenging task to decide which variables should be enforced with value X, so as to construct a precise but simple abstraction that does not produce the inconclusive X at the property under verification.

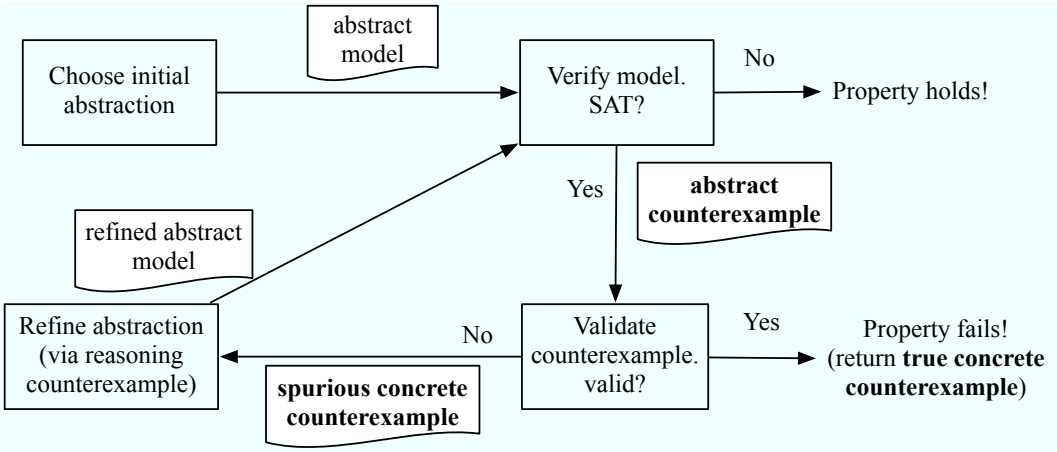


Figure 2.22: Basic flow of CEGAR

2.4.4 Two Refinement Methods

Counterexample guided abstraction refinement (CEGAR) and proof-based refinement are the two most widely used abstraction refinement methods. CEGAR was initially proposed to automate the localization abstraction [78] and has been extended with several variations [32, 52, 81]. The combination of CEGAR with predicate abstraction has been applied in several software model checking tools [9,26,60]. The overall iteration of CEGAR with four stages is shown in Figure 2.22. Given a design to be verified and the specified property, this method begins with an initial abstraction chosen by the user (It can be as coarse as a *TRUE* state). In the second stage, it applies some verification technique, like model checking, to check the property in the abstraction. If the property holds in the abstraction, the verification is done; otherwise, an abstract counterexample λ is found. In the third stage, the feasibility of λ in the original design is validated. If the answer is yes, the property fails and the corresponding true counterexample in the original design is returned; otherwise, the constraints along the spurious counterexample in the original design are reasoned in the fourth stage of refining the abstraction to produce a refined abstract model. The iteration continues until the property is proved to be valid, or a counterexample in the original design is found and returned for debugging.

Proof-based refinement [87] is an alternative abstraction refinement method, which was proposed based on the BMC and SAT/SMT solver. The basic work-flow of this method, also with four stages, is shown in Figure 2.23. (1) Given an initial bound K, it first verifies the satisfiability of the *negation* of the given

property in the design unrolled with bounded K, similar to a BMC run. If the instance is SAT, a true counterexample with bound K can be obtained to show the violation of the property; if the instance is UNSAT, it means that there is no counterexample within bound K. (2) It then builds an abstraction α with the set of design constraints, used in the UNSAT proof generated from a SAT solver. (3) It verifies α without any bound constraints. If the property holds in α, it means that the property holds in the original design as well; otherwise, an abstract counterexample λ that violates the property in abstract model α can be obtained. (4) Finally, the bound of λ, instead of the values assigned in λ, is used to determine a larger bound K' for starting a new loop iteration.

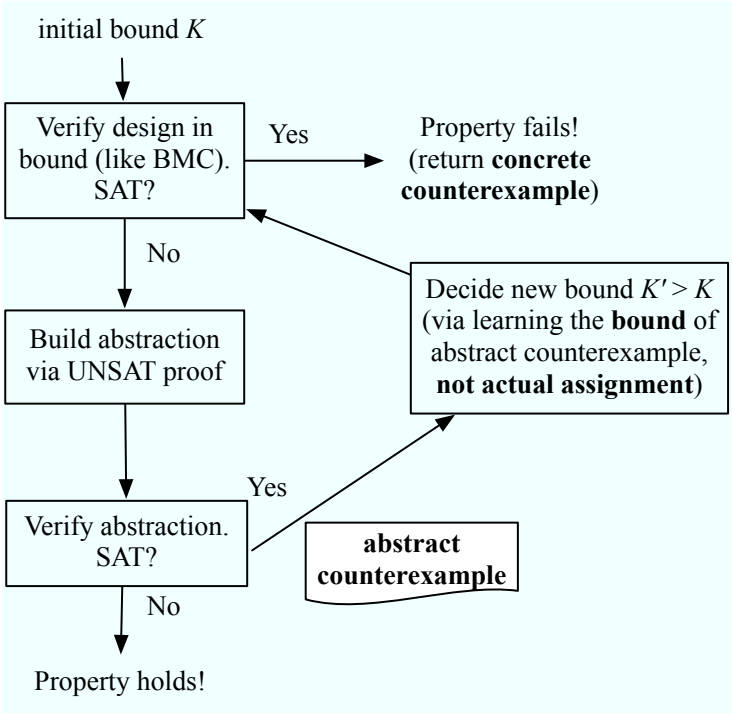


Figure 2.23: Basic flow of proof-based refinement

Comparatively speaking, with the proof-based abstraction refinement approach, all counterexamples within the bound K are ruled out at once, whereas CEGAR may require many refinement iterations to achieve this. Moreover, using a single abstract counterexample — as in CEGAR for the refinement— is risky because there may be many ways to remove one specific counterexample. It is highly possible that the restored constraints actually are not relevant to the property being proved. However, the proof-based approach has one main drawback. It may be much more computationally intensive to extract the refutation proof that there are no counterexamples of bound K, than to refute a single abstract

counterexample [79]. Thus, the refinement step could become a bottleneck in the verification process. In practice, it is hard to find the pure CEGAR process or the proof-based refinement process. For example, some CEGAR processes use interpolants to obtain new predicates, where interpolants are the results of the UNSAT proof extraction. Several approaches have been proposed to combine these two refinement methods, which take advantages of both, and at the same time alleviate their disadvantages [2, 13].

Chapter 3

Scalable BMC via Integration of Program Slicing and Proof-based Localization Abstraction Refinement

As explained in the previous chapter, due to the state explosion problem, scalability is one of the major obstacles in the application of model checking to large software designs. In this chapter, we present a new verification approach to combine aggressive program slicing with a proof-based localization abstraction-refinement strategy toward the scalability enhancement of bounded model checking embedded software. While many software model-checking tools use program slicing as a separate or optional step [26, 35], our program slicing is tightly integrated in the model construction and reduction process. Furthermore, it incorporates the compilation optimization techniques, so as to compute a more accurate slice. In exploring the application of a proof-based localization abstraction-refinement strategy based on the under/over-approximation of our proposed software model, we propose a heuristic method of deciding new encoding size of bit vector variables to effectively refine the under-approximation. Experiments on C programs from wireless cognitive radio systems show that, this approach can greatly reduce the model size, and shorten the solving time by the SAT-solver.

3.1 Motivation and Overview

Since the verified properties usually depend only on a small portion of the program, our proposed approach aims to accurately and efficiently find this portion with low cost and complexity.

Program slicing [31, 113] and localization abstraction are two important techniques to enhance the verification scalability. Program (static) slicing [30, 63, 107] automatically extracts a subset of program segments called slice(s), which involves only the variables referred to in the slicing criteria. Here, the slicing criteria can simply be the target property assertion statement. The slice(s) is an accurate abstraction without incurring spurious errors, since it includes all the program constraints, which have the *computation relevance* to the variables referred to in the assertion. In this regard, program slicing is conservative. In other words, the sliced program is sufficient to prove the target property.

Localization abstraction is another important abstraction technique, which could further enhance scalability. It can identify a small set of program constraints, which have *verification relevance* to the asserted property. A proof-based refinement method using under- and over-approximation is applied to iteratively automate this abstraction process. Figure 3.1 illustrates the differences between program slicing and abstraction refinement in the model reduction. Slicing removes the irrelevant constraints step by step, so the complexity of the abstraction is also reduced step by step. On the other hand, abstraction starts with a small representation of the original design and gradually learns verification relevant constraints during refinement. So the complexity or the accuracy of the abstraction increases step by step. To accomplish our goal of identifying the small set of program constraints relevant to the verification for scalable BMC, we explore the combination of these two techniques, considering their different behaviors in the reduction of model complexity.

The rest of the chapter is organized as follows. In Section 3.2, we present our proposed software verification model, model construction and the program slicing based model reduction. In Section 3.3, we introduce our bounded model checking with the proof-based abstraction-refinement strategy to construct the localization abstraction. Experiment results are reported in 3.4 followed by related work and summary in Section 3.5 and 3.6.

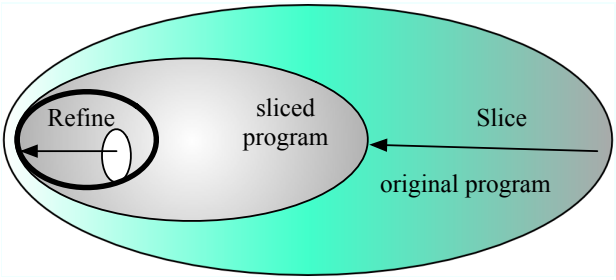


Figure 3.1: Comparison of program slicing and abstraction refinement (Circle: program constraints or verification complexity, circle with the darkest border is the final abstract model)

3.2 Software Verification Model

In this section, we first introduce the structure of our proposed software verification model, followed by its construction and reduction.

3.2.1 Model Structure

Our proposed structural verification model M is a Digraph. In M, each node represents a computation, which operates on variable values from its input edges and produces the result to the output edges. Each directed edge represents a data flow, which carries the variable value computed by its source node to its target nodes. Our verification model M has three derivatives: M_H, M_{BV} and M_B. In the high-level model M_H, the nodes represent the computation of the program variables defined in the code, and the edges represent the computation result in the first-order variable domain. Every node belongs to one of the following three categories:

1. Input or constant nodes;
2. Arithmetic or relational operations;
3. Ternary conditional operation $c?T:F$.

As a data-flow-like representation of the program, M_H is similar to the value dependence graph (VDG) [112], which was proposed for simplifying the compiler optimization and supporting the program slic-

ing [44] in fine granularity. However, our M_H is built on the pre-processed programs, in which all loops' iterations have been unwound to a certain bound, and all function calls have been inlined; moreover, the SSA-transformation has been performed to the code. So M_H can be structurally simpler than VDG, which is constructed from the raw programs with full programming language features. M_H has one important feature that the edges from the *guard* nodes only connect with the c?T:F nodes. This is because only these nodes' outputs are conditional, which need to be determined by the guards at verification time.

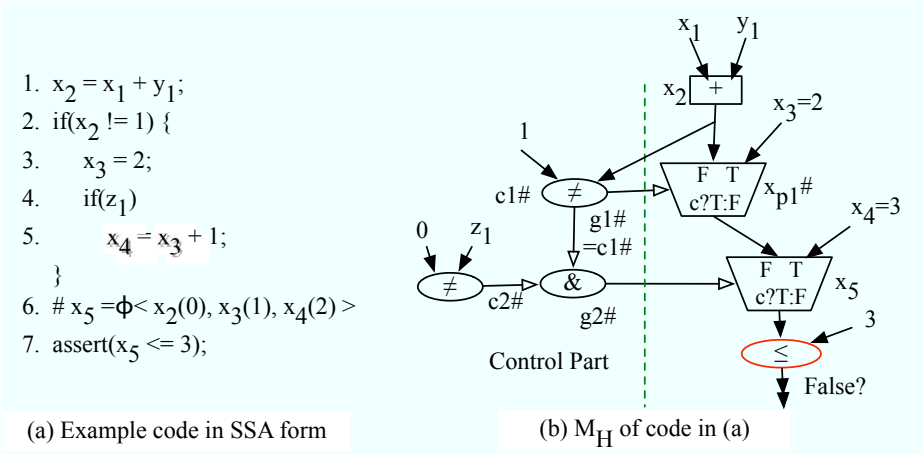


Figure 3.2: An example of M_H

Figure 3.2 gives an example of an SSA-transformed code and its model M_H. In Figure 3.2(a), the subscripts distinguish different variable versions in the SSA form. For example, the variable version x_5 is defined by a ϕ function parameterized with three variable versions x_i $(i \in I = \{2,3,4\})$ reachable to x_5, listed by their definition order. To use the definition of some x_i during execution, its Definition Condition (Def-C) must be satisfied. Def-C is actually the guard of the BB, in which this variable version is statically defined. x_5 is defined by the variable version x_k (k∈I), whose Def-C is true at the actual execution, and whose order in the parameter list of ϕ is the biggest (latest defined in the program). In Figure 3.2 (b), all node names ending with the symbol # represent new intermediate variables inserted in M_H. Two "c?T:F" nodes together represent the ϕ function. For example, node x_5 has three inputs: $g2\#$ is a condition, x_4 is True branch and a new variable $x_{p1}\#$ is False branch. $g2\#$ modeled as the guard of BB2 $((x_2! = 1) \wedge (z_1! = 0))$ is the Def-C of x_4. Similarly, $g1\#$ modeled as the guard of BB1 $(x_2! = 1)$ is the Def-C of x_3 connected to the node $x_{p1}\#$. If $g2\#$ is true, x_5 is assigned by x_4; otherwise it is assigned

by x_{p1}#, whose value is decided in a similar way by $g1$#. So the two "c?T:F" nodes correctly model the ϕ function. The relational node in red represents the property assertion. Thus, the property verification problem is converted to checking if the value of the output node of the model can be False.

After the high-level model reduction (details to be presented in the following subsection 3.2.2), we compute another model derivation M_{BV} by modeling every program variable as a bit-vector variable with a fixed width, but leaving all program operations as before. The main purpose of this step is to model the data in bound, instead of considering them as words, which can have the arbitrary or infinite values like integer numbers as in M_H.

To compute the model derivation M_B, we convert each program operator in M_{BV} to the corresponding Boolean logic operators through circuit translation. During the conversion from M_{BV} to M_B, some new nodes may be added to complete this translation, like in the translation of arithmetic addition operation with Boolean logic gates. In M_B, each node represents a logic operator and each edge represents a bit. By converting the logic gate to its representative CNF clauses (this can be done in a single pass through the logic circuit), we could use one of the state-of-the-art SAT solvers, to decide the satisfiability of the monitor output node set equal to a certain value, for our purpose of formal assertion checking.

All model derivatives are implemented as doubly linked circuit netlists. Every element in the netlist stores the properties of a model node and its connection with other nodes.

3.2.2 High Level Model Construction

Given a SSA-transformed program P and its property Ω which is specified as an assertion statement $S(\Omega)$ in P, the high-level model M_H is constructed in the following four steps:

1. Perform program slicing to slice BBs with respect to $S(\Omega)$ via the reachability analysis of G_{CF}^{SSA} of P.

   Result: A set B of BBs {BB[1],..., BB[N]} is obtained, which are reachable to $S(\Omega)$; a set PHI_SET of *Phi* functions $\{\phi[1],\ldots,\phi[M]\}$, each of which can be found in the front of some Merge node BB in the set B.

2. Compute *guard* of every BB in the set *B*, which must be satisfied in order to enter this BB.

   Let b and g denote the current BB and its guard, respectively, and let b' and g' denote the target BB that b jumps to and the corresponding guard expression of b' computed by the path from b. First, g of every BB is initialized to $TRUE$. The computation of g' proceeds by a case split on the last statement s of b. Figure 3.3 gives the SSA transformed control flow graph G_{CF}^{SSA} of the example code in Figure 3.2(a). We use it for illustrating the computation.

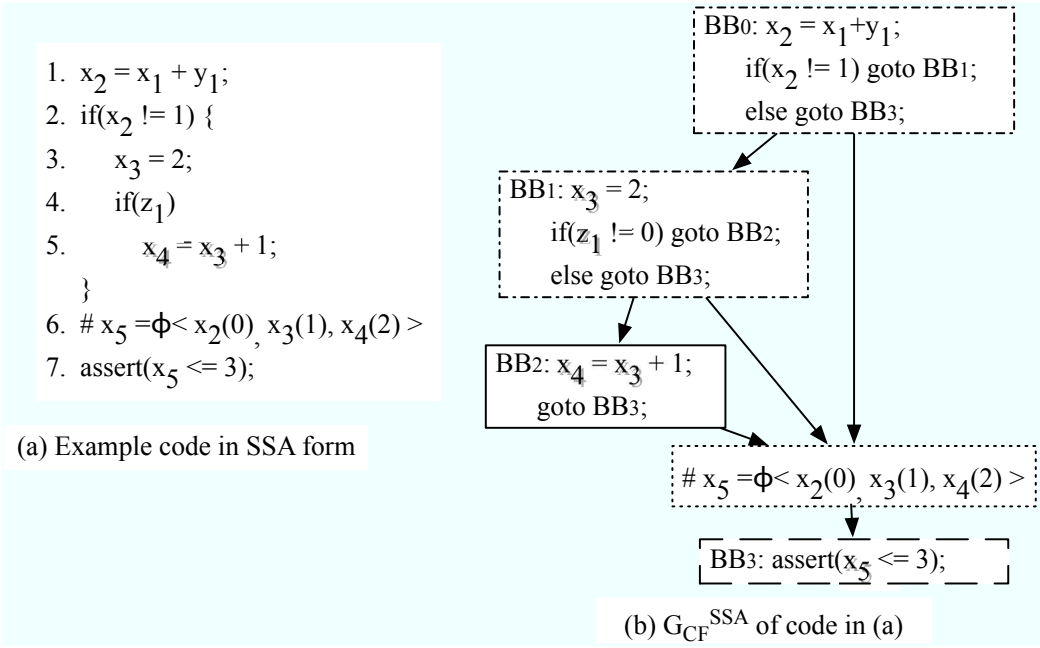


Figure 3.3: An example of G_{CF}^{SSA}

- Case 1: s be a conditional *goto* statement as *"if(c) goto b' else goto b''"*.

  Analysis: b' is entering the control range of c. So $g' = g \wedge c$.

  For example of the *"goto BB_1"* statement in BB_0 of Figure 3.3, we could compute the guard of BB_1 as $g(BB_1) = g(BB_0) \wedge c_{BB_0} = (x_2 ! = 1)$.

- Case 2: s be a conditional *goto* statement as *"if(c) goto b'' else goto b'"*.

  Analysis: b' is entering the control range of $\bar{c}$. So $g' = g \wedge \bar{c}$.

  For example of the *"goto BB_3"* statement in BB_0 of Figure 3.3, the guard of BB_3 is computed as $g(BB_3) = g(BB_0) \wedge c_{BB_0}^{-} = \neg(x_2 ! = 1)$.

- Case 3: s be an unconditional *goto* statement *"goto b'"*.

Analysis: b' is in the same control range as b. So $g' = g$.

Considering the "*goto* BB_3" statement in BB_2 of Figure 3.3, we could compute the guard of BB_3 as $g(BB_3) = g(BB_2) = ((x_2! = 1) \wedge (z_1! = 0))$.

For a merge node which has multiple incoming paths, the final guard is the disjunction of all guard expressions computed from these paths. For example of BB_3 in Figure 3.3, its guard is finally computed as $g(BB_3) = \neg(x_2! = 1) \vee ((x_2! = 1) \wedge \neg(z_1! = 0)) \vee ((x_2! = 1) \wedge (z_1! = 0)) = TRUE$. Different from computing the *guard* for every statement as in [35], we build one guard for every BB, so that all statements inside the BB can share the same *guard*.

3. Compute program constraints C_P and the target property constraints C_Ω as program equations.

   (a) C_P consists of two parts: C_B, which is the conjunction of constraints from all BBs in set B, and C_Φ, which is the conjunction of constraints from all ϕ functions in set PHI_SET. Since every assignment $A(i, j)$ in set B is deterministic, we can easily derive C_B as:

   $$C_B = \wedge_{i=1}^{N}(\wedge_{j=1}^{L} A(i, j))$$

   where N is the size of set B, and L is the number of assignments in $BB[i](i \in [0, N-1])$.

   The variable version on the Left-Hand-Side (LHS) of a ϕ function may be assigned by any variable version in the parameter list, which is decided by the definition order and Def-C of each parameter. As discussed before, during the actual execution, LHS is assigned by the last variable version (according to the definition order) whose Def-C is true. Since we assume that the variable versions in the parameter list have already been ordered by their definition sequence, we could simply compute $C_\phi[i]$ $(i \in [0, M-1])$ for the i^{th} ϕ function $(LHS[i] = \phi(v[1, i], \ldots, v[K, i]))$ in the set PHI_SET as:

   $$\begin{aligned} C_\phi[i] = ((&tv[1, i] == v[1, i]) \\ &\wedge_{k=1}^{K-2} (tv[k+1, i] == (G(v[k+1, i])\ ?\ v[k+1, i] : tv[k, i])) \\ &\wedge (LHS[i] == (G(v[K, i])\ ?\ v[K, i] : tv[K-1, i]))) \end{aligned}$$

   where K is the size of parameter list of $\phi[i]$, $tv[k, i]$ is the k^{th} newly conjuncted variable of $\phi[i]$, $v[k, i]$ is the k^{th} RHS parameter of $\phi[i]$, G is a function that returns the *guard* of the given parameter.

   And $C_\phi = \wedge_{i=1}^{M} C_\phi[i]$, where M is the size of the set PHI_SET.

   Finally $C_P = C_B \wedge C_\phi$.

(b) To compute C_Ω, let a and g denote the specified assertion and the *guard* of the BB where a is defined respectively.

$$P_\Omega = \bar{g} \vee a$$

$C_p := (x_2 == x_1 + y_1) \wedge (x_3 == 2) \wedge (x_4 == x_3 + 1)$
$\wedge (x_{p1} = (x_2 != 1) \; ? \; x_3 : x_2)$
$\wedge (x_5 = ((x_2 != 1) \&\& (z_1 != 1)) ? \; x_4 : x_{p1})$
$C_\Omega := x_5 <= 3$

Figure 3.4: Program equation C_P and C_Ω

Figure 3.4 shows the program equations C_P and C_Ω of the SSA-transformed program in Figure 3.2(a). For instance, the first statement $x_2 = x_1 + y_1$ is transformed to the first literal in C_P. The assignment with *Phi* function at line 6 are transformed to the 4^{th} and 5^{th} literals of C_P. The assertion statement at line 7 becomes C_Ω.

4. Construct the high-level model M_H.

   This step is straightforward as M_H is essentially the graphic representation of the constraints in the C_P and C_Ω equations. The M_H for our running example is illustrated in Figure 3.2(b).

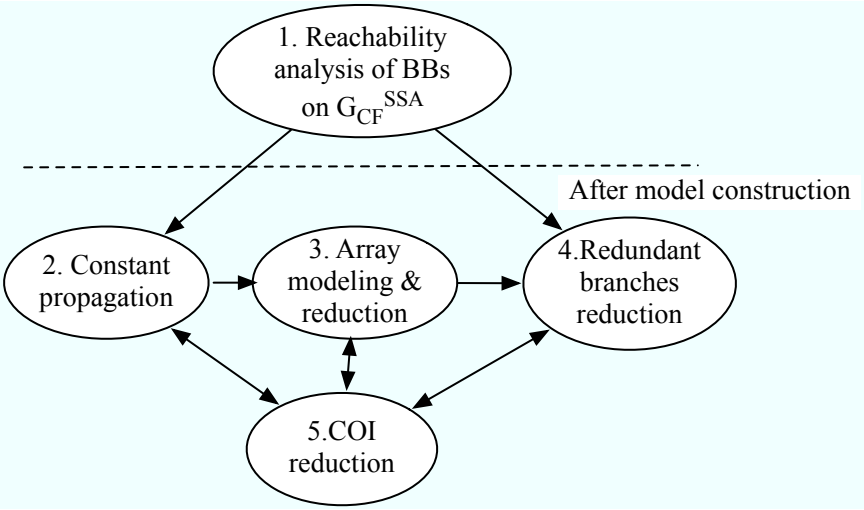


Figure 3.5: Basic procedures of static slicing

3.2.3 High Level Model Reduction and Array Modeling

We perform static program slicing through reduction steps as shown in Figure 3.5. The slicing criterion is set as the target property assertion statement. Slicing starts by the backward reachability analysis of Basic Blocks (BBs) during the model construction. All BBs unreachable to the target assertion statement are immediately sliced away. But the obtained slice is considered to be very coarse because not all computations inside the reachable BBs are necessarily relevant. At this model reduction stage, the slicing process can go to either procedure 2 or 4, depending on whether the current slice has operations over constants and array variables or not. COI (Cone-of-Influence) reduction, as a separate procedure can be invoked after any procedure 2, 3 and 4. Its purpose is to remove the nodes identified as irrelevant by any invoking procedure. Finally the model reduction stops at procedure 5.

Constant Propagation

It is a popular compiler optimization technique [1] whose goal is to discover and propagate constants through the program [111]. Any node in M_H whose inputs are all constant values can also be evaluated as a constant node and propagated further. We use a simulator similar to a logic simulator to implement the constant propagation on model M_H. But, the value evaluation here is conducted in the Integer or Floating Point domain, instead of being restricted in the Boolean domain. After the propagation, we could remove the redundant constant nodes, which have no connections to other non-constant nodes. This removal reduces the M_{BV} size, and further reduces the computation cost of logic implication with constant value cost at the level of bit vectors. In the case that the constant True (False) is propagated to the conditional input c of the "$c?T : F''$ node, the connection of the False (True) input branch could be removed, as the branch can be determined as never taken.

Array Modeling and Reduction

We first assume all array variables have fixed sizes, which is generally true in the embedded programs being verified. We also assume that there are only two program operations on array variables. Let A be an array variable, and e and x are scalar variables; these two operations are: "A[x] = e" for implementing

Store access; and "e = A[x]" for **Select** access. Actually, all other array operations can be transformed as a series of steps involving only these two simple operations.

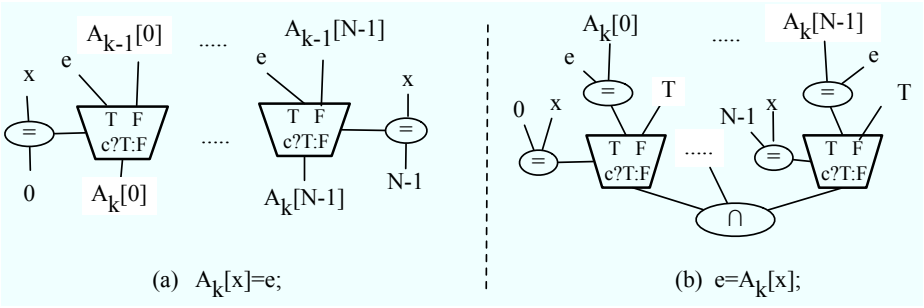


Figure 3.6: Array node expansion

An array node A is first expanded to a set of nodes, each of which represents an element of A. For "A[x] = e," only the x^{th} element of A receives the new value e, while all other elements of A retain their previous values. Figure 3.6(a) models the following constraint, using the SSA representations of A imposed by this operation, where A_{k-1} is the latest version of array variable A at this operation location:

$$\forall i \in [0, N-1], A_k[i] = ((i == x)\ ?\ e : A_{k-1}[i])$$

Figure 3.6(b) models the following constraints imposed by Select access:

$$\wedge_{i=0}^{N-1}((i == x)\ ?\ (e = A_k[i]) : True)$$

The modeling of array in Figure 3.6 is for the general case, where the index x is not known. After all loop structures are fully unwounded and constant propagation applied, the value x can almost always be computed. Then the complexity of modeling array access operations can be greatly reduced. For **Store** accesses, all "c?T:F" nodes are simply substituted by nodes with "=" operator (as the condition c is deterministic): the node representing the x^{th} element $A_k[x]$ is connected with e, while any other element node $A_k[x](i \neq x)$ is connected with node $A_{k-1}[i]$ correspondingly. For **Select** accesses, only the array element, whose index equals x, is connected with the node representing variable e, leaving the rest of the array element nodes unconnected from node e.

For the nodes with the assignment ("=") operator, which are redundant in logic reasoning, we perform the transitive equality reduction to slice away these nodes to reduce the model size. For the nodes

with no output connection, they can be easily identified and removed during COI reduction. Through this reduction, the model size can be independent from the array size, but on the number of **Store** and **Select** accesses on the array. In a word, this reduction exploits the deterministic array accesses after the unwinding of loops operating on arrays, as the referred array element index becomes constant.

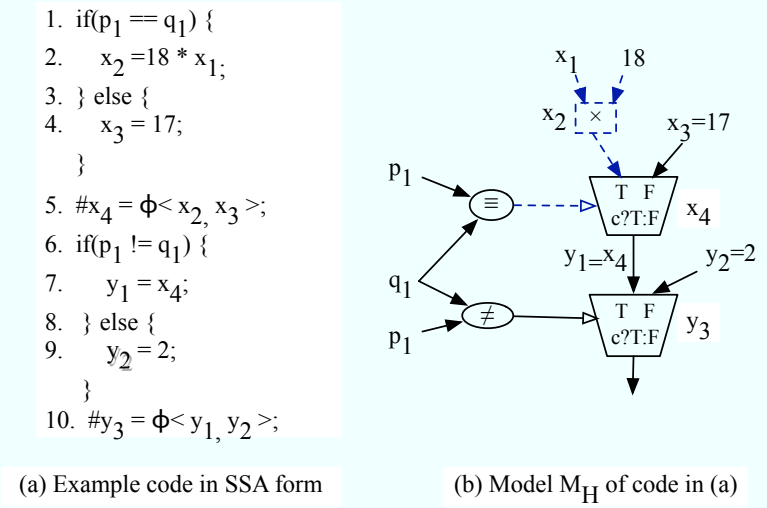


Figure 3.7: An example of redundant branches

Redundant Branch Reduction

A motivational example is given in Figure 3.7. In Figure 3.7(a), line 2 is redundant to the computation of *y* in line 10, due to the constant negation between two conditional predicates. This kind of redundancy is reported as undetectable, by most traditional slicing techniques without compiler optimization [107]. But it can be identified and removed in the two steps performed on Model M_H:

1. Identify the pairs of conditional predicate nodes, which have the constant equality or inequality correlation.

2. Determine the redundant computations according to case pattern recognition. Currently, we only consider a limited number of redundant case patterns. For one example of Figure 3.7(b): as two condition predicates are constantly negated and node x_4 is connected with True branch of node y_3, the constraints on True branch of x_4 are identified as redundant shown in the dash line.

3.3 Software Bounded Model Checking

A proof-based abstraction-refinement strategy proposed for bit-vector arithmetic reasoning [20] is explored on the bounded model checking to enhance scalability.

3.3.1 Major Steps in Refinement Framework

1. **Under-approximation construction**

   Every free variable node v in the model M_{BV} has an encoding size W_e used for under-approximation. For bit-vector variables whose modeling width W is greater than W_e, we enforce additional constraints on its most significant bits by setting them to a constant value. For the example of unsigned free variables with their most significant bits set to "0", their value that is originally arbitrary in range $[0, 2^W - 1]$ is now restricted in range $[0, 2^{W_e} - 1]$. In our implementation, we simply use single-literal clauses to restrict these most significant bits in the CNF formula. If the SAT solver returns satisfiable for this constrained CNF formula, we can conclude that the target assertion fails, and the SAT solution can easily be converted to a counterexample in the program to show the error trace of violating the assertion. This counterexample is definitely feasible in the program, since the state space of an under-approximation model is a subset of that in the original model. If the SAT solver returns UNSAT, it is inconclusive about the assertion. But we could extract the UNSAT core as a proof to guide the following abstraction construction.

   It is important to maintain consistency among the added constant constraints to avoid the UNSAT case caused by these inconsistent constraints. This is because the SAT solver may return an arbitrary UNSAT core, which is not useful for identifying the relevant program constraints to build the abstraction. For instance, consider the example program statement "y==x+1". We set the same encoding sizes W_e for all node variables, which involve translating this word-level statement in the model M_{BV}, and set the most significant bits of these variables to be the same kind of constant constraint: either all "1" (TRUE) or all "0" (FALSE). Assume that, we enforce the sign bit(the most significant bit) of x to 1, which means x is a negative number, but enforce the sign bit of y to 0, which represents y as a positive number. We could find that these enforced constant values conflict with each other, which causes the constrained instance UNSAT. But this

information is not considered as directly relevant to the actual verification problem.

2. **Abstraction construction and verification**

   Constructing the abstraction model M_{abs} is an over-approximation procedure, which extracts a subset of nodes from model M_B. The main idea of Algorithm in Figure 3.8 is to group the set of nodes in M_B, according to the translation relationship with the nodes in model M_H. So, if any bit of node variables in this set is used in the UNSAT core, all other bits of the node variables are also included in M_{abs}; otherwise, they don't contribute to M_{abs}. This is to simplify the abstraction and reduce the non-determinism of the over-approximation. Some popular SAT-solvers provide an UNSAT core if a formula is UNSAT. The core C usually has two parts: a set of bit variables V used in C, and a set of clauses CL involved in C. Since all clauses in the original CNF formula, which use variables in the set V, are a superset of the set CL, it is safe to use V to build the abstraction.

```
Over_appr (Model M, Abstract Model M_abs, set V)
1: for each node i in high-level model M_H
2:    Map to a set N of nodes in M_B that translate node i;
3:    Get a set V2 of bit variables from all nodes in set N;
      /* Intersection set of V2 and V; */
4:      Compute set V3 = V2 ∧ V ;
5:    if V3 not Empty
6:       Add all nodes in set N to M_abs;
7:       Remove all bit variables in V3 from V;
      /*stops if V has been included in M_abs*/
8:    if set V empty
9:       break;
10: end for
```

Figure 3.8: Over-approximation algorithm

Theorem 1. M_{abs} is an abstraction of M_B.

Proof. Since M_{abs} has a subset of nodes in M_B, the set of CNF clauses K_{abs} of M_{abs} is also a subset of CNF clauses K_B of M_B. Let α be a satisfying assignment of M_B. Under this assignment, since all clauses in K_B are true, any subset of clauses must also be true. K_{abs} must be satisfied, and so M_{abs} is also SAT. Conversely, if K_{abs} is UNSAT, K_B must be UNSAT as all clauses in K_{abs}

referred to prove UNSAT are also included in the set K_B. So, we could conclude that M_{abs} is UNSAT implies that M_B is also UNSAT. □

Theorem 2. M_{abs} encoded with W_e is UNSAT.

Proof. First, M_B encoded with W_e was UNSAT with a UNSAT Core C; Second, M_{abs} includes all the bit variables used in the set V of C. Third, after constraining M_{abs} with the encoding size W_e, all clauses that contribute to C also exist in the constrained M_{abs}. So the constrained M_{abs} is UNSAT. □

Based on Theorem 2, all counterexamples within bit-width W_e for variable v can be ruled out from M_{abs}. If there is a counterexample in M_{abs}, variable v must have a width larger than W_e. Since at least one variable needs to increase its encoding size in every iteration, this refinement procedure is ensured to terminate. If M_{abs} is UNSAT, the assertion proves to be true. Otherwise, a counterexample (potentially spurious) is returned, which is used to direct refining the under-approximation in next iteration.

3. **Computation of encoding size W_e for refining under-approximation**
   Given a counterexample, each variable in M_{abs} is assigned a value O. One direct way to determine an updated value W_e for each variable in M_{abs}, is to let W_e be big enough to cover O. For the variables not in M_{abs}, we derive their W_e by data dependency to keep the consistency among new constraints added by the updated encoding size. In a word, we make use of the width of assigned bits in the counterexample instead of the assigned values on the bits to direct the next refinement iteration, as the counterexample-based refinement does. Note that after this process, different variables may be assigned different updated W_e'.

3.3.2 New Encoding Width Computation

Due to the decision heuristics of the SAT solver, a counterexample may assign large values on variables [54]. So the updated encoding width based on the values of variables given in the counterexample may be unnecessarily large. Since extracting UNSAT cores becomes increasingly difficult with an increasing

```
/* Given 1) a counterexample Cex
 2) a set N of Boolean variable nodes corresponding to conditional
predicates in program and its value implied in the under-
approximation without decision. */
Procedure New_Encode (Cex, set N)
1:  for each Boolean variable node i assigned in Cex
2:     if (node i in set N) && (value of i in Cex != value of i in N)
3:        Mark node i;
    end for
4:  if (any node marked)
5:     for each marked node i
6:        Compute forward & backward slice of node i;
7:        Decide larger S_i' that i be free value in under-approx;
          /* via data dependence analysis */
8:         Compute updated S_v' to all nodes v in slice;
9:      end for
11: else
12:    Follow old encoding size computation;
```

Figure 3.9: New encoding size computation algorithm

encoding size, the whole refinement may suffer the performance loss. We propose a heuristic method to improve the accuracy of the updated encoding size using program analysis. In the algorithm shown in Figure 3.9, when a counterexample λ is produced, we first identify all node variables in M_B that correspond to the control predicates in the program. Then we find out those whose value T implied by the enforce constraints in the under-approximation are different from the value assigned in λ. These nodes that decide which program paths to be taken are important to help find the SAT solution in the under-approximation. We then compute their enlarged W_e' to allow the non-fixed values in the refined under-approximation, and we use the data dependency to determine W_e' of other nodes. For example: $(x > 100)$ with old $W_e = 4$ is constantly false, W_e' of x is then set 7 to avoid the constant value in new under-approximation. Since many control predicates have a constant RHS in our verification problems, it is easy to compute the new W_e'.

3.4 Experimental Results

We implemented the proposed approach in C++, which is called C2BIT and used it to verify the safety properties in C programs from a wireless cognitive radio system [98]. Our benchmark programs are extracted from two safety-critical components in this system. One is the policy engine that enforces regulatory restrictions on the waveform. Another is the cognitive controller. These programs have two important features:

1. Most loop structures have an upper bound so the execution always terminates. For example, there are 10 C programs in the controller with 23 for loops whose upper bound is explicit, and there are 15 while loops, 12 of which the maximum bound is statically known. The other 3 loops are $while(true)$ for monitoring sockets with very simple operations, which are not our verification targets.

2. Most Array variables have constant size.

With all loops unrolled to a certain bound, all function calls inlined and properties specified as assertions, we use open source GCC 4.0 compiler to generate the SSA form for the pre-processing. The maximum bound K of the unrolling can be statically identified for formally proving the program correctness. Then C2BIT uses this form as input for BMC. During model checking with the proof-based abstraction refinement, C2BIT uses zChaff to check the under-approximation, and extract UNSAT core as the proof for the UNSAT instance, and uses MINISAT to check the satisfiability of the over-approximate abstraction. This is because zChaff provides a more user friendly UNSAT core compared with MINISAT. All experiments run on Intel Xeon 2.8GHz processor with 2 GB RAM.

In Figure 3.10, we first show the effectiveness of our slicing method using a simple BubbleSort code [76], compared to CBMC *v*2.4 with "–slice" option. The property we checked is $assert(A[N-2] <= A[N-1])$ where A is an array variable, N is the size of A. All loops are maximally unwound. The left plot shows the CNF formula size generated by CBMC grows very fast. While the growth of C2BIT is almost proportional to N. The results also confirm our claim that the CNF formula size generated from C2BIT depends on the number of array accesses, not on the array size. (Otherwise the CNF size should

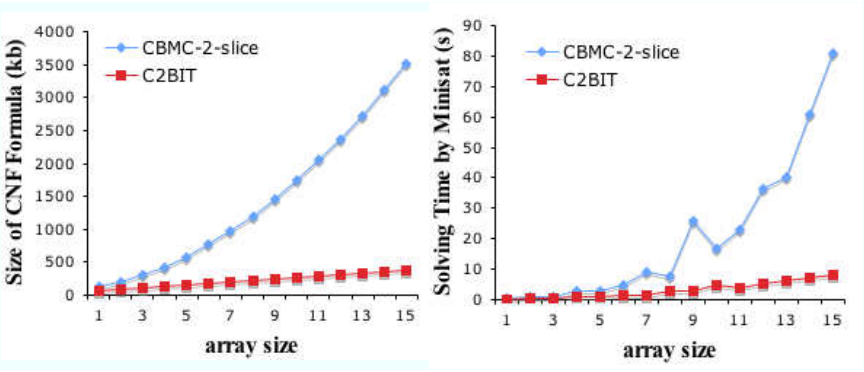


Figure 3.10: Results on bubbleSort

be proportional to $N2$.) The right plot shows that the solving time by MINISAT solver is basically consistent with CNF size for both CBMC and C2BIT. The detailed data is given in Table 3.1.

Table 3.2 shows the results on 10 selected property verification problems on *C* programs. "TO" means exceeding 3600s. The max bitvector width W is 32 for every property. P3 to P9 are different properties on the same unrolled program. The rest are different properties on different programs. Col (Column) 2 and 3 show the number of lines of code after loop unrolling and the satisfiability of properties being checked. The memory usage and solving time by MINISAT for CBMC and C2BIT+S are given in Col 4-5 and Col 6-7 respectively. Col 4 to 8 of Table 3.3 give results from C2BIT+S+R, including total runtime (encoding+solving), number of refinement iterations, ratio of size(M_{abs})/size(M_B) and runtime speedup over the setup without abstraction. C2BIT+S shows great reduction on the memory usage and runtime compared to CBMC on both SAT and UNSAT properties. One main reason is due to our efficient modeling of array variables and aggressive slicing. C2BIT+S+R shows speedup compared to C2BIT+S on all SAT properties, because their satisfying solutions can be obtained in the under-approximate model with small encoding width.

For P2-P4 whose UNSAT cores are all small, C2BIT+S+R shows faster than C2BIT+S. For P1 whose UNSAT core is very big, M_{abs} is similar to M_B in both size and solving time, while C2BIT+S+R needs extra time of constructing the abstraction. For P5, although the size of M_B is 13 times bigger than that of M_{abs}, their solving time is similar, because MINISAT itself is efficient enough in searching relevant constraints among redundant ones in this case. For P6, C2BIT+S+R is slower than C2BIT+S, although the abstract model size is rather small. Since some constraints not in the UNSAT core may still be useful

Table 3.1: Results on bubbleSort program

Array Size	CBMC-slice CNF Size	C2BIT CNF Size	CBMC-slice Solving Time	C2BIT Solving Time
4	128	64	0.37	0.14
5	208	84	1.05	0.26
6	308	108	1.11	0.36
7	428	128	2.72	0.87
8	580	152	3.00	0.97
9	768	172	5.01	1.62
10	976	196	9.30	1.53
11	1204	220	7.64	2.90
12	1468	240	26.00	2.91
13	1744	264	16.55	4.75
14	2048	284	22.87	3.71
15	2372	308	36.29	5.10
16	2724	328	40.15	6.10
17	3116	352	60.88	7.27
18	3516	372	80.98	8.28

to assist the SAT solver to find conflicts quickly, the abstraction may increase the non-determinism by removing these constraints. Therefore, there is a trade off between removing constraints to reduce model size and reducing solving time when the UNSAT core is not very small. Note that for P9, the values in the braces indicate the results without the proposed improvement in computing updated encoding width that is even worse than without refinement. In summary, the slicing can greatly reduce the memory usage and runtime, and the refinement is efficient to further reduce the solving time in two cases. One is that the satisfying solution for SAT properties can be found in the small encoding range of the under-approximation. Another is that the UNSAT core is small for UNSAT properties.

Table 3.2: Comparison of runtime and memory cost of *CBMC* – 2.4, C2BIT with slicing and C2BIT with slicing plus refinement (Part 1).

	LOC	Property	CBMC		C2BIT with slicing C2BIT+S	
			Mem(kb)	T(s)	Mem(kb)	T(s)
P1	2411	UNSAT	2560	TO	1536	356
P2	4255	UNSAT	22928	1075	5764	411
P3	4255	UNSAT	22928	570	6508	151
P4	4255	UNSAT	22928	767	7236	44.3
P5	4255	UNSAT	22928	252	8496	14.7
P6	4255	UNSAT	22928	TO	2860	172
P7	4255	SAT	22928	TO	5012	119.9
P8	4255	SAT	22928	TO	9280	664.8
P9	260	SAT	941	45	561	7.5
P10	4565	SAT	23540	TO	11616	1233

3.5 Related Work

The application of model checking for software verification has been investigated in [8,26,35,43,70,109, 115]. Some apply symbolic model checking with predicate abstraction [8, 26, 70] or without abstraction [43, 109]. Others apply SAT-based bounded model checking to verify asserted safety properties of the program [35, 70, 115]. Our approach belongs in the second category.

Both Edwards's work [43] and the recent work of Wang [109] applied symbolic model checking approaches to verify the asserted property of the embedded software by building a finite states transition verification model. In [109], the authors exploited a unique feature of the software that program variables have a high degree of locality. They reported being able to directly handle software models having thousands of state variables without predicate abstraction. The major advantage of modeling software program as a states transition system is that, it is conceptually sound for proving the absence of bugs, even in programs with unbounded loops and recursions. However, due to the limited scalability, symbolic model checking is mostly combined with predicate abstraction, which transforms the program

Table 3.3: Comparison of runtime and memory cost of *CBMC* – 2.4, C2BIT with slicing and C2BIT with slicing plus refinement (Part 2).

	LOC	Property	C2BIT with slicing + Refinement (C2BIT+S+R)				
			Total T(s)	Iteration	Max S	Max (M_{abs}/M_B)	Speedup
P1	2411	UNSAT	343 + 152	1	4	1532/1536	0.72
P2	4255	UNSAT	**35.5 + 7.6**	1	4	1196/5764	>9
P3	4255	UNSAT	**41 + 8.5**	1	4	908/6508	>3
P4	4255	UNSAT	**15.3 + 9.6**	1	4	1088/7236	>1.5
P5	4255	UNSAT	10.8 + 9.2	1	4	612/8496	0.73
P6	4255	UNSAT	388 + 5.1	1	4	968/2860	0.44
P7	4255	SAT	**5.14**	1	4	–	>23
P8	4255	SAT	**45.79**	1	4	–	>14
P9	260	SAT	5.4(15.5)	2(2)	7	–	>1
P10	4565	SAT	29.5	1	4	–	>40

written in the high-level programming language to a simple Boolean program [8, 26, 70].

CBMC [35] is the first SAT-based bounded model checker for embedded software in C, to the best of our knowledge. It has been used for assertion checking, as well as equivalence checking to other hardware description language like Verilog. It does not document the scalability enhancement method, although the tool supports limited program slicing and translating the C program into the Satisfiability Modulo Theories (SMT) [80] format, which can facilitate using a SMT solver for verification. Saturn [115] applies BMC to detect hard errors, and uses function summaries, which are represented as finite state systems, to be scalable enough to handle inter-procedural calls. This kind of function summary targets on verification properties involving with few states, such as having only locked and unlocked two states in the lock management problem. F-Soft [70] supports both SAT-based BMC and BDD based unbounded MC. The translation from program to the Boolean model does not require the unwinding of loops, as their model is sequential instead of combinational. It makes use of predicate abstraction and counterexample-guided abstraction refinement technique to enhance the scalability of model checking.

3.6 Summary

In this chapter, we have presented a new approach that applies the aggressive program slicing combined with compiler optimization to compute an accurate slice. We also explored a proof-based abstraction-refinement technique on our software model to build a localization abstraction for further enhancing the scalability of software bounded model checking. Experiments show that our technique can achieve significant speedups compared to the conventional BMC tool.

Chapter 4

BMC via Symbolic Simulation

In this chapter, we explore symbolic simulation on the verification model used in Chapter 3 to achieve complete input coverage. As in previous chapters, abstraction is also used. We used a counterexample-guided abstraction-refinement procedure to automatically abstract the property-relevant constraints. Experimental results suggest that this approach is promising to formally verify some safety properties, using a small number of simulation runs with symbolic values.

4.1 Motivation and Overview

We aim to investigate the applicability of some approaches in the hardware formal verification to verify the embedded software. There has been a great deal of research conducted in this manner in recent years. For example, Counterexample-Guided-Abstraction-Refinement (CEGAR), first proposed for model checking hardware designs, has been increasingly used in software verification. Actually, software and hardware model checking have reinforced each other in recent years [34]. In addition, our research was also motivated by the experiences acquired from using CBMC (Bounded Model Checking of ANSI-C) [35]. CBMC was originally developed at Carnegie Mellon University with the aim of model checking embedded software. The basic idea is to transform the program under verification to a Boolean formula so that the property checking problem is reduced to searching for a satisfying assignment to the Boolean formula by a SAT-solver. However, the flattened Boolean formula loses the explicit data de-

pendency and the structural information in the program, which can be used to aid the search process. In addition, the complete transformation to the Boolean formula reduces the chance of applying abstraction techniques to improve the scalability of model checking, and using word-level decision procedures like SMT-solvers to enhance the searching ability of SAT-solver in software verification. (The latest version of CBMC supports dumping program constraints in the SMT format.)

The rest of the chapter is organized as follows. In Section 4.2, we present some preliminaries about symbolic simulation. In Section 4.3, we introduce our proposed symbolic simulation-based model checking. Experiment results are reported in 4.4, followed by the summary in Section 4.5.

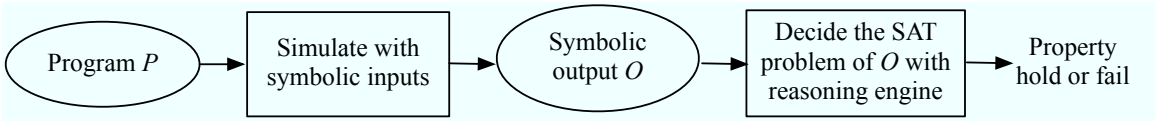


Figure 4.1: Basic symbolic simulation procedure

4.2 Preliminaries

4.2.1 Symbolic Simulation

Symbolic simulation is a rather broad concept and has become increasingly popular in the system verification [18, 114] and software testing and verification [3, 50, 74, 75]. The basic idea is that the program execution is conducted by assigning each input unknown symbolic values, instead of specific concrete values as the conventional software testing. The main advantage of symbolic simulation is that one symbolic simulation run can replace multiple conventional simulation runs. Formal verification requires proving that the software satisfies the specification for all combinations of input values. Symbolic simulation holds promise for this purpose, since a single symbolic simulation can capture multiple scenarios. For example, when verifying a 3-input *AND* function, one symbolic run with input vector $0XX$, where the logic value of symbol X is unspecified, is equivalent to four fully specified testing runs $\{011, 000, 001, 010\}$, as long as the output values obtained from simulating $0XX$ do not contain any unknowns. In this particular example, the output of the AND gate is 0 for all four vectors. Figure 4.1 shows the flow of a general symbolic simulation procedure. Given a program P, we apply symbolic

inputs to perform symbolic simulation, and we obtain the symbolic result *O*. We then use some decision procedure to reason the SAT problem of *O*, so as to conclude that the property being verified holds or fails in *P*.

Broadly speaking, our model construction process introduced in Chapter 3 is a kind of symbolic simulation. As in Figure 4.2, the symbolic values enforced on program inputs have symbolic names, which are the same as the corresponding program variable names in this case. After symbolic execution, we can obtain the symbolic output value as a first-order formula using nested ITEs. This symbolic simulation approach is precise; however, as the program becomes larger, the number of paths may increase exponentially. As a result, the terms representing the symbolic value may easily blow up [103]. Although some research has been conducted to alleviate this problem, such as term rewriting [4, 42] and ITE simplification [88, 92], it is still one of the obstacles in the scalable applications.

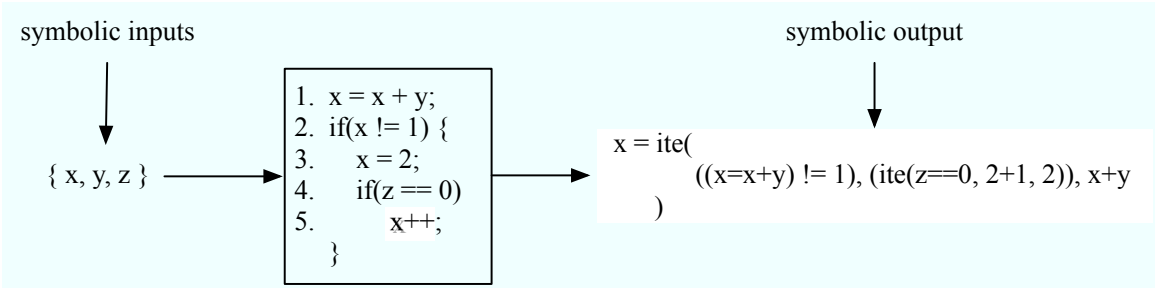


Figure 4.2: An example of symbolic simulation

In this work, we apply another kind of symbolic simulation [56]. We assume only a small number of bits in the input variables' bit-vectors need to be specified to verify program invariants in the verification problems. Therefore, rather than using OBDD-based symbolic simulation, which describes the accurate program functionality with all input bits, we use distinguishing Xs introduced in [72] as symbolic variables for simulation. It distinguishes every don't-care (X) with a unique ID. Every X_i has a correlated X_j. In particular, an odd id *i* has a correlated id $(i-1)$, and an even id *j* has a correlated id $(j+1)$. The cost of simulating distinguishing Xs is almost the same as the traditional 3-value simulation, and it has a strong reasoning ability among correlated Xs. Figure 4.3 gives examples to show the difference between simple X, distinguishing X and BDD. Consider the NOT gate in Figure 4.3(a) and (b), a distinguishing X uses an even id 0 and its incremented id 1 to record the negation relation of the two Xs (if input X with odd id, the output id is its decrement.), but simple X loses this correlation. For the example cases

of AND gate, simple Xs cannot differentiate two related Xs on an AND gate. For distinguishing Xs, it can derive the output value 0 if two inputs have negation relation, and the output equals one input if another input has non-dominant value (i.e., 1 in this case). But if two inputs have no correlation, distinguishing X uses a new id to label the output. Finally, BDDs can be used to capture the full functionality of AND logic. However, they may require excessive memory space for complex functions. In terms of the accuracy of symbolically representing the functionality, BDD is the most accurate, while 3-value logic values is the least accurate, distinguishing X lies in between.

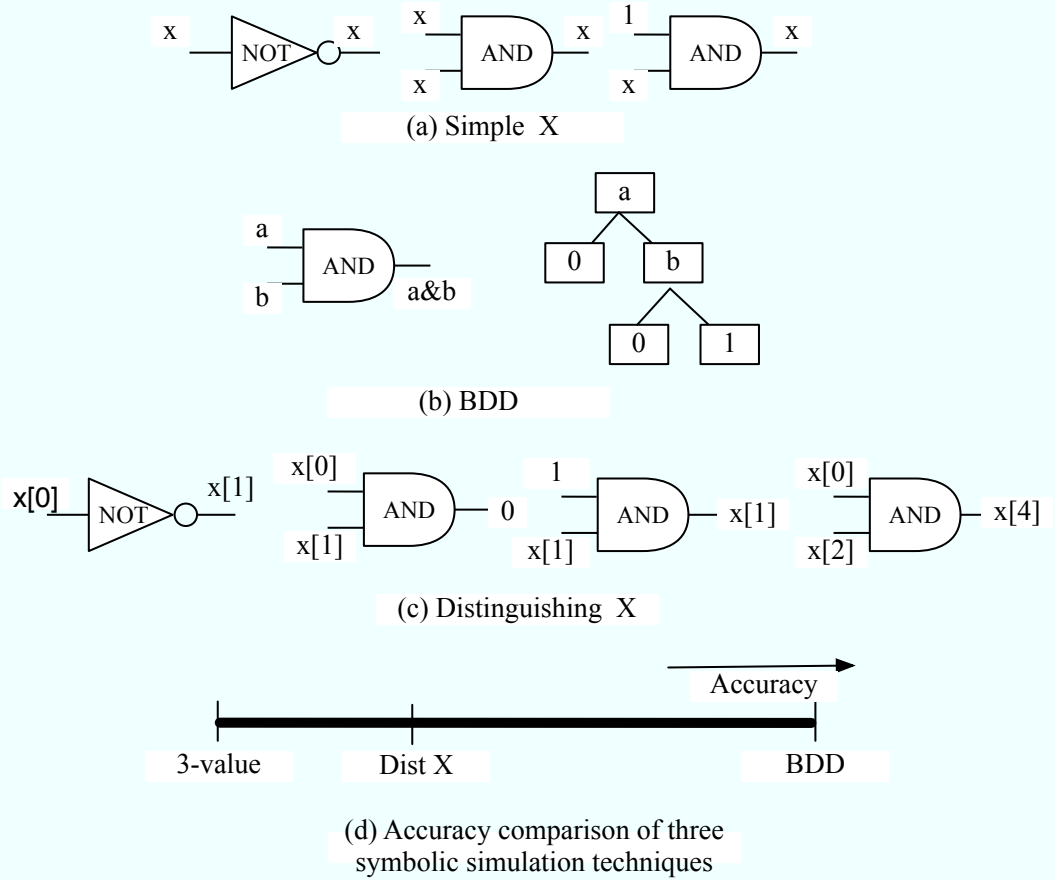


Figure 4.3: Example of simple X, BDD, Distinguishing X and accuracy comparison.

In our approach, we enhance distinguishing Xs by enabling it to reason the consistency among the same type of the un-interpreted functions (UFs) [21]. Recall that the UF operator loses all semantics of the underlying functions, except the functional consistency among different instances of the same function. So the computational constraints between inputs and output of each individual UF are abstracted away.

Figure 4.4 gives an example of how the correlation among input Xs are propagated through UF operators. The two UF operators are abstracted from the same operation type (e.g., "$+$"), their input pairs are both $(X[m], X[n])$ although the exact values of $X[m]$ and $X[n]$ are unknown, so their outputs are both $X[k]$. We can easily conclude that the output of the connected $==$ operation is always 1. In this case, we can only get the inconclusive X at the output with the simple Xs simulation.

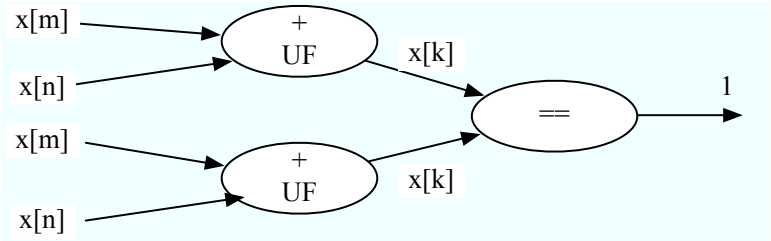


Figure 4.4: An example of UF functions simulated with distinguishing X

4.3 Model Checking in CEGAR Framework

The model construction is similar to that in Chapter 3. So we focus on the model checking procedure. Given the program models M_H and M_B, we first choose an initial abstraction M_γ. Then, we model check M_γ with the symbolic simulation. If M_γ holds the property, the concrete program also holds the property, since M_γ is an *over-approximation based abstraction*. Otherwise, we could get a counterexample, which violates the property. The counterexample E is a set of value assignments on some internal and input edges. We validate the counterexample by justifying these value assignments in the low-level model M_B. If E is valid in M_B, we can conclude that the program violates the property; otherwise, we restore the constraints along the internal edges in E to remove the invalid counterexample for refining M_B, and jump to the step of model checking for the next CEGAR loop. The CEGAR procedure with these four steps is illustrated in Figure 4.5. We describe the first two steps with more details in the following.

- **Choose an initial abstract model** M_γ

  The selection of M_γ involves a trade-off between the model checking cost and the model refinement cost. If M_γ is too coarse, the cost of removing false negatives through the refinement steps can be very high. On the other hand, if it has a similar complexity to the original program, the

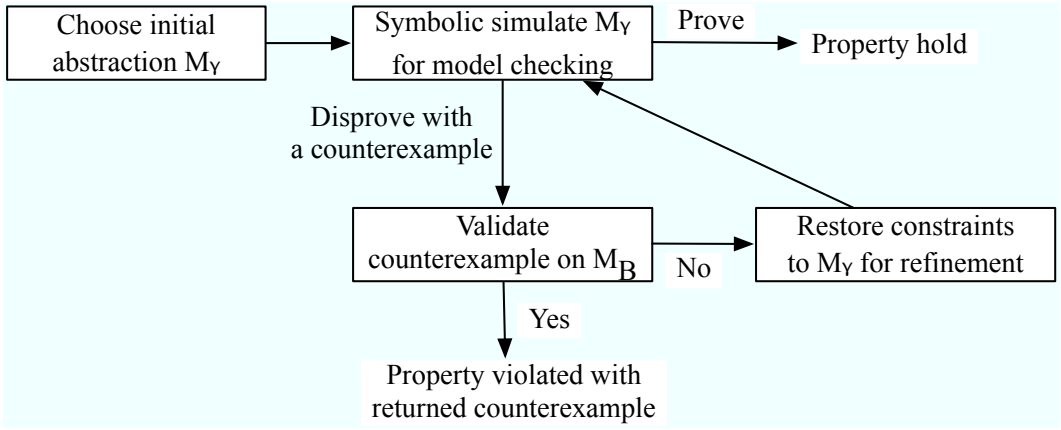


Figure 4.5: CEGAR procedure with symbolic simulation

model checking operation would be costly. In order to find the suitable initial abstract model, we take the following steps.

1. We make use of the equality assumptions in the program to increase the chance of reducing the redundant variables. These assumptions claim the conditions that must be satisfied for the execution to reach the property assertion being verified. For example, if a property assertion is located within the control range of an equality condition $(a == b)$ as shown in Figure 4.6 (a), we could replace all the usage of either a or b after the last definition with the other. If one variable is the primary input without new definition in the program, we could safely remove it to reduce the number of variables as variable b in Figure 4.6 (b). This transformation can reduce the model complexity without invoking false negative errors; it can be easily implemented in the model M_H.

2. Following the last step, some variables that have the same type of operation may share the same inputs. So the nodes corresponding to these variables in the model can be merged as the variable y in Figure 4.6(c).

3. We remove the variables with complex arithmetic operations like multiplication and division, replace other arithmetic operations like addition with un-interpreted function (UF), and keep the variables with predicate operations. For each removed node, we also remove its data dependencies with its input nodes and its data dependencies with its output nodes. In some cases, the model at this step contains only the program constraints defined in the

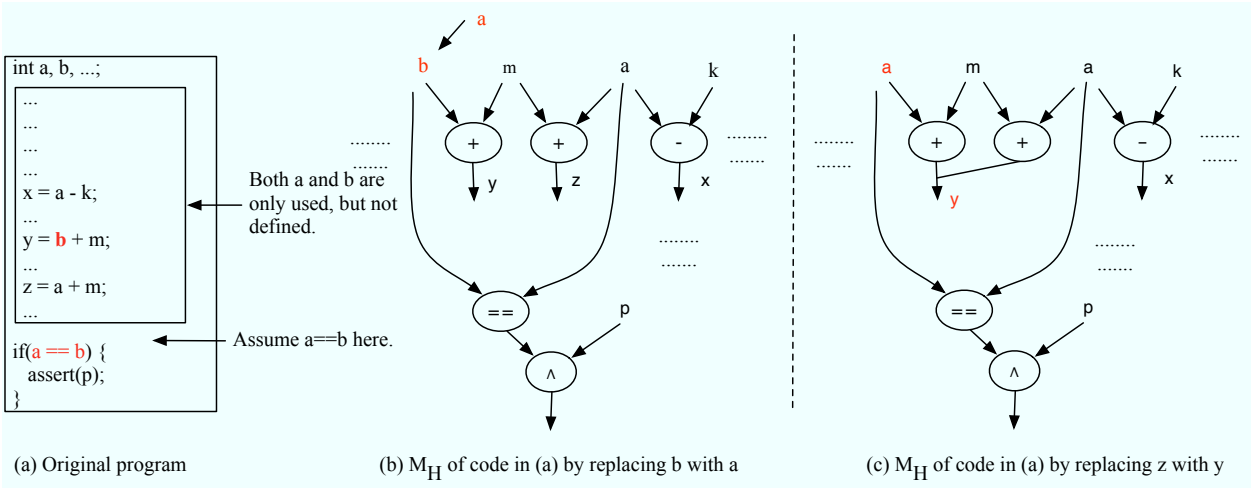


Figure 4.6: An example of model reduction

property assertion. In order to avoid the abstract model M_γ being too coarse, we restore the variables that have the data dependency with the assertion statement at the distance S, where S is usually set to 3. Finally, the model we get is the initial abstract model M_γ.

- **Model checking by symbolic simulation with Distinguishing Xs**

  We symbolically simulate the abstract model M_γ to verify whether its output node is an invariant under all possible input values. Our goal is to implement the complete coverage through a small number of symbolic input vectors. There are three general cases in which the properties are formally verified with symbolic inputs.

  1. One is using distinguishing Xs without specifying any input bit value to constant “1” or “0”, so that the size of V is 1. This is easy to conduct.
  2. Another is constructing V by the enumeration of all specified logic values on only a small subset of input bits and leaving rest bits as X, for example $\{XX1X, XX0X\}$. This could abstract away the variables which are not relevant to the verification.
  3. A third option is the identification of a set of limited input patterns to build V, for example $\{0001, XXX0\}$. This relaxes the restriction of the previous case on enumerating the value of only certain input bits, but still with a small number of input vectors.

  We use the value justification to realize cases 2) and 3) [96]. The basic idea is that the input values

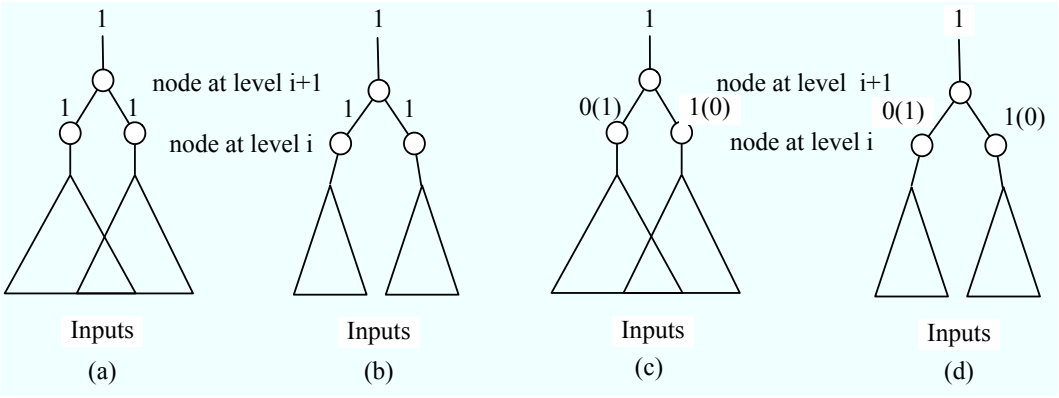


Figure 4.7: Four scenarios of input space partition

of a node can be justified by the specified output value according to the operator type, so that the whole input space can be partitioned. Figure 4.7 shows four scenarios. In scenarios (a) and (b), in order to verify the node at the level $i+1$ who has an *AND* operator as "$1''$, both edges connected with the nodes at the level i must be justified as "$1''$. In parts (c) and (d) of the figure, since the node at the level $i+1$ has a *OR* relation with the input nodes at level i, only one edge connecting with nodes at the level i needs to be justified as "$1''$. The subset of input bits, on which each node output at the level i depends, may be correlated as in scenario (a) and (c), or may be uncorrelated as in scenario (b) and (d).

For scenario (a), we give the correlated input bits higher priority for specifying their values or searching their patterns. For scenario (b), we could separate the symbolic simulation into two independent tasks, and each with a smaller input space. For scenario (c) and (d), the manner in which the pattern is searched, depends on the justification order among the optional edges. Using the value justification to partition input space into smaller subspaces, we can have the better chances of obtaining the symbolic input patterns, which may avoid the don't-care value at the monitor output, so as to relieve the conservativeness of symbolic simulation.

We set the threshold of the size of V as 32. So the number of value-specified input bits is 5 for case 2). In fact, the case 3) is a complement of case 2). With the set V, we symbolically simulate the abstract model M_γ for model checking. The pseudo code of our simulation is given in Figure 4.8.

```
1. Procedure SymSim (Abstraction M')
     /*levelizing nodes*/
2.     level(Input Nodes) = 0;
3.     for each node n
4.       level(n) = max(level (n's predecessor))+1;
5.     end for
     /*symbolic inputs with 0, 1 and distinguishing X[id]*/
6.   for each input vector V
7.    for level l = 0 to max level
8.       for each node n with level(n)==l
9.           evaluate(n);
           end for
         end for
       end for
   end
```

Figure 4.8: Pseudo code of proposed simulation

4.4 Primary Experimental Results

Currently, we implemented our approach in the prototype C2CKT with C++. For the comparison purpose, we also implemented in the prototype C2CNF, the transformation from the program to CNF formula for bounded model checking by the SAT solver, which was proposed in CBMC [35]. The programs that we experimented with have only integer type variable. Table 4.1 gives the basic analysis of the test results on the five C experimented programs.

Table 4.1: Test results

	BB# in SSA	*SS* by case 1	*SS* by case 2,3	CEGAR Loop#	#input patterns (>32)
Tut4.c	4	Y		0	N
Tut6.c	5		Y	1	N
Lock.c	12	Y		0	N
Ineq.c	5			0	Y
Date06.c	3	Y			N

In Table 4.1, *SS* means the symbolic simulation for short. The 2nd column describes the property of the C benchmark program, the 3rd and the 4th columns record the case type, by which the programs are

verified. The 5th column is the number of CEGAR loops being taken. The 6th column records whether the program is completely verified with the small threshold of 32 symbolic input patterns. For example, consider the benchmark program Tut6.c. There are 5 BBs in this code, and its verification falls in case 2,3) with one CEGAR refinement iteration. The required input patterns to cover the complete input space are smaller than 32 (actually 5 patterns in this case). Only one benchmark program Ineq.c is not verified under this threshold.

Table 4.2 shows that the size of V does not change with the width of the bit-vector for data representation, but the size of CNF increases as the width of the bit-vector increases. This reason is that our approach aims to find input patterns that can cover whole search space. The size of these patterns is not directly dependent on the integer representation, like the size of CNF clauses in the C2CNF approach. For example, with the modeling width of bit-vector increased from 4 to 16, the number of patterns in the set V is still 5, but the number of clauses in CNF is increased to 1143.

Table 4.2: Comparison between C2CKT and C2CNF in verifying Tut6.c

Size of bit-vector	Size of V	# of clauses in CNF
4	5	763
16	5	1143
32	5	2383

4.5 Summary

We investigate the potential of combining symbolic simulation with localization abstraction [78] for scalable formal verification. Our approach uses distinguishing X as symbolic values to replace the concrete variables' values, so that a limited number of identified symbolic input vectors can cover the complete input space. In order to reduce the verification time and ease the identification of symbolic input vectors, we apply the symbolic simulation to check the abstract model, which is automatically constructed in the counterexample-guided abstraction-refinement framework. This abstract model is a localization abstraction of the program, which includes a subset of property-relevant program constraints, and so it is more easily verified than the original program. The property-relevance of program

constraints is automatically determined during the iterative abstraction refinement procedure.

Chapter 5

A New Testability Guided Abstraction to Solving Bit-vector Formulae

In this chapter, we present a new abstraction approach based on the concept of under- and over-approximation, to efficiently solve bit-vector formulae generated from software verification instances, which include intensive control structures. Our proposed approach applies two common testability metrics — controllability metric (CM) and observability metric (OM) — for guiding the abstraction refinement procedure. We construct the under-approximation by enforcing constant constraints on a small set of single-bit variables, which control the branch selection of some *ITE* variable nodes. Subsequently, each constructed under-approximate model includes only a subset of paths in the formula. We use CM and OM to guide building such models so that a counterexample can be obtained with little effort. If the under-approximate model is unsatisfiable, an over-approximate abstraction is obtained by refining along the paths included in the model. This is conducted by using the UNSAT proof to learn the relevant formula constraints. We also use OM as a guide to heuristically and efficiently restore additional verification-relevant constraints in the iteration. The experimental results show a significant reduction of the solving time compared to state-of-the-art solvers for bit-vector arithmetic.

5.1 Motivation and Overview

Many existing software verification techniques [5,35,37,110,115], model the bounded data-types as bit-vectors with fixed bit-widths, and the verification constraints are modeled as the bit-vector arithmetic formula. For example, in the embedded software model checking tool called CBMC [35], the program verification instances derived from the bounded model checking of ANSI-C programs are formulated as bit-vector equations. Another example of an application can be found in an automatic theorem prover called Cogent [37]. It has been applied to precisely reason the satisfiability of the queries produced in the predicate abstraction refinement procedure for symbolic software model checking, like in SLAM [9] and ComFoRT [71]. This bounded modeling allows the bit-precise reasoning to be applicable to almost all programming language constructs. However, due to the program complexity, the existing decision procedures are still not scalable to directly reason the validity of the bit-vector formulas, obtained from the practical software verification without aggressive abstraction.

Compared with other abstraction techniques like predicate abstraction, weakening abstraction is simple and easy to apply since it weakens the transition relations without mapping the state space of the design to another state space domain. Several existing abstraction methods, such as those proposed in [5, 20], are considered to be weakening abstraction. In order to make full use of the ease and simplicity of weakening abstraction, it is critical to develop a refinement procedure that can efficiently and accurately identify the verification-relevant variables to construct the precise abstract model.

The goal of our work is to apply a weakening abstraction technique — localization abstraction to quicken the decision procedure of bit-vector formulas, especially those generated from the verification of control-dominated software properties. Our contributions are as follows:

1. We propose a new localization abstraction procedure using the concept of under- and over-approximation. The under-approximate model is built by enforcing constant constraints to a small set of single-bit variables that control the branch selection of ITE nodes. This restricts the search space to only a subset of paths in the formula. The over-approximate abstraction is obtained basically via learning the relevant variables from the UNSAT proof of the under-approximate model. To start a new refinement iteration, we use the satisfiable assignment of the current over-approximate abstraction,

to guide the construction of a new under-approximate model, with an unvisited subset of paths in the formula.

2. We present a new CM and OM computation and apply them to efficiently guide the proposed abstraction refinement procedure. CM estimates the ease of finding a SAT solution by the SAT solver on the paths reaching the target variable V. It is different from traditional CM metric, which evaluates the testability of a structure to guide the *simulation-based testing*. Our proposed OM approximates the influence of the variation of V on the outputs. Our approach can quickly find a satisfying solution on the easily controllable subset of paths in the formula, as long as a solution exists on them. It also restores additional verification relevant constraints, according to the OM heuristics in the iteration, to significantly reduce the refinement cost by only using the UNSAT proof. As a result, an abstract model that is precise enough for the verification can be quickly built, long before all paths have been enumerated.

The experimental results show that a significant amount of solving time can be reduced for the benchmarks generated from the bit-precise software verification applications.

The rest of the chapter is organized as follows. In Section 5.2, we present the controllability and observability computation. Our proposed abstraction approach is presented in Section 5.3. We report our experimental results in Section 5.4. A discussion of related work is given in Section 5.5, followed by some conclusions in Section 5.6.

5.2 Controllability / Observability Metric

In order to quantifiably estimate the amount of influence that each bit-vector variable has on the property under verification, we propose a new method to compute CM and OM of each variable in the formula. CM and OM respectively approximate the difficulty of finding a solution on the paths leading up to the variable, and the amount of impact that the variable has on the target property at the output.

One informal definition of the testability of a variable in a design is: the degree to which the variation on that variable can be controlled from the inputs or observed at some observation point. CM and OM are

two popular metrics widely used to evaluate the testability of hardware designs at either the RTL level or the logic-level. Examples include the test case quality evaluation [46], identification of hard-to-test regions in a design, where faults can hide from /easily be reached by testing [27, 51, 55, 64, 65, 106], and testability analysis of software components [93].

One way to view formal verification is "testing with complete input-coverage". Thus, CM is useful to provide some high-level structural information to guide the search of a SAT solution with different expectations. For example, we can focus on the easy-to-control paths if we want to test those common cases; or focus on the hard-to-control paths if we aim at the corner cases where the random/directed testing have difficulty with. Furthermore, the hard-to-observe variables are less likely to be relevant to the verification property, as variations of their values have the limited impact on the property. In this regard, OM provides a guidance to estimate whether a given variable should be involved in the abstraction or not.

Our proposed CM and OM computation is performed on the bit-vector variables directly, instead of the individual bits. It also makes use of the pre-calculated controllability and observability coefficients (COC) of basic operators, to efficiently compute the CM and OM of each individual variable computed by that operator. These coefficients approximately reflect the different amounts of influence that the basic operators have on the CM and OM computation. We apply the graph-based model of the bit-vector formula to facilitate the computation.

5.2.1 Controllability and Observability Coefficient

We first present how we estimate *Controllability and Observability Coefficient*(COC) of basic operators. Since we make use of the SAT-solver to finally determine the satisfiability of the bit-vector formula being solved, the number of clauses encoded from the bit-vector arithmetic operator is a good indicator of its complexity in searching for a solution. The adder is the basic element of almost all predicate operators and arithmetic operators in our encoding, and 32-bit is the most widely used width to represent program variables. Therefore, we use the 32-bit adder as the standard and set its COC value as 10.

The COC values for other operators (*OP*s) are computed, based on the number of encoded CNF clauses

and bit-vector width. It is formulated as the following (For all non-predicate operators, K is the output bit vector width of OP; for predicate operators, K is the maximum of inputs bit vector width of OP):

$$COC(OP) = \lceil \frac{\#\ of\ CNF\ clauses(OP_{32})}{\#\ of\ CNF\ clauses(ADDER_{32})} \times 10 \times \frac{K}{32} \rceil$$

Table 5.1 lists the COC values for some commonly used bit-vector operators.

Table 5.1: COC Values for Common Operators

Bitwise OPs	
OR, AND	2
XOR	2.5
Predicate OPs	
==, $\neq$, =¿	3
$>, <$	9
Arithmetic OPs	
+	10
-	11
Others	
$\geq, \leq$	12
ITE	2.5

5.2.2 CM and OM Computation

The CMs of all primary inputs (PIs) are first initialized to zero. Then, since the CM of an internal variable depends on the CM of its inputs and the type of the operator by which the variable is computed, we formulate the relationship between the CM of any variable v with the operator op and the CMs of all its inputs IN as the following (where N is the number of inputs of v):

$$CM(v) = max\{CM(IN_i)\} + COC(op), (0 \leq i < N)$$

After computing the CM values of all variables in the graph-based model of the formula, we begin to calculate the OM value of each variable backwards from POs. The OMs of the one or more POs that correspond to the properties under verification, are first set to zero. As the observability of an input of any variable v depends on both the observability of v and the controllability of any other inputs to v, we compute the OM of the input variable IN_i as the sum of the COC value of v's operator op, and the maximum value among the CM of all side inputs of v and the OM of v as:

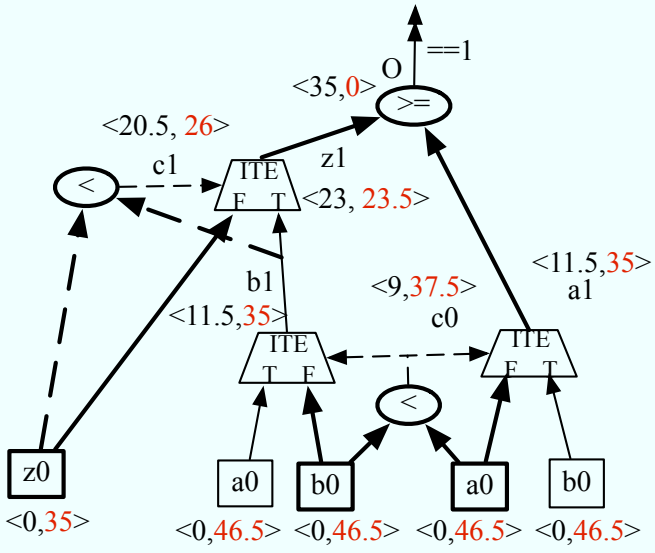
$$OM(IN_i) = COC(op) + max\{\begin{matrix} OM(v) \\ max\{CM(IN_{j\neq i})\} \end{matrix}\}$$

```
S1:  if(b0 < a0) { b1=a0; a1=b0; } else { b1=b0; a1=a0; }
S2:  if(z0 < b1) { z1=b1; } else{ z1=z0; }
S3: assert(z1 >= a1);
```

(a) Program statements in SSA

F = (c0 = (b0 < a0)) ∧ (c1 = (z0 < b1)) ∧ (b1 ite c0 a0 b0) ∧ (a1 ite c0 b0 a0) ∧ (z1 ite c1 b1 z0) ∧¬(z1>=a1)

(b) Bit-vector formula F generated from (a)



(c) Graph-based model of F labeled with <CM, OM>

Figure 5.1: An example of bit-vector arithmetic formula with its Model

In Figure 5.1, we use a bit-vector formula generated from a small program verification instance as an example of the CM and OM computation on its graph-based model. Figure 5.1(a) gives a simple sorting code with two statements in the Single Static Assignment (SSA) form and one property assertion at statement $S3$. In $S3$, $b0 < a0$ and $z0 < b1$ are two control predicates determining the execution path.

In the corresponding generated bit-vector formula F given in Figure 5.1(b), the program control flow information is decomposed by a set of ITE operators, each of which has a selection input *iSel* to control which of the two inputs to propagate to the output of the operator. *iSel* represents the conjunction of all control predicates, which must be satisfied to take the new variable definition on the true branch input. All variable definitions in the same control range share one *iSel*. For instance, as $b1$ and $a1$ are all under the control of predicate $b0 < a0$ in Figure 5.1(a), $c0$ defined as $b0 < a0$ is the *iSel* of two ITEs $a1$ and $b1$ shown in F. Thus, enforcing constant constraints on *iSel*s of F can be regarded as restricting the execution on a subset of control-flow paths in the program.

In the graph-based model M of the formula F shown in Figure 5.1(c), we label the computed CM and OM in the pairs next to each variable node. As an example, the CM of the output O is computed as

$$CM(O) = max\{CM(z1), CM(a1)\} + COC(O) = 23 + 12 = 35$$

Computation of OM begins after the CM values for all variables have been computed. The OM of $a1$ is calculated as

$$OM(a1) = max\{OM(O), max\{CM(z1)\}\} + COC(O) = 23 + 12 = 35$$

For the fanout structure in the graph, where one variable has more than one output connection, the OM of this variable may be computed several times. We choose the maximum one as the final OM of this variable. For the example of $z0$, which has two fanouts $c1$ and $z1$, $OM(z0)$ computed from $c1$ and $z1$ is 35 and 26. So $OM(z0)$ is set to 35.

In summary, the CM/OM has three important properties:

- Larger values of the CM/OM indicate harder controllability/observability.

- On any path from PIs to a PO, $CM(v_1) < CM(v_2)$ and $OM(v_1) > OM(v_2)$ always hold if v_1 is the predecessor of v_2.
- For fan-out free inputs $IN_0, \ldots, IN_{N-1}$ of v, $OM(IN_i) \geq OM(IN_j)$ if $CM(IN_i) < CM(IN_j)(i \neq j)$.

5.3 Testability Guided Abstraction and Refinement

In this section, we present our new testability guided abstraction and refinement approach to efficiently solve the bit-vector formulae. We first give an overall framework of our approach. Details regarding the steps of choosing *iSel* variables, deciding enforced constant constraints to build the under-approximate model and constructing the over-approximate abstraction, are described in later subsections.

5.3.1 Overall Framework

Given a bit-vector formula Φ, our proposed approach has four steps as illustrated in Figure 5.2. Steps $2-4$ are iterative steps during the refinement.

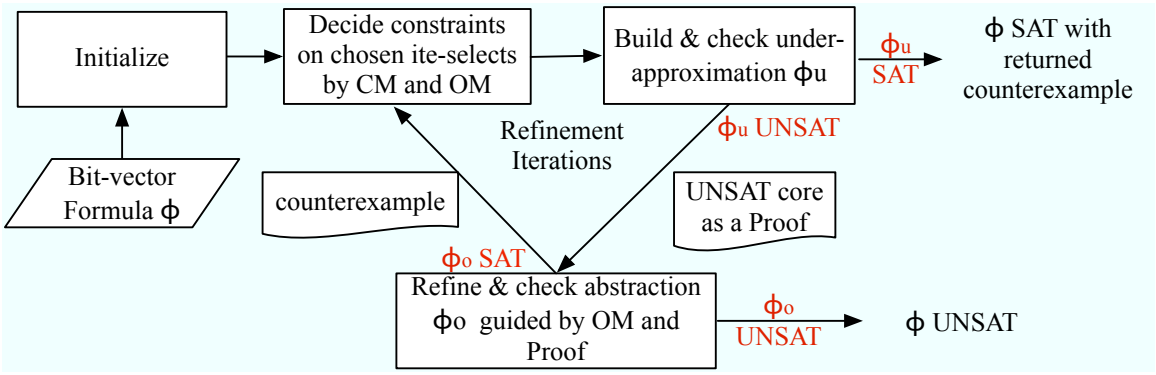


Figure 5.2: Overview of testability guided abstraction approach to solving Bit-vector formula

1. *Initialize:* We first build the graph-based model M of Φ, and calculate the CM and OM of every variable node in M, according to our proposed computation method discussed in the previous section. We also define some data sets based on Φ for the abstraction:
   - I: a collection of all the ITE variables.

- *IS*: a collection of all the *iSel* variables, each of which controls the branch-taking of at least one ITE variable.
- IS_S ($\subset IS$): a collection of chosen *iSel*s to which the constant constraints (0 or 1) are enforced to build the under-approximate model.

Enforcing all enumerated combinations of constant constraints on the *iSel*s in the set IS_S, is actually the act of partitioning of all control-flow paths of Φ into a number of subsets of paths, each of which is an under-approximation. This is because the search space is limited to only one branch of all ITEs controlled by the constrained *iSel*s. In order to minimize the similarities among the enumerated partitions and the complexity of each partition, we set two basic criteria of choosing *iSel*s into the set IS_S. Recall that multiple ITE operators may be controlled by the same *iSel* variable. So first, we prefer choosing those *iSel*s with the largest number of connected ITEs. Second, considering the different degrees of verification relevance of variables, we prefer selecting the *iSel*s whose connected ITEs are most relevant to the property under verification.

2. *Decide 0 or 1 for each iSel constraint:* For each *iSel* in IS_S, we decide which value (0 or 1) to enforce for building a Φ_u. First, we must ensure that the enforced constant constraints have not been applied before, to avoid building the same Φ_u again. Second, we give the priority to the ones that can restrict the search on the branches of the controlled ITEs with smaller CM values. Thus, Φ_us consisting of a set of easy paths according to CM are constructed early. This facilitates the finding of a SAT solution with less effort. Furthermore, since these Φ_us usually involve either fewer variables or simpler formula constraints among the variables in Φ, it may also ease the UNSAT proof extraction for refining the abstraction.

3. *Build and check under-approximation Φ_u:* The chosen constant constraints on the *iSel*s in IS_S are added to the formula Φ to build Φ_u. Since the variables on the unselected branch of each ITE are irrelevant to reasoning the validity of Φ_u, we can safely slice them away. After encoding Φ_u into a Boolean formula β_u, we check the satisfiability of β_u with a SAT-solver. If it is SAT, we can conclude that Φ is also SAT, because the constant constraints added on the variables in IS_S are also achievable in Φ. On the other hand, if β_u is unsatisfiable, we use the UNSAT core C generated from β_u, as a proof to help us obtain an over-approximate abstraction Φ_o in the next step.

4. *Refine and check abstraction* Φ_o*:* In the case that Φ_u is unsatisfiable from the previous step, we restore the verification relevant variables to refine Φ_o in two ways. First, we identify *all* bit-vector variables V_H involved in the given UNSAT core C and add them into Φ_o. Note that the constant constraints enforced on the *iSel* variables in Φ_u are not carried over, so that we can make sure that Φ_o only contains the original formula constraints of Φ. Second, considering that the variables with similar OM values have similar impact on the property, we can heuristically restore more verification relevant variables, guided by the OM values of variables that are already learned as being relevant. Thus, after computing the maximum OM_{max} of the variables in V_H, we use OM_{max} as a threshold to restore more variables, whose OMs are equal to or smaller than OM_{max} into Φ_o. Thirdly, we check the satisfiability of the Boolean formula β_o encoded from Φ_o. If β_o is UNSAT, we could conclude that Φ is also UNSAT; otherwise, we use the SAT assignments on the *iSel*s from the set IS_S to guide enforcing new constant constraints in step 2 iteratively.

Due to the finite search space, the refinement procedure always terminates. In the worst case, the number of iterations can be exponential to the size of IS_S as the explicit enumerations of enforced constant constraints are exponential. However, earlier enumerations can actually be sufficient to refute a large number of paths that need not be enumerated in subsequent iterations. According to the experimental results, many properties can be verified, with only a very small number of iterations assisted by the proposed CM and OM guidance.

5.3.2 CM/OM Guided Under-approximation

As discussed earlier, we prefer choosing those *iSel* variables that control a large number of verification-relevant ITEs. Thus, we define a parameter *Priority* for each *iSel* to quantify this preference. In order to differentiate the verification relevance of ITEs, we classify all ITEs into K subsets $I = \{I_1, \ldots, I_K\}$ depending on their OM values. Let OM_{max} and OM_{min} be the *Maximum* and *Minimum* OM of all ITE nodes. The OM value of every ITE classified in the subset I_i $(1 \leq i < K)$ must fall into the interval $[OM_{min} + \frac{(OM_{max}-OM_{min})*(i-1)}{K}, OM_{min} + \frac{(OM_{max}-OM_{min})*i}{K})$. Every ITE in the subset I_K has OM in the range $[OM_{min} + \frac{(OM_{max}-OM_{min})*(K-1)}{K}, OM_{max}]$. A *weight* with the value of $(K-i)$ is assigned to all ITEs in each subset I_i $(1 \leq i \leq K)$ to approximately quantify their verification relevance. The computation of

Priority is formulated as the following (where $1 \leq j \leq Size(IS)$ and $ITE\_count(j,i)$ returns the number of ITEs in the subset I_i controlled by $iSel_j$):

$$Priority(iSel_j) = \sum_{i=1}^{K}((K-i)_{weight} * ITE\_count(iSel_j, i))$$

We then define a tuple T = (ID, *Priority*, $< CM_T, CM_F >$, L) to describe an *iSel*. Each *iSel* has a unique *ID*. The third element records the *average* CM of the True and False branches among all controlled ITEs. Due to the existence of embedded control structures like $if(c1)\{..;if(c2)\{..\}\}$, some *iSel*s may be the conjunction of other *iSel*s in the formula generated from such instances. (We assume each control predicate in the program has only one predicate operator). In order to express this dependency among *iSel*s, we use L to record the list of *iSel*s that include this *iSel* in their conjunctive form.

With the defined tuple T and a given threshold θ, we choose all *iSel*s each of which has the Priority P above θ, and store them in the decreasing order of P into the set IS_S, i.e., $(P(iSel_i) >= P(iSel_j))$ *where* $(1 \leq i < j \leq Size(IS_S))$. The order of enforcing constant constraints on the *iSel*s in IS_S follows the two basic principles:

1. To build the initial Φ_u, we prefer the constant 0 to 1 if CM_T of the *iSel* is greater than CM_F (*False* input branch is more controllable); otherwise, 1 is preferable. In the special cases that the L of the *iSel* is not empty, we prefer enforcing the constant 0, which means that the enforced constant constraints on all other *iSel*s in L have to be 0 as well. As a result, the initial model includes the easily controllable paths with a low computation complexity.

2. To build a new Φ_u during the refinement iteration, we only modify the enforced constant values for the *iSel* variables involved in the current UNSAT proof. We prefer that the enforced constant constraints can produce a new Φ_u with the *least* similarity to the current Φ_u. These enforced constraints must be valid in the abstraction Φ_o.

5.3.3 OM Guided Abstraction Refinement

If Φ_u yields no solution, the SAT solver (like [14]) extracts an UNSAT core C from β_u and returns a set of bit variables V present in C. With V, we learn those verification-relevant variables guided by OM to

```
Over_appr  (set V of UNSAT core C )
  /*1. Learn new constraints from V */
1:  for each bit-vector variable i in formula ϕ
2:    Map to a set V2 of bits in β that encode i;
3:     /* the intersection set of V2 and V */
4:     Compute set V3 = V2 ∩V ;
5:     if V3 not Empty
6:        Add variable i to ϕo;
7:        Remove all bits in V3 from V;
8:        Mark the variable i;
  /*stop if all bits in V have been included in ϕo*/
9:     if V empty                break;
10: end for

/*2. Learn more constraints guided by OM */
11: OMmax = maximum OM of all marked variables in ϕ;
12: for each bit-vector variable i in ϕ
13:    if i marked               continue;
14:    if OM(i) <= OMmax    Add variable i to ϕo;
15: end for
```

Figure 5.3: Abstraction refinement algorithm

obtain a refined abstraction Φ_o in two steps. The algorithm for the refinement is given in Figure 5.3.

First, to facilitate learning those relevant variables from V, a bit-vector variable and the original formula constraints with all its input variables in Φ are added in Φ_o, if any bit of this variable is included in V. Therefore, this Φ_o only contains a subset of formula constraints of Φ without the enforced constant constraints included in Φ_u. For the example shown in Figure 5.1, the under-approximate model built by constraining $c0 = 0$ and $c1 = 0$ involves the same set of variables as the original formula shown in the dark color of Figure 5.1(c) except that there is less control dependency among them. The abstraction built by the proof of such an under-approximation has all verification-relevant constraints (same constraints as the original formula), so we only use *one* out of all four possible combinations of added constraints to learn enough relevant variables for the abstraction in one iteration. This example illustrates that the proof-based refinement iterations need not enumerate all combinations of the enforced constant constraints to reach a conclusion.

Second, in order to reduce the burden of proof-based abstraction, we also apply OM as a guide to heuristically restore more verification-relevant variables in each refinement iteration. If a variable v

with OM equal λ is identified relevant by the proof, other variables whose OM is no bigger (i.e., equal or more observable) than λ are also considered relevant by the observability heuristic. This is reasonable because v is considered to have equal or greater impact on the target output. One explanation of why these variables were not in the proof of the current under-approximate model ϕ_u is that they may not have been needed in the restricted path(s) of this ϕ_u. But they may be referred to in the proof of ϕ_us that have not been enumerated. Furthermore, this OM-guided approach can identify these relevant variables with very little effort. It can be very efficient in the cases that we use the UNSAT proof to learn relevant constraints from the easy under-approximate model (partitions) of ϕ, and apply the guidance of OM to learn additional constraints in the other portions of ϕ that are complex and have not been involved in the previous enumerated partitions. So, it can increase the likelihood that the verification task is done without actually building the under-approximate models on those complex partitions. The third property of our CM/OM computation helps to increase the probability of such cases. It means that the branch with the smaller CM has the bigger OM value. So, if the variables on the easy-to-control branch are found to be included in the UNSAT core, the variables on another branch are also considered as relevant according to our OM heuristics, which is true for most properties being verified.

Table 5.2: Results Comparison

Benchmark	Vars#	iSels#	Spear(s)	Our CM/OM-Guided Abstraction-Refinement				
				Vars#	iSels#	% CNF(% SF)	Iter.	MINISAT (Spear)
vc331256	27248	3110	314.42	9259	1019	55.3 (48.50)	1	58.05 (**51.46**)
vc331257	27369	3149	437.13	9160	994	55.2 (48.15)	1	**52.30** (79.22)
vc331185	21990	2468	545.28	7466	831	55.4 (48.46)	2	**35.85** (44.17)
vc331190	23188	2656	632.33	7713	852	54.5 (48.18)	2	**26.83** (101.77)
vc331211	23554	2681	449.62	7959	877	56.2 (48.20)	2	**42.46** (60.98)
vc331179	21918	2553	44.31	7206	787	52.5 (48.06)	1	**25.80** (40.72)
vc331180	21094	2398	90.64	7036	771	52.3 (48.39)	1	**41.11** (56.37)
vc331228	21918	2920	175.01	8504	948	52.1 (48.18)	2	48.64 (**28.22**)
vc331218	25413	2920	140.99	8504	948	52.4 (48.00)	1	**28.67** (38.11)
vc1225825	17628	1985	41.39	5916	594	52.8 (34.9)	1	21.85 (**8.43**)
vc1225314	17481	1991	28.54	5847	604	54.0 (34.3)	1	22.51 (**13.51**)
vc1225832	19867	2273	64.80	7360	665	53.3 (43.1)	2	41.56 (**23.97**)
bs10-3-6*	488	45	N/A	195(132)	45	45.1	1	165.41/411.14
bs39-1-5*	21997	252	N/A	107(76)	252	0.5	1	23.65/80.04

* represents UNSAT property.

5.4 Experimental Results

To validate the effectiveness of our approach, we implemented the proposed method in C++, which is called C2BIT-2, and applied it to reason the quantifier-free bit-vector logic benchmarks [7] in (1) Spear_samba_v2.3.35_log suite, (2) Spear_samba_v2.3.35_client suite; (3) Some other software bounded model-checking instances. The selected benchmarks from 1) and 2) all consist of a large number of ITE nodes, and their satisfiability decisions required long execution times by Spear v2.3 (which reports best results for the bit-vector logic section in SMT competition 2007). C2BIT-2 uses Booleforce [14] to extract the UNSAT core, and MINISAT [89] to check the satisfiability of both the under-approximate model and the over-approximate abstraction. The number of subsets of the ITE variables in the set I classified by OM values is set as 7, and the size of IS_S is set as 20. All experiments are conducted on an Intel Xeon 2.8GHz processor with 2 GB RAM.

The results are reported in Table 5.2. First, for every benchmark, the attributes of the original bit-vector formula are given: Vars# reports the number of all bit-vector variables including constants; *iSel*# reports the number of *iSel*s. The runtime of Spear v2.3 is then reported. Then, for our approach, the number of all variables, *iSel*s and the ratio of CNF file size of the verification model after the slicing and abstraction of the original one is given. The corresponding percentage of input SF (Spear Format) file size for Spear is also given in parentheses. In the final column, the total runtime of our approach (including slicing, encoding, UNSAT core extraction and solving time on MINISAT) is reported. In parenthesis, we also report the run time of Spear on the final reduced model for comparison.

All benchmarks in Table 5.2, with the exception of the last two, have SAT properties. We first discuss the results from the SAT benchmarks. Compared with Spear, we have achieved significant speedup for all SAT instances. For vc331185 and vc331190, more than 10$\times$ speedup was observed using C2BIT-2. The SAT solution found in each benchmark has been validated with the provision of the CNF file and the variable mapping file generated by Spear. After slicing away redundant variables of the given formula, we found that only about one-third of *iSel*s are unique as shown in the 6^{th} column, which implies that multiple ITE variables share the same *iSel*. Furthermore, these *iSel*s have little dependency on each other, and their inputs are mostly close to PIs that have the high controllability. As shown in the 5^{th} column, we observed that the verification model (under-approximation) usually has only about

33% of all variables in the original formula and the generated CNF file size is about 50% of that from the original formula. For the benchmarks that require multiple iterations to verify, the *Vars#* shown is in the largest under-approximate model. For the benchmarks that could be verified in just one iteration, the non SAT-solving time is less than 5 seconds; for the rest of the benchmarks, this time is around 10 seconds, mainly due to the cost of handling the UNSAT core for refining the abstraction.

We also applied C2BIT-2 to two UNSAT properties of software BMC instances. We currently could not apply Spear to them for comparison because their format is not easily convertible to the input format that Spear requires due to the presence of array variables. Thus, we evaluate the effectiveness of our approach by comparing the approach with and without the proposed abstraction refinement framework. The primary results are shown in the last two rows. The number in the parenthesis of the 5^{th} column is the number of variables restored by OM heuristics out of all variables in the over-approximate abstraction. The two numbers given in the last column show the total runtime with and without the abstraction. The speedup reaches 3 times.

5.5 Related Work

Recently, Bryant et al. [20] proposed a proof-based abstraction refinement framework based on the under- and over-approximation to decide bit-vector arithmetic. The under-approximate model F_U is built through enforcing constraints on the encoding value range of variables. With the UNSAT proof of F_U, the over-approximate abstraction F_A includes only the subset of variables referred to the proof. This approach has shown effectiveness in two scenarios: (1) a SAT solution exists with a small number of bits encoded in F_U, and (2) only a small number of variables included in F_A participate in the UNSAT proof of the original formula.

Our approach is similar in that the over-approximate abstraction is also built based on the UNSAT proof of the under-approximate model. However, our model consists of a subset of paths obtained from the added constraints on the bit-vector variables that control the branch selection. Since we do not enforce constraints on the encoding size, the SAT solution does not need to be limited to a reduced value range. Furthermore, through the path(s)-based partition, we could perform the verification task incrementally

in the sense that the abstraction refined in the iteration only needs to be accurate enough to prove the part of the problem instance. It is different from [20] that the abstraction of the entire instance is created in every refinement iteration.

The symbolic software model-checking tool BLAST [60] uses predicate abstraction and counterexample driven refinement to lazily construct the abstraction during path enumeration. The new predicates are inferred to be interpolants found from the infeasibility proof of the path being traced. Our approach also uses the counterexample-guided refinement strategy and the UNSAT proof of the path-based partitions to gradually identify the verification-relevant variables for refining the abstraction. However, besides the basic difference that BLAST applies predicate abstraction while we use WA, our approach is different from BLAST in two additional aspects. First, BLAST considers the data-type as infinite integers, while we model the program data as bounded bit-vectors, so our reasoning is bit-precise. Second, we use the controllability and observability to guide the order of enumerating the set of paths, but BLAST uses the predicates for path enumeration.

In [5], the authors proposed a structural abstraction to facilitate the checking of software verification conditions by exploiting the natural function-level abstraction boundaries like function calls defined in the program. Multiple low-cost counter-example driven refinement steps are used to decide which function calls' constraints need to be included in the abstraction without requiring of the unsatisfiability proofs. We also aim to exploit the structural information of programs to develop a low-cost and accurate abstraction. However, we use the control structures to divide-and-conquer the problem.

5.6 Summary

We have presented a new abstraction refinement approach to solve bit-vector arithmetic formulas. It speeds up the search for a SAT solution by first building a small under-approximate model that includes a subset of paths in the formula. This is performed by imposing constant constraints on a small set of branching control variables in the formula, guided by controllability and observability metrics, together with slicing away those irrelevant portions. If the under-approximate model is unsatisfiable, it uses the UNSAT proof to obtain an over-approximate abstraction incrementally. In order to reduce the

computational cost of the proof-based refinement and avoid enumerating all possible paths in the program, we use the observability heuristics to guide restoring variables, such that the verification-relevant variables outside the UNSAT core are also brought into the abstract model. Our approach has shown both effectiveness and efficiency in solving the formulae generated from the verification instances of the embedded software with intensive control structures.

Chapter 6

A Reduced Bit-vector Encoding Width Computation Algorithm for Bit-precise Verification

Bit-precise verification with variables modeled as bit-vectors has recently drawn much interest. However, a huge search space usually results after bit-blasting. To accelerate the verification of bit-vector formulae, we propose an efficient algorithm to discover non-uniform bit-vectors' encoding widths W_e which may be smaller than their original modeling widths but sufficient to find a counterexample. Different from existing approaches, our algorithm is path-oriented, in that it takes advantage of the controllability and observability values in the structure of the model to guide the computation of the paths, their encoding widths, and the effective adjustment of these widths in subsequent steps. For path selection, a subset of single-bit path-controlling variables is set to constant values. This can restrict the search from those paths that have been deemed less favorable or have been checked in previous steps, thus simplifying the problem. Experiments show that our algorithm can significantly speed up the search by focusing first on those promising, easy paths for verifying those path-intensive models with reduced, non-uniform bit width encoding.

6.1 Motivation and Overview

In formal verification, modeling data variables as *bit-vectors* with a bounded width has shown some unique benefits. Bounded modeling is capable of accurately capturing the true semantics of the verification instances constrained by a physical word-size on a computer. Furthermore, with the advances in Boolean and bit-vector arithmetic reasoning, SAT (or SMT) based formal verification has the potential to deal with large problems. Many existing software model checking tools (e.g., CBMC [35], SATABS [36], Saturn [115], F-SOFT [70]) and hardware design validation techniques (e.g., [66, 82]) have taken bit-vector modeling.

However, with bit-vectors, the search space can be huge after bit-blasting, especially when dealing with large hardware designs and/or software programs. For example, in a large instance, it is extremely challenging to find a satisfying assignment (counterexample) with a full-size encoding. One way to handle this problem is to reduce the encoded bit-vector width of the variables, thereby restricting the searching space. Then, the verification is conducted on the restricted model instead of the original one. Several approaches have been proposed to compute the reduced bit-vector width for enhancing the verification scalability.

In [73], an abstraction approach was introduced to scale down the data path for formal RTL property checking. Based on the static data dependency analysis and granularity analysis of bit-vector equations, it computes an abstract model in which the bit-vector width of variables is reduced with respect to the property. To alleviate the state explosion in software model checking, the authors of [117] reduce the bit-vector width of variables according to their lower and upper bounds determined by a symbolic value range analysis technique. Both approaches are applied as preprocessing steps that directly decrease the modeling width W of each variable and still preserve the verification property. However, they do not consider the dynamic information during verification.

Recently, a new approach was proposed in an under- and over-approximation based abstraction-refinement framework to iteratively learn the sufficient encoding width W_e of variables for verification, which is smaller than their individual modeling widths [20]. Starting with a small W_e for every free variable, their approach enlarges W_e of some variables in each refinement step by analyzing the abstract counterexam-

ple of the over-approximate abstraction. For refutable properties (where a counterexample exists), the refinement process continues until a SAT assignment is found (with a smaller W_e) or when the W_e of all variables have reached their original W. This approach dynamically computes small values for W_e during verification instead of scaling down the width beforehand using static analysis as in [73, 117]. Although it is flexible, its claimed efficiency is limited in scenarios where the SAT assignment can be represented with a smaller encoding width. Moreover, the values assigned in the abstract counterexample may be unnecessarily large derived from a large W_e, and thus increases the verification difficulty.

In this chapter, we present an efficient path-oriented bit-vector encoding width computation algorithm to alleviate the above limitations. Similar to [20], our algorithm embeds the *dynamic* computation of W_e in the abstraction-refinement framework. However, it is distinguished by its path-oriented analysis with the guidance of *static* controllability metric (CM) and observability metric (OM) in three major ways. First, it computes the initial *non-uniform* W_e of variables on different paths. By setting a bigger initial W_e for the variables on the easy-to-control paths while setting a smaller W_e for the other paths, our approach can greatly increase the chance of finding a SAT assignment in the restricted search space directly, without the need to adjust W_e multiple times. Second, in the W_e adjustment steps (if necessary), our approach gives priority to enlarging the W_e of the easily-controllable variables first through the manipulating of the abstract counterexample generation guided by CM and OM. This helps to systematically search for the concrete counterexample with a reduced effort. Third, it sets W_e to zero for some single-bit variables that determine the path(s) selection, thereby enforcing constant values on them to restrict choosing only a subset of paths. This can avoid searching those partitions that have been checked in previous steps, especially the ones on which the variables' W_e experienced no increase, thus simplifying the problem.

The remainder of the Chapter is organized as follows. In Section 6.2, we will give some preliminaries related to our work. Our proposed encoding algorithm is presented in Section 6.3. We report our experimental results in Section 6.4 followed by the conclusion in Section 6.5.

6.2 Preliminaries

6.2.1 Bit-vector Formula Encoding

The bit-vector arithmetic formula we focus on is a conjunction of terms, where each term is in the format (Identifier == Identifier [op Identifier]). Every Identifier represents a bit-vector variable which is interpreted as a numeric value represented in two's complement form. According to whether the operator accepts single-bit inputs and whether the output is a single bit, we group operators into three categories:

- Bitwise operators: & (and), | (or), $\otimes$ (xor), $\sim$ (not);
- Predicate operators: $==,\ \neq,\ >,\ <,\ \geq,\ \leq$;
- Non-Boolean operators: $+,\ -,\ \times,\ /$, %, shift, type cast (Concatenation, Extraction and Extension), ITE $(if-then-else)$, etc;

The resulting formula can be represented as a directed acyclic graph model. An example is shown in Figure 6.1, where three possible paths (highlighted by dash lines) are possible to reach p from the inputs.

Definition 31. Starting from the least significant bit, the encoding width $W_e(v)$ for a bit-vector variable v is the number of consecutive bits in the vector whose values have not been assigned, $0 \leq W_e(v) \leq W(v)$. For each of the remaining $W(v) - W_e(v)$ bits, the value is set to be constant 0 (or 1).

If the W_e of individual variables is smaller than their W, the search space can be restricted. For example, a variable v with $W = 32, W_e = 6$, the original value range is $[-2^{31}, 2^{31} - 1]$ and the constrained value range is reduced to $[0, 63]$ by enforcing the 22 most significant bits to 0. When W_e is set equal to 0, the variable simply becomes a constant. We observe that the selector input of an ITE (if-then-else) variable (called *iSel*) has a special property. (In Figure 6.1, C_2 is an *iSel*.) When constant 0 (1) is enforced to an *iSel* (setting its W_e to zero), the false (true) branch is always taken, and some variables on the true (false) branch may become dangling variables. Thus, it is safe to slice away these dangling variables, as they do not feed other portions of the code.

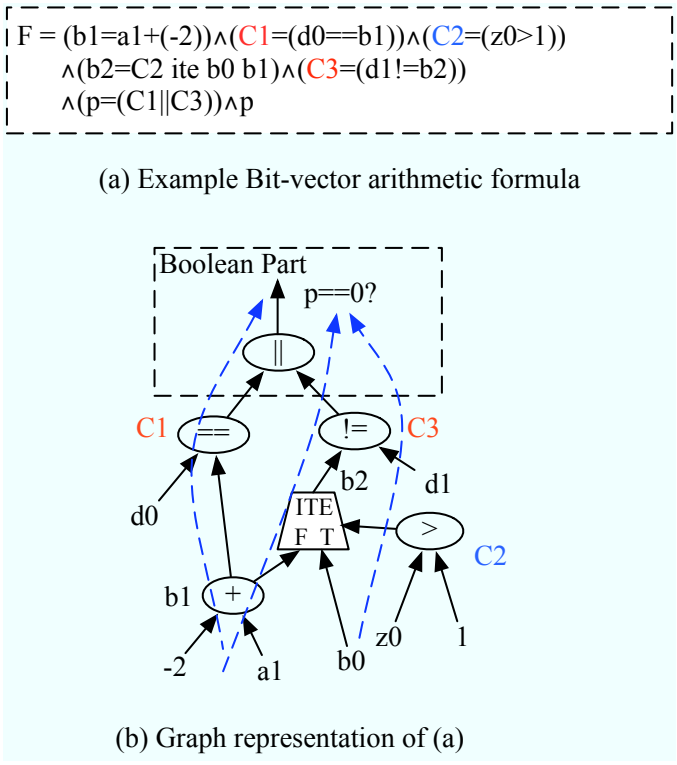


Figure 6.1: An example of Bit-vector formula with its graph model

Definition 32. A variable is a Boolean Frontier Variable (BFV) if it is the output of a predicate operator and all its fanouts are variables with Bitwise operators.

In Figure 6.1, C_1 and C_3 are BFVs. We consider them as pseudo inputs of the Boolean portion of the formula where every variable has $W = 1$.

6.2.2 Controllability/Observability Metrics

CM and OM are two generic metrics widely used to evaluate the testability of a hardware design ([64, 106]) or software components [93]. In [58], CM and OM have been used to estimate the amount of influence that each bit-vector variable has on the property verification. Specifically, the CM of a variable approximates the *difficulty* of setting a value along the paths reaching a target variable from the inputs. The *difficulty* is defined by two main factors: the lengths of the paths and the computation complexity along these paths. The OM approximates the amount of impact that a value-change on a variable has on the output. It is used to estimate the verification relevance of a variable to the target property of interest.

The CM/OM computation defined in [58] is adopted here. It uses pre-calculated controllability and observability coefficients (COC) of basic bit-vector operators to represent the operators' computational efficiency. The COC approximates the different amounts of influence the operators have on the CM and OM. The details of COC values and the formulas of the CM and OM computation are omitted due to the space limit. We introduce two properties of our CM/OM computation relevant to this work:

- Larger values of the CM/OM indicate harder controllable/observable.
- On any path from primary inputs (PIs) to a primary output (PO), $CM(v_1) < CM(v_2)$ and $OM(v_1) > OM(v_2)$ always hold if v_1 is the predecessor of v_2.

However, in [58], the 1- and 0-state CM of the variables in the Boolean portion of the instance were not differentiated. Here, we first estimate the 0-state CM^0 and 1-state CM^1 of every BFV *var* according to four cases below:

1. $op(var) \in \{\geq, \leq, <, >\} : CM^0 = CM^1 = 0.5$.

2. $op(var) \in \{==, \neq\}$ and at least one argument of *var* is fully controllable like PI: $CM^0 = CM^1 = 0.5$.

3. $op(var) \in \{==\}$ and no arguments of *var* is fully controllable: $CM^0 = 1 - 2^{-W}, CM^1 = 2^{-W}$.

4. $op(var) \in \{\neq\}$ and no arguments of *var* is fully controllable: $CM^0 = 2^{-W}, CM^1 = 1 - 2^{-W}$.

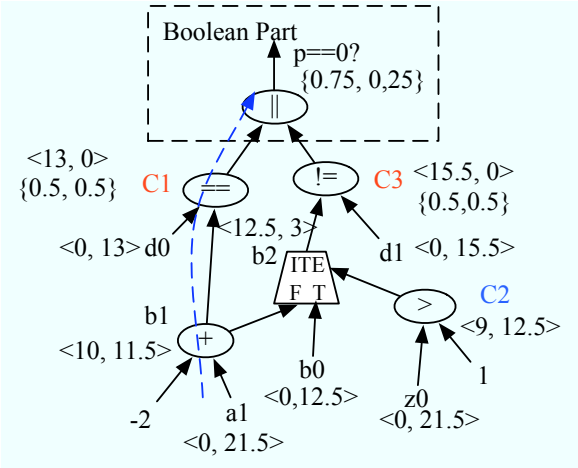


Figure 6.2: Graph model of Figure 6.1 labeled with CM/OM

We propagate the CM^0 and CM^1 of BFVs to all other variables in the Boolean portion according to the 1 and 0 probability measure of the corresponding bitwise operator. In Figure 6.2, it shows CM and OM labeled as a $< CM, OM >$ pair next to each variable outside or at the boundary of the Boolean portion. It also gives the CM^1 and CM^0 next to each variable in the Boolean portion (enclosed by the brace). For example, CM^0 and CM^1 of variable C_1 are both 0.5 based on Case(2) since $d0$ is a PI. To verify the property $p == 0$, C_1 and C_3 have the same CM^1. But since CM of C_1 is smaller than that of C_3 which means C_1 is more easily controllable, the path following the dash line is considered the easy-to-control path.

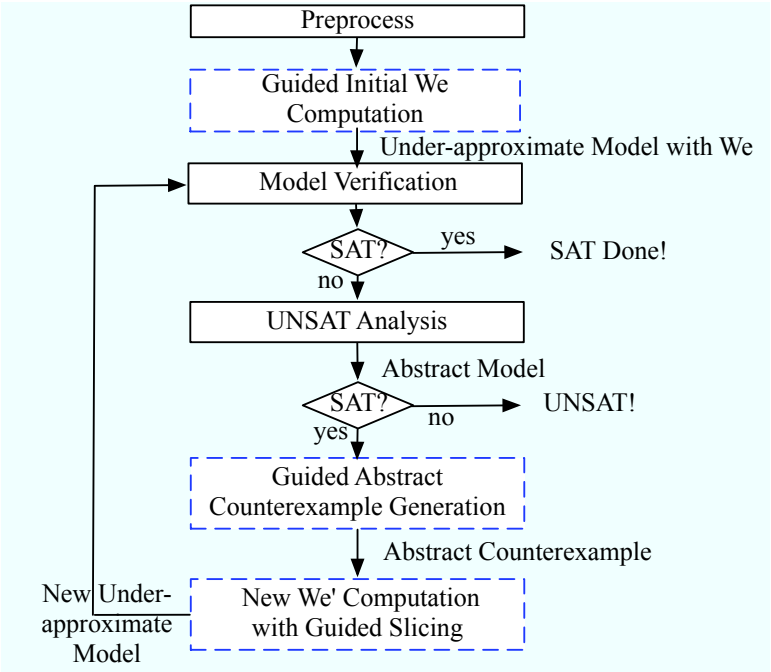


Figure 6.3: Basic flow of our algorithm

6.3 Our Proposed Algorithm

6.3.1 Overview of the Steps

To enhance the scalability of bit-precise verification, we propose an efficient path-oriented algorithm to compute a small but verification-sufficient encoding width W_e of individual variables in the instance. The algorithm exploits not only the dynamic information learned in the abstraction-refinement iterations, but also the high-level static structural information through the guidance of the controllability and observability metrics. We assume a verification instance formulated as a bit-vector arithmetic formula whose satisfiability corresponds to the negation of a given property (i.e., SAT means the property is refuted). We also assume many paths exist in the instance, which is common in practical problems. Finally, our current work focuses on refuting properties.

The basic flow of our algorithm is illustrated in Figure 6.3. We give an overview of each step below. Three important steps enclosed with dash line borders will be presented in more detail in following subsections.

1. *Preprocessing*: Given a bit-vector formula, our algorithm first builds its graph model M, then computes *BFVs*, CM, and OM. Next, it selects some *iSel*s to build the set *IS*. These *iSel*s stored in *IS* will be used to conduct the path(s) oriented abstract counterexample generation in Step 5 and slicing in Step 6. We prefer selecting the *iSel*s with a larger number of connections to the verification-relevant *ITE*s because they have a greater impact on reducing the partition size. The work proposed in [73, 117] can also be applied here to reduce the modeling width.

2. *Guided initial W_e computation*: Our algorithm computes a small initial W_e and determines the constant value to be placed for the bits outside of the encoding widths. The main idea is to give preference (larger initial W_e) to variables on easy-to-control paths so as to increase the chances of finding a SAT assignment fast. Moreover, it keeps the encoding size of a variable consistent with the input/output variables and the given operator type. The algorithm also considers the effect of constants in the encoding computation. It then applies the computed initial W_e of variables to build an initial under-approximate model M_u.

3. *Model verification*: Through bit-blasting, M_u is transformed to a Boolean model β_u. If β_u is SAT, we can conclude M is also SAT as the values enforced on M_u are achievable in M; otherwise, go to Step 4.

4. *UNSAT analysis*: Our algorithm adopts the method used in [57] to conduct the UNSAT analysis for refining the abstraction M_o. The basic idea is to identify *all* variables involved in the UNSAT proof of M_u and add them into M_o. Note that the constant values used outside the encoding widths and constant values for the *iSel* variables in M_u are not carried over. Thus, all variables in M_o have the original modeling width W. The refined M_o can refute all spurious counterexamples where the assignment of the variable is within the encoded value range of its W_e, so that it prevents repeated generation of the same spurious counterexample. A theorem on this can be found in [57]. This method is simpler than the technique in [20], whose proof-based abstraction also considers the special usage of Boolean nodes.

5. *Guided abstract counterexample generation*: Since M_o is small, it is easy to handle and find an abstract counterexample γ with certain expectations. This is done by enforcing some extra constraints on M_o guided by CM and OM to steer the search. To avoid generating a new M_u that is similar to the previous M_u in which no SAT assignment was found, we prefer generating

γ on M_o that can help enlarge those least frequently enlarged W_e. As some variables have values assigned in γ falling beyond the value ranges of their previous W_e values, it is also preferred that they are the easy-to-control ones.

6. *New W_e computation with guided slicing*: Given the abstract counterexample γ, we enlarge the W_e of some variables in the previous M_u so that the new value ranges can adequately cover their values assigned in γ. We update the W_e of all other data dependent variables in M and apply the new W_e on variables to building the new under-approximate model $M_u^{'}$. To avoid repeatedly searching the same space among the iterations, our algorithm enforces certain constant values on some *iSel*s and applies model slicing to remove from the new under-approximate model those variables whose W_e were not enlarged and have thus become dangling variables. The process goes back to Step 3 to start a new iteration.

Due to the finite search space, the algorithm always terminates. In the worst case, the W_e of individual variables may need to be enlarged to their original W. However, experiments show that a SAT assignment with small value ranges or on the path with the small number of variables exists in most instances.

6.3.2 Guided Initial W_e Computation

The initial W_e of variables including the constant value C_e chosen on bits outside of W_e are very important to the efficiency of our algorithm. We enforce the same constant value C_e to all bits beyond W_e starting from the most significant one of bit-vector variables. The algorithm of the initial W_e computation with two phases is shown in Figure 6.4. In phase 1, it identifies the PIs on the easy-to-control paths and sets a larger W_e to them; the other PIs are given a smaller W_e. Guided by CM and OM, our algorithm first backtraces from the target property along the easy paths in the Boolean portion of the instance to a set of *BFV*s. Then it extracts all non-Boolean part variables in the cone of these *BFV*s and place them in a new set S. It removes from S any variable on the hard-to-control branches of ITEs connected with the *iSel*s in the IS. We empirically set C_e as 0, choose 6 as the initial W_e for the PIs in S and 2 for the remaining PIs.

In phase 2, our algorithm adjusts the computed W_e and C_e considering the effects of constants in M.

```
Initial_encoding_width (M, P, IS )
 /*phase 1. CM/OM guided We computation*/
1. Backtrace from P to BFVs on easy-to-control paths;
2. Extract variables on traced paths to set S;
3. for each isel in IS
4.    Remove hard-to-control branches from S guided by CM;
5. end for
6. Set We=min{max{celling(W/6), 6},W}&(Ce=0) for all PIs in
S;
7. Set We=min{2,W}&(Ce=0) for all other PIs in M;
8. Propagate We for all other variables in M;

 /* phase 2.  We adjustment */
9. for each constant O connected with predicate OP
10.  if (O > 0)
11.     Set W_T=ceiling(log2(O), C_T=0;
12.  else
13.     Set W_T=ceiling(log2(-O), C_T=1;
14.  endif
15.   for each V related to O with predicate OP
16.     if ((We of V < W_T) || (Ce!= C_T) )
17.        Adjust We=W_T and (Ce=C_T) for V;
18.        Adjust We for PIs that Vs depends;
19.     endif
20.   end for
21. end for
22. Propagate adjusted We for all other variables in M;
23. end
```

Figure 6.4: Alg. of initial W_e computation

We observe from experiments that it is preferable to give the same negative or positive polarity for the variables with which the constant is computed. We set the W_e of variables that allow their encoded value range to cover the absolute value of the constant, especially for predicate operations. This is to avoid fixing the output value of such predicate operations. For example, consider the constraint $a > -4$; its value is not fixed only when setting W_e bigger than 2 and $C_e = 1$ for a. Finally, the adjusting of W_e and C_e on PIs are also propagated to all internal variables, while considering computation consistency on the operators along the paths.

```
//Choose K path selection variables from first Mo model to set Sp
 Cex_gen (Mo,  Sp )
1.  while (1)
2.    {C1,..,CK} = pathsSelGen(Sp);
3.    Enforce {C1,..,CK} assignment on Mo  to get Mo*;
4.    if(Mo* is SAT)    break;
5.  endwhile

6.  Sort all pseudo PIs of Mo w.r.t. CM in increasing order;
7.  Divide all sorted pseudo PIs into L groups;
8.  Enforce current We for all pseudo PIs to get M**;
9.  for i = 1 to L
10.   Refine M** by enlarging We for pseudo PIs in group i;
11.   if (Refined M** is SAT)
12.       CEXRET = SAT-SOL;
13.       break;
14.   endif
15.  endfor
16. end
```

Figure 6.5: Alg. of abstract counterexample generation

6.3.3 Abstract Counterexample Generation

In this step, our algorithm sequentially enforces two kinds of constant values on M_o to steer the search for an abstract counterexample as shown in Figure 6.5. The set S_p, which consists of some *BFV*s and *iSel*s included in the M_o, is constructed beforehand. A combination of constant values is first imposed on the variables in the set S_p so as to restrict searching on a subset of paths. The function *pathsSelGen*, which returns such constant assignments $\{C_1,..,C_K\}$, starts enumerating variables in S_p at the first iteration and stops until the verification is finished. Since no SAT assignment was found in M_u, we assume a low chance that a SAT one exists on the partition of the instance similar to M_u, and we expect that the new M_u bears the least similarity to the previous M_u in the search space. So, our goal is to return a value combination with the least similarity to the last one and not being found as infeasible so as to constrain the search in a different subset of paths. Once a value assignments is found satisfying M_o, we apply it to M_o to restrict the search space to ease the generation task. Next, small encoding widths W_es are applied to some pseudo PIs of M_o to further constrain the search. A greedy search starts from the most easily controllable PIs, so that the value assignments on the harder-to-control pseudo PIs in the returned counterexample can still be covered by their present W_e.

Theorem 1. *An infeasible assignment on M_o is also infeasible in M.*

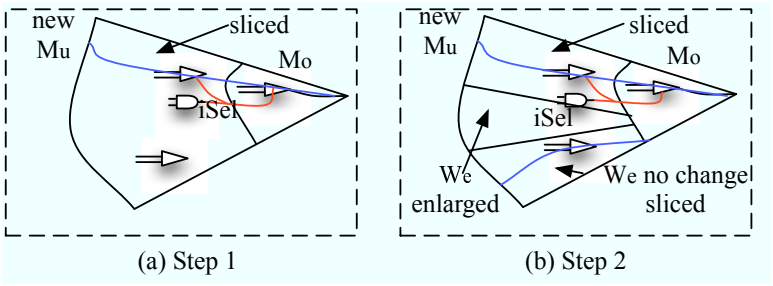


Figure 6.6: Two-steps guided slicing

Proof. Since the set of clauses β_o of M_o is a subset of CNF clauses β of M, if an assignment $v_0,..,v_n$ cannot satisfy β_o, this same assignment also cannot satisfy β because at least a subset (β_o) cannot be satisfied in β. □

With this theorem, we can safely check the invalidity of some assignments in the M_o with a low cost. It is especially efficient to identify a subset of infeasible paths using M_o.

6.3.4 New W_e with Guided Slicing

We focus on introducing the guided slicing process as illustrated in Figure 6.6. First, we apply the assignments $\{C_1,..,C_K\}$ (that were obtained from M_o^*) to the new M_u and slice away any dangling variables. In Figure 6.6(a), the *iSel* whose value is enforced controls the *ITE*s of M both inside M_o and outside of M_o. So, more branches can be sliced away from the new M_u compared to M_o. Next, the branches of the ITEs controlled by *iSel*s in *IS* on which variables had no enlarged W_e are removed. In Figure 6.6(b), the variables whose W_e need to be changed are in the center cone. The path branches shown in the bottom are outside of this cone and are removed from the new M_u.

6.4 Experimental Results

To validate the effectiveness of our approach, we implemented the proposed method in C++, which is called C2BIT-2, and applied it to the bit-vector arithmetic benchmarks [7] in the following suites: *Spear* and *TACAS*07. There are two main reasons that 14 benchmarks were chosen: either 1) a benchmark is

Table 6.1: Results Comparison (Benchmarks from Spear, TACAS07)

Benchmark	Vars#	iSels#	Spear	Proof-Based			CM/OM Guided		
			(s)	W_e	Iter.	T(s)	W_e	Iter.	T(s)
log_331256	27773	3110/1019	314.42	4/12	2	117.71	4/6	2	43.67
log_331257	27369	3149/994	437.13	4/18	2	76.59	4/12	2	47.27
log_331190	23188	2656/852	632.33	4/18	2	46.96	4/12	2	34.36
log_331211	23554	2681/877	515.31	4/12	2	91.20	4/12	2	33.32
innd_33359	1368	77/33	49.66	8/32	2	41.28	8/12	2	37.90
innd_33725	1025	52/22	55.26	8/32	2	43.32	8/12	2	37.16
nnrpd_21453	1330	76/32	20.94	8/18	2	47.21	8/18	3	50.76
wget_17909	1042	37/32	380.74	8/18	2	520.3	8/18	2	148.97
wget_18506	1062	38/30	39.86	8/18	2	30.26	8/18	2	29.14
cli_1225314	17481	1991/604	40.45	4/12	2	33.52	4/12	2	27.70
cli_1225757	18171	2090/616	25.02	4/12	2	36.79	4/12	2	25.84
cli_1225783	18766	2159/637	51.06	4/6	2	57.80	4/6	2	28.19
cli_1225832	19867	2273/665	65.25	4/12	2	75.28	4/12	2	29.67
S-40-50	1321	0(156)/0	14.36	4/32	4	98.47	4/16	2	30.29

Table 6.2: Results Comparison (Benchmarks from Spear, TACAS07)

Benchmark	Vars#	iSels#	C2BIT-2		
			$W_e <,>$	Iter.	T(s)
log_331256	27773	3110/1019	$<6,2>$	1	10.21
log_331257	27369	3149/994	$<6,2>$	1	9.38
log_331190	23188	2656/852	$<6,2>$	1	5.34
log_331211	23554	2681/877	$<6,2>$	1	6.47
innd_33359	1368	77/33	$<12,2>/12$	2	30.29
innd_33725	1025	52/22	$<12,2>/12$	2	33.71
nnrpd_21453	1330	76/32	$<12,2>/12$	2	34.39
wget_17909	1042	37/32	$<12,2>/12$	2	50.38
wget_18506	1062	38/30	$<12,2>$	1	15.34
cli_1225314	17481	1991/604	$<6,2>$	1	5.2
cli_1225757	18171	2090/616	$<6,2>$	1	6.21
cli_1225783	18766	2159/637	$<6,2>$	1	4.53
cli_1225832	19867	2273/665	$<6,2>$	1	6.31
S-40-50	1321	0(156)/0	$<8,2>$	1	12.31

path intensive that has a large amount of *iSel*s or *BFV*s, or 2) verification takes a long time with existing tools. They are all refutable properties (a satisfiable solution exists). C2BIT-2 uses Booleforce v1.0 to extract UNSAT core in the UNSAT case, and uses MINISAT v1.14 to generate abstract counterexamples and verify the satisfiability of the under-approximate model. The experiments were conducted on an Intel Xeon 2.8GHz processor with 2 GB RAM.

The results are reported in Table 6.1, 6.2. First, for every benchmark, the characteristics of the original bit-vector formula are given: Vars# reports the number of bit-vector variables including constants; *iSel*# reports the number of *iSel*s both before and after pruning. After pruning the formula instance, we found that only about one-third of *iSel*s are unique. For example, in log_vc331256, only 1019 out of 3110 *iSel*s are unique which corresponds to $3110/1019$ in the table. This implies that multiple ITE variables may share the same *iSel*. Furthermore, these *iSel*s have no dependency to each other, and their inputs are close to PIs, thus they are very controllable. For the last benchmark, *iSel*# is zero, which means no ITE variables, but the number of *BFV*s is big, as shown in parentheses. Next, the runtime of Spear v2.6 is reported. We apply three methods on the benchmarks:

1. uniform initial W_e with the proof-based refinement as in [20];
2. uniform initial W_e with CM/OM guided refinement;
3. non-uniform initial $W_e < max, min >$ with CM/OM guided refinement (C2BIT-2).

For each benchmark, the initial/final *modal* value of encoding widths after refinement are reported as $4/12$ given in the table, followed by the number of refinement iterations and total runtime (including pruning, encoding, UNSAT core extraction and solving time) of these three methods. For instance, in innd_33359, the initial uniform W_e is 8 and the final *mode* of W_e after two iterations is 32 for method 1 and 12 for method 2. With method 3) the non-uniform initial W_e is $< 12, 2 >$ (variables on easy-to-control paths have width of 12 and the rest have widths of 2), and the final *mode* of W_e is still 12, as only few variables have the enlarged W_e after refinement. All methods need 2 iterations to find the SAT solution for this particular instance but C2BIT-2 is the fastest.

Compared with Spear, C2BIT-2 has achieved significant speedups. Some were even greater than $10\times$. Note that the assignment found for each benchmark was validated using the CNF file and the variable

mapping file generated by Spear. We also observe that the maximum W_e computed by two refinement methods are the same or similar for many benchmarks. One reason is that modern SAT solvers typically return a 'maximally-false' solution that contains as many false bits as possible that can produce small value assignments. However, with the CM/OM guidance, the enlargement of variables' encoding widths focuses on the subset of paths so that the slicing can be conducted to reduce the model size and solving time. With C2BIT-2, the SAT assignment for 10 out of 14 benchmarks can be obtained in just one iteration with our initial non-uniform bit width encoding. It shows that this encoding effectively increases the chances of finding a SAT solution on the easy paths and requires minimal effort for searching on hard-to-control paths.

6.5 Summary

We have presented an efficient algorithm of computing small encoding widths for bit-vectors by utilizing a path-oriented abstraction-refinement framework. This algorithm exploits both the high-level structure and dynamic verification knowledge to effectively steer the search. It takes advantage of the controllability and observability metrics to guide three major steps: initial encoding width computation, abstract counterexample generation, and under-approximate model slicing. Experiments show that our proposed algorithm can reduce the solving time significantly, especially in verifying the paths-intensive designs.

Chapter 7

Conclusions

In this dissertation, we mainly address the limited scalability problem of automatic formal verification techniques like model checking, in the application of ensuring the correctness of embedded software programs with respect to the critical safety properties. We introduce performing the bit-precise verification for the bounded verification model of software, in which all the data variables are modeled as the fixed-width bit vectors, all the loops or recursions are unrolled in certain bound. In order to promote the scalable application of such bit-precise verification to the real-world embedded software in practice, we proposed several efficient property-based automatic abstraction refinement techniques with the assist of static program analysis, symbolic simulation, and testability guidance.

Firstly, based on the observation that the properties under verification usually depend on a small portion of the program, we proposed an approach to accurately and efficiently find this portion, by incorporating the program slicing and the proof-based abstraction refinement. Our proposed approach tightly integrates an aggressive static program slicing approach, which can reduce programs to the segments relevant for a particular computation into the software verification model construction and reduction process. This allows for effectively removing the program segments that have no computational relevance to the property under verification, so as to greatly reduce the model size. Our slicing operations also naturally combine the compilation optimization techniques, such as constant propagation, to effectively compute the accurate program slice.

After the slicing, we explored a proof-based abstraction-refinement strategy using under and over-

approximation of the verification model to construct a localization abstraction, which further removes the program segments having no verification relevance to the property. A heuristics method by program analysis was also proposed to more effectively refine the under-approximation in each iteration. Experiments conducted on the programs from wireless cognitive radio software systems have shown the effectiveness of the proposed approach.

Second, we investigated the potential of combining symbolic simulation for the scalable formal verification. Our approach uses distinguishing X as symbolic values to abstract the concrete variables' values, so that a small number of identified symbolic input vectors can cover the complete input space. In order to reduce the verification time and ease computing the symbolic input vectors, we apply the symbolic simulation to check the abstract model instead of the original program, which is automatically constructed in the counterexample-guided abstraction-refinement framework. This abstract model is a localization abstraction of the program, which includes a subset of property-relevant program constraints. So it may be more easily verified than the original program. The property-relevance of program constraints is automatically determined during the iterative abstraction refinement procedure.

In the next two proposed abstraction techniques, we explored the common testability metrics — controllability metric (CM) and observability metric (OM) — as the high-level structural guidance to construct the accurate abstract model with fewer control paths and smaller variable value ranges for the scalable bit-precise verification.

Third, we have presented a new abstraction refinement approach, based on the concept of the under- and over-approximation for solving bit-vector arithmetic formulae, generated from control-dominated embedded software verification instances. We also designed a new CM and OM computation method and applied them to efficiently guide the proposed abstraction procedure. Our approach builds the under-approximate model by enforcing constant constraints to a small set of single-bit variables, which control the branch selection of ITE nodes. This restricts the search space to only a subset of formula constraints. With the guidance of CM and OM, our approach can quickly find a satisfying solution on the easily controllable portion of the formula if a solution exists on it.

Our approach computes the over-approximate abstraction via learning the UNSAT proof of the under-approximate model. It also restores additional verification relevant constraints according to the OM

heuristics to reduce the high computational cost of refinement by only using the UNSAT proof. As a result, a sufficiently accurate abstract model for the verification can be built quickly, long before all partitions have been enumerated. To start the new refinement iteration, we use the satisfiable assignment of the current over-approximate abstraction to guide the construction of a new under-approximate model with an unvisited portion in the formula. With our approach, the verification can be conducted incrementally due to the partition-based feature of our approach. The experimental results show that a significant amount of solving time can be reduced for the benchmarks generated from the bit-precise verification of embedded software compared to state-of-the-art solvers for bit-vector arithmetic.

Finally, we propose an efficient algorithm to iteratively discover non-uniform encoding widths W_e of variables in the verification model, which may be smaller than their original modeling widths but sufficient to the verification. This algorithm exploits both the high-level structure and dynamic verification knowledge to effectively steer the search. Different from existing approaches, our algorithm is path-oriented in that it takes advantage of the CM and OM values to guide the computation of the encoding widths in three major operations: (1) computation of the initial non-uniform W_e of variables on different paths, (2) generation of an abstract counterexample, which affects the computation of the enlarged W_e of variables in the new under-approximate model, and (3) under-approximate model slicing to avoid searching the paths that have been checked. Our approach is capable of restricting the search from those paths that are deemed less favorable or have been checked in earlier steps, thus simplifying the verification problem. Experiments demonstrate that our algorithm can significantly speed up the verification, especially in searching for the counterexample violating the property.

7.1 Recommendations for Future Research

Our research presented in this dissertation shows the promise of bit-precise verification of embedded software programs in scalable applications. It can serve as the foundation for promoting the scalable bit-precise verification of software programs in general. We recommend several avenues for the future research based on our current vision.

7.1.1 Larger Experiments

Due to the variety of software programs, the standard or even widely accepted benchmarks still do not exist. It is always a challenge to conduct the research related to the software. The benchmarks used in our current research are from the SMT competition [80]. Most of them are derived from the verification of NULL pointers in real-world embedded software. In the future, we expect to produce more benchmarks for checking complex properties from large scale software systems in the embedded applications, such as software defined radio, automotive and avionic control systems, etc.

7.1.2 Program Analysis Enhanced Abstraction Refinement

As we know, the automatic abstraction refinement becomes a necessary technique in practical software formal or semi-formal verification. However, many abstraction refinement procedures used in software verification follow the general strategies of their applications in hardware design verification, without considering the uniqueness of software programs. Moreover, in the most abstraction practices that we observed, the verification performance decreases, once the number of refinement iterations becomes large. Therefore, one potential direction of the future research is to make use of the knowledge specific to programs, which can be exploited by the code analysis techniques, thereby enhancing the accuracy and efficiency of abstraction refinement.

Concept analysis [49] separates groups of *objects* that have common *attributes* based on the concept lattice. As a static code analysis technique, it has been applied in support of program understanding, change impact analysis [97,108] and modularization of legacy code [105]. Concept analysis can provide a decomposition of program with the cohesive grouping of functions or program statements. It is related to, but different from another popular code analysis technique — program slicing that decomposes the program by focusing on the subcomputations performed on it with respect to some slicing criterion. In [108], they have taken advantage of both concept analysis and program slicing by proposing a program representation called *concept lattice of decomposition slices*, where all dependences between the computations performed by a program are explicitly indicated. So, it is easy to determine whether the computation performed on a variable x is relevant to the computation on another variable y by querying this lattice.

One core task of abstraction refinement is to automatically learn whether a program variable is relevant or irrelevant to the verification of the specified property. We hope that the concept lattice built on program decomposition slices [108] can help improve the effectiveness of refining the abstraction with the low computational cost at the same time, as well as reducing the complexity of the abstract model. Here, we use the proof-based abstraction refinement approach with under- and over- approximation as the basis for the following discussion. For example, we can use the concept lattice to guide building the under-approximate model with certain expectations that depend on the specific problem, such as including the program segments shared by a large number of program decomposition slices, which may increase the chances of finding a counterexample violating the property at the early refinement iterations. For another example, we can first use the UNSAT proof to precisely identify a set of variables relevant to the verification with respect to a simple under-approximate model. Then we use the concept lattice to select the variables that need to be restored to refine the abstract model. For some existing variables on which the program constraints will not be used for the verification in future iterations, we could remove them from the model as well. As a result, the complexity of the abstract model can be reduced, without incurring the verification cost increase during the refinement. We expect that such a code analysis based method can also be applied to enhance other abstraction refinement techniques.

An invariant at certain line l is defined as an assertion δ over program variables, such that δ always holds on all the execution traces that reach at line l. Inductive program invariant plays a critical role for both proving program correctness and finding bugs. Some techniques have been established to dynamically detect the potential invariants by learning the simulation profiling, as in [29, 45]. The validity of these invariants needs to be checked using some static verification methods [94]. The constraint-based invariant generation [100] was also proposed to find true invariants directly.

In order to accomplish trimming the abstract model, we hope apply the potential program invariants identified by the dynamic program analysis technique in addition to applying the concept analysis. Due to the conservative property of abstraction, an invariant is a true invariant in the more precise abstraction if it holds in the coarse abstract model. If we can validate an invariant in the abstract model in the earlier iteration with low computational cost, we can use it in the later refined abstract model to compact the state space, so as to trim the verification model.

7.1.3 Verification of Concurrent Programs

Concurrency is a model of computation that allows many units of execution to coexist. With the emergence of multi-core systems for ever increasing the processing power, more and more vicious bugs occur in concurrent software systems with multiple threads that communicate via shared memory or message passing. Traditional verification techniques like testing fail in the presence of concurrency due to the difficulties of reproducing erroneous behavior. We expect that the bit-precise verification of concurrent programs is one of future research directions.

Bibliography

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Press, 1986.

[2] N. Amla and K.L. McMillan. A hybrid of counterexample-based and proof-based abstraction. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD)*, 2004.

[3] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2008.

[4] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

[5] D. Babic and A.J. Hu. Structural abstaction of software verification conditions. In *Proc. of Intl. Conf. on Computer Aided Verification (CAV)*, 2007.

[6] D. Babic and M. Musuvathi. Modular arithmetic decision procedure. In *Technical report, Microsoft Research, Redmond*, 2005.

[7] Domagoj Babi'c. *Exploiting Structure for Scalable Software Verification*. PhD thesis, University of British Columbia, Vancouver, Canada, 2008.

[8] T. Ball and S.K. Rajamani. Behop: A symbolic model checker for boolean programs. In *Proc. of SPIN*, 2000.

[9] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of Model Checking Software, 8th Intl. SPIN Workshop*, 2001.

[10] C.W. Barrett, D.L. Dill, and J.R. Levitt. A decision procedure for bit-vector arithmetic. In *Proc. of ACM/IEEE Design Automation Conference (DAC)*, 1998.

[11] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proc. of ACM/IEEE Design Automation Conference (DAC)*, pages 317–320, 1999.

[12] B. Biezer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[13] P. Bjesse and J. Kukula. Using counterexample guided abstraction refinement to find complex bugs. In *Proc. of ACM/IEEE Design, Automation and Test in Europe (DATE)*, 2004.

[14] Booleforce. http://fmv.jku.at/booleforce.

[15] A.B.Y Bres, G. Berry, and E.M. Sentovich. State abstraction techniques for the verification of reactive circuits. In *Proc. of Designing Correct Circuits*, 2002.

[16] R. Brinkmann and R. Drechsler. Rtl-datapath verification using integer linear programming. In *Proc. of VLSI Design*, 2002.

[17] R.E. Bryant. Formal verification of digital circuits using symbolic ternary system models. In *Proc. of Intl. Conf. on Computer-Aided Verification (CAV)*, 1990.

[18] R.E. Bryant. Symbolic simulation-techniques and applications. In *Proc. of ACM/IEEE Design Automaton Conference (DAC)*, 1990.

[19] R.E. Bryant. Modeling data in formal verification: Bits, bit vectors, or words. In *www.cs.cmu.edu/ bryant/presentations/fmcad07-tutorial.ppt*, 2007.

[20] R.E. Bryant, D. Kroening, J. Ouaknine, S.A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2007.

[21] R.E. Bryant, S.K. Lahiri, and S.A. Sehia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. of Intl. Conf. on Computer Aided Verification (CAV)*, 2002.

[22] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In *Proc. of 1991 Intl. Conf. on Very Large Scale Integration (VLSI)*, 1991.

[23] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. In *IEEE Trans. on Computers-Aided Design of Integrated Circuits*, 1994.

[24] J.R. Burch, E.M. Clarke, and K.L. McMillan. Symbolic model checking: 10(20) states and beyond. In *Information and Computation*, volume 98, pages 142–170, 1992.

[25] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler. Exe: Automatically generating inputs of death. In *Proc. of 13th ACM Conference on Computer and Communications Security*, 2006.

[26] S. Chai, E.M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. In *Proc. of Intl. Conf. on Software Engineering (ICSE)*, 2003.

[27] C.H. Chen and P.R. Menon. An approach to functional level testability analysis. In *Proc. of Intl. Test Conf (ITC)*, 1989.

[28] X. Cheng and M.S. Hsiao. Ant colony optimization directed program abstraction for software bounded model checking. In *Proc. of the Intl. Conf. on Computer Design (ICCD)*, 2008.

[29] X. Cheng and M.S. Hsiao. Simulation-directed invariant mining for software verification. In *Proc. of ACM/IEEE Design, Automation and Test in Europe (DATE)*, 2008.

[30] J.D. Choi and J. Ferrante. Static slicing in the presence of goto statements. In *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 1994.

[31] E.M. Clarke, M. Fujita, S.P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing for vhdl. In *Proc. of Software Tools for Technology Transfer (STTT)*, volume 4(1), pages 125–137, 2002.

[32] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of Intl. Conf. on Computer Aided Verification (CAV)*, 2002.

[33] E.M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[34] E.M. Clarke, A. Gupta, H. Jain, and H. Veith. Model checking: back and forth between hardware and software. In *Proc. of Verfied Software: Theories, Tools Experiments Conf. (VSTTE)*, 2005.

[35] E.M. Clarke and D. Kroening. A tool for checking ansi-c programs. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 168–176, 2004.

[36] E.M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 570–574, 2005.

[37] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *Proc. of Intl. Conf. on Computer Aided Verification (CAV)*, 2005.

[38] W. Craig. Linear reasoning: A new form of the herbrand-gentzen theorem. In *Journal of Symbolc Logic*, 1957.

[39] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *ACM Trans. on Programming Languages and Systems*, pages 451–490, 1991.

[40] Satyaki Das. *Predicate Abstraction*. PhD thesis, Dept. of Electrical Engineering in Stanford University, Stanford, CA, USA, 2003.

[41] M. Davis, G. Logemann, and D.W. Loveland. A machine program for theorem proving. In *Communications of the ACM*, volume 5, pages 394–397, 1962.

[42] N. Dershowitz and D.A. Plaisted. Rewriting. In *Handbook of Automated Reasoning*, pages 535–610, 2001.

[43] S. Edwards, T. Ma, and R. Damiano. Using a hardware model checker to verify software. In *Proc. of Intl. Conf. on ASIC*, 2001.

[44] M. Ernst. Practical fine-grained static slicing of optimized code. In *Technical Report MSR-TR-94-14, Microsoft Research*, 1994.

[45] M.D. Ernst, J.H. Perkins, and etc. P.J Guo. The daikon system for dynamic detection of likely invariants. In *Proc. of Science of Computer Programming*, 2007.

[46] F. Fallah, S. Devadas, and K. Keutzer. Occom: efficient computation of observability-based code coverage metrics for functional verification. In *Proc. of ACM/IEEE Design Automation Conference (DAC)*, 1998.

[47] V. Ganesh, S. Berezin, and D.L. Dill. A decision procedure for fixed-width bit-vectors. In *Technical Report CSTR 2007-06, Standford University, Stanford, CA, USA 94305-9025*, 2007.

[48] Vijay Ganesh. *Decision Procedures for Bit-vectors, Arrays and Integers*. PhD thesis, Dept. of Computer Science in Stanford University, Stanford, CA, USA, 2007.

[49] B. Ganter and R. Wille. *Formal Concept Analysis*. Springer-Verlag, 1996.

[50] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. of Programming Language Design and Implementation (PLDI)*, 2005.

[51] L. Goldstein and E. Thigpen. Scoap: sandia controllability/observability analysis program. In *Proc. of ACM/IEEE Design Automation Conference (DAC)*, 1980.

[52] S.G. Govindaraju and D.L. Dill. Counterexample-guided choice of projections in approximate symbolic model checking. In *Proc. of Intl. Conf. on Computer Aided Design (ICCAD)*, 2000.

[53] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Proc. of Intl. Conf. on Computer Aided Verification (CAV)*, 1997.

[54] A. Groce and D. Kroening. Making the most of bmc counterexamples. In *Workshop on BMC*, 2004.

[55] X. Gu. Rt level testability improvement by testability analysis and transormations. In *PhD thesis, Linkoping University, Sweden*, 1996.

[56] N. He and M. Hsiao. Using symbolic simulation and weakening abstraction for formal verification of embedded software. In *Proc. of Intl. Conf. on Software Engineering and Applications*, 2006.

[57] N. He and M. Hsiao. Bounded model checking of embedded software in wireless cognitive radio systems. In *Proc. of Intl. Conf. on Computer Design (ICCD)*, 2007.

[58] N. He and M. Hsiao. A new testability guided abstraction to solving bit-vector formula. In *ACM Intl. Conf. Proceeding Series: Proc. of the Joint Workshops of the 6th Intl. Workshop on Satisfiability Modulo Theories and 1st Intl. Workshop on Bit-Precise Reasoning*, 2008.

[59] N. He and M. Hsiao. An efficient path-oriented bit-vector encoding width computation algorithm for bit-precise verification. In *Proc. of ACM/IEEE Design Automation and Test in Europe Conference (DATE)*, 2009.

[60] T.A. Henzinger, R. Jhala, R. Majumda, and G. Sutre. Lazy abstraction. In *ACM Conf. on Principles of Programming Languages (POPL)*, pages 58–70, 2002.

[61] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *ACM Conf. on Principles of Programming Languages (POPL)*, 2004.

[62] W. Hodges. *Model Theory*. Cambridge University Press, 1993.

[63] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *ACM Trans. on Programming Languages and Systems (TOPLAS)*, volume 12(1), pages 26–60, 1990.

[64] F. Hsu and J.H. Patel. High-level variable selection for partial-scan implementation. In *Proc. of Intl. Conf. on Computer Aided Design (ICCAD)*, 1998.

[65] F. Hsu, E.M. Rudnick, and J.H. Patel. Enhancing high-level control-flow for improved testability. In *Proc. of Intl. Conf. on Computer Aided Design (ICCAD)*, 1996.

[66] C.Y. Huang and K.T. Cheng. Assertion checking by combined word-level atpg and modular arithmetic constraint-solving techniques. In *Proc. of ACM/IEEE Design Automation Conference (DAC)*, 2000.

[67] C.Y. Huang and K.T. Cheng. Assertion checking by combined word-level atpg and modular arithmetic constraint-solving techniques. In *Proc. of IEEE/ACM Design Automation Conference (DAC)*, 2000.

[68] R.I.G Hughes. *A Philosophical Companion to First-Order Logic*. Hackett Publishing Company, 1993.

[69] Michael Huth. Abstraction. In *http://www.doc.ic.ac.uk/ mrh/talks.html*, 2002.

[70] F. Ivancic, I. Shlyakhter, M. Ganai, and A. Gupta. Model checking c programs using f-soft. In *Proc. of Intl. Conf. on Computer Design (ICCD)*, 2005.

[71] J. Ivers and N. Sharygina. Overview of comfort: A model checking reasoning framework. In *Technical Report, CMU/SEI-2004-TN-018*, 2004.

[72] et al J.L. Carter. Restricted symbolic evaluation is fast and useful. In *Proc. of Intl. Conf. on Computer Aided Design (ICCAD)*, 1998.

[73] P. Johannsen and R. Drechsler. Formal verification on the rt level computing one-to-one design abstractions by signal width reduction. In *Proc. IFIP International Conference on Very Large Scale Integration*, 2001.

[74] S. Khurshid, C.S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2003.

[75] J.C. King. Symbolic execution and program testing. In *Communications of ACM*, 1976.

[76] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1997.

[77] D. Kroening and O. Strichman. *Decision Procedures an Algorithmic Point of View*. Springer, 2008.

[78] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[79] B. Li and F. Somenzi. Efficient computation of small abstraction refinements. In *Proc. of the Intl. Conf. on Computer Aided Design (ICCAD)*, 2004.

[80] SMT lib. http://combination.cs.uiowa.edu/smtlib/.

[81] F.Y.C. Mang and P.H. Ho. Abstraction refinement by controllability and cooperativeness analysis. In *Proc. of ACM/IEEE Design Automation Conference (DAC)*, 2004.

[82] P. Manolios, S.K. Srinivasan, and D. Vroon. Automatic memory reductions for rtl-level verification. In *Proc. of Intl. Conf. on Computer Aided Design (ICCAD)*, pages 786–793, 2006.

[83] J.P. Marques-Silva and K.A. Sakallah. Grasp: A search algorithm for propositional satisfiability. In *IEEE Trans. on Computers*, volume 48, pages 506–521, 1999.

[84] K.L. McMillan. www.kenmcmil.com/cav05tut.ppt.

[85] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[86] K.L. McMillan. Applications of craig interpolants in model checking. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2005.

[87] K.L. McMillan and N. Amla. Automatic abstraction without counterexample. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2003.

[88] A.H. Mekler and E.M. Nelson. Equational bases for if-then-else. In *SIAM Journal on Computing*, volume 16(3), pages 465–485, 1987.

[89] Minisat. www.cs.chalmers.se/cs/research/formalmethods/minisat.

[90] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering and efficient sat solver. In *Proc. of ACM/IEEE Design Automation Conference (DAC)*, 2001.

[91] M. Murciano, G. Gabodi, and P. Camurati. Automated abstraction by incremental refinement in interpolant-based model checking. In *Proc. of Intl. Conf. on Computer Aided Design (ICCAD)*, 2008.

[92] G. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. In *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 1979.

[93] T. Nguyen, M. Delaunay, and C. Robach. Testability analysis for software components. In *Proc. of Intl. Conf. on Software Maintenance(ICSM)*, 2002.

[94] J.W. Nimmer and M.D. Ernst. Static verification of dynamically detected program invariants: Integrating daikon and esc/java. In *Proc. of First Workshop on Runtime Verification (RV)*, 2001.

[95] G. Parthasarathy, M.K. Iyer, K.T. Cheng, and L.C. Wang. An efficient finite-domain constraint solver for circuits. in: Design automation conference. In *Proc. of IEEE/ACM Design Automation Conference (DAC)*, 2004.

[96] G. Parthasarathy, M.K. Iyer, T. Feng, L.-C. Wang, K.-T. Cheng, and M.S. Abadir. Combining atpg and symbolic simulation for efficient validation of embedded array systems. In *Proc. of Intl. Test Conf. (ITC)*, 2002.

[97] X. Ren, F. Shah, F. Tip, B.G. Ryder, and O. Chesley. Chianti: A tool for practical change impact analysis of java programs. In *Proc. of ACM Conference on Object Oriented Programming, Systems and Applications (OOPSLA)*, 2005.

[98] T. Rondeau, T. Bielawa, D. Maldonado, M.S. Hsiao, and C.W. Bostian. A methodolgy for a verifiable software platform to secure software defined and cognitive radios. In *Software Define Radio (SDR)*, 2005.

[99] H. RueB and N. Shankar. Deconstructing shostak. In *Proc. of 16th IEEE Symposium on Logic in Computer Science (LICS)*, 2001.

[100] S. Sankaranarayanan. *Mathematical Analysis of Programs*. PhD thesis, Stanford University, Stanford, California, 2005.

[101] C.H. Seger and R.E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. In *Journal of Formal Methods in System Design*, 1995.

[102] R. Shostak. Deciding combinations of theories. In *Journal of the Association for Computing Machinery*, 1984.

[103] N. Sinha. Symbolic program analysis using term rewriting and generalization. In *Proc. of Formal Methods in Computer-Aided Design (FMCAD)*, 2008.

[104] N. Sinha and E.M. Clarke. Abstraction in model checking. In *www.cs.cmu.edu/ emc/15817-s05/cegar.ppt*, 2005.

[105] G. Snelting. Concept analysis - a new framework for program understanding. In *Proc. of ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 1998.

[106] K. Thearling and J. Abraham. An easily computed functional level testability measure. In *Proc. of Intl. Test Conf. (ITC)*, 1989.

[107] F. Tip. A survey of program slicing techniques. In *Journal on Program Languages*, 1995.

[108] P. Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. In *IEEE Trans. on Software Engineering*, 2003.

[109] C. Wang, Z. Yang, F. Ivancic, and A. Gupta. Disjunctive image computation for embedded software verification. In *Proc. of ACM/IEEE Design, Automation and Test in Europe (DATE)*, 2006.

[110] C. Wang, Z. Yang, F. Ivancic, and A. Gupta. Disjunctive image computation for embedded software verification. In *Proc. of ACM/IEEE Design, Automation and Test in Europe (DATE)*, 2006.

[111] M.N. Wegman and F.K. Zadeck. Constant propagation with conditional branches. In *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 1991.

[112] D. Weise, R.F. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation. In *Technical Report MSR-TR-94-03, Microsoft Research*, 1994.

[113] M. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, Ann Arbor, Michigan, 1979.

[114] C. Wilson, D.L. Dill, and R.E. Bryant. Symbolic simulation with approximate values. In *Proc. of Formal methods in computer aided design (FMCAD)*, 2000.

[115] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2005.

[116] Yices. http://yices.csl.sri.com/tool-paper.pdf.

[117] A. Zaks, Z. Yang, I. Shlyakhter, and F. Ivancic etl. Bitwidth reduction via symbolic interval analysis for software model checking. In *Trans. on Computer Aided Design (TCAD)*, pages 1513–1517, 2008.

[118] L. Zhang and S. Malik. Conflict driven learning in a quantified boolean satisfiability solver. In *Proc. of Intl. Conf. on Computer Aided Design (ICCAD)*, 2002.

[119] L. Zhang, M.R. Prasad, and M.S. Hsiao. Incremental deductive and inductive reasoning for sat-based bounded model checking. In *Proc. of the Intl. Conf. on Computer Aided Design (ICCAD)*, 2004.

[120] L. Zhang, M.R. Prasad, M.S. Hsiao, and T. Sidle. Dynamic abstraction using sat-based bmc. In *Proc. of IEEE/ACM Design Automation Conference (DAC)*, 2005.