FocusCheck: a tool for verification and counterexample analysis of sequential C programs

Curtis William Keller

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation
Keller, Curtis William, "FocusCheck: a tool for verification and counterexample analysis of sequential C programs" (2005). Retrospective Theses and Dissertations. 19132.
https://lib.dr.iastate.edu/rtd/19132

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
FocusCheck: A tool for verification and counterexample analysis of sequential C programs

by

Curtis William Keller

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Samik Basu, Major Professor
    Robyn Lutz
    Simanta Mitra
    Ratnesh Kumar

Iowa State University
Ames, Iowa
2005

Copyright © Curtis William Keller, 2005. All rights reserved.
This is to certify that the master's thesis of

Curtis William Keller

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
DEDICATION

To my parents, for always providing support and encouragement.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS .................................................. vi

ABSTRACT ............................................................... vii

CHAPTER 1. INTRODUCTION ........................................ 1

CHAPTER 2. BACKGROUND ............................................ 4
  2.1 Abstraction ......................................................... 4
    2.1.1 Abstraction-Refinement ................................ 6
    2.1.2 Lazy Abstraction ......................................... 8
    2.1.3 Lazy Abstraction Example ............................... 9
    2.1.4 Lazy Abstraction Limitation ............................ 12
  2.2 Analyzing counterexamples .................................. 13
    2.2.1 SLAM ....................................................... 13
    2.2.2 Distance Metrics ....................................... 16

CHAPTER 3. FOCUSCHECK: A MODEL CHECKER FOR C PROGRAMS 17
  3.1 The Program Model ............................................. 19
  3.2 XTra: Representing Properties ............................... 21
  3.3 Reachability Analyzer ....................................... 24
  3.4 Creating the Focus Statement Sequence ..................... 27
    3.4.1 Data and Control Dependencies ....................... 27
    3.4.2 Program Slicing ....................................... 28
3.5 Checking for Feasibility .............................................. 30
3.6 Experimental Results ............................................... 31
3.7 Analyzing the Focus Statement Sequence ...................... 34
  3.7.1 Ranking the FSSs ............................................... 34
  3.7.2 Localizing the Program Errors ............................... 35
3.8 Graphical User Interface of FocusCheck ....................... 39
  3.8.1 Text Editor .................................................. 39
  3.8.2 Checker ..................................................... 40
  3.8.3 Help Options ................................................ 45
  3.8.4 Web Applet ................................................ 46

CHAPTER 4. IMPROVING THE SEARCH WITH HEURISTICS .... 48
  4.1 Existing Heuristics .............................................. 48
  4.2 Heuristic Approach ............................................. 50
    4.2.1 Automatically Generating Heuristic-Information ........... 50
    4.2.2 How the heuristic works ................................... 52
  4.3 Discussion .................................................... 55

CHAPTER 5. CONCLUSION ............................................. 57

BIBLIOGRAPHY .......................................................... 58
ACKNOWLEDGEMENTS

I would like to thank Dr. Samik Basu for your help throughout this process. Your wisdom, patience, understanding, and support were very appreciated. I would also like to thank Dr. Robyn Lutz, Dr. Simanta Mitra, and Dr. Ratnesh Kumar for your input into this thesis.

Thanks to Josh, Qian, and Janet for providing a good and fun atmosphere to work in. To all the people I would regularly go to lunch with thanks for all the laughs. Finally, to my sister, Katie, for all your encouragement.

"Woo hoo! Im a college man! I wont need my high school diploma anymore!" -Homer Simpson
Model checking is emerging as a promising technique for verification of large software systems. The major focus of research in this context is to address the problem of reasoning about infinite-domain variables and to provide helpful, easy-to-understand information for debugging in the event of an error. In this thesis we present the model checker, **FocusCheck**, for verification and debugging of sequential C programs. **FocusCheck** (a) constructs and applies on-the-fly data-abstraction on the push-down representation of programs, (b) identifies *Focus-Statement Sequences* from the counterexamples which define the context of errors, (c) is able to zoom in on specific program segments that most likely harbor the statements that caused the error, and (d) includes an intuitive graphical user interface which provides the user with various views of the counterexample increasing the readability of the results. We demonstrate the effectiveness of the techniques using a number of case studies. We also propose a heuristic that looks ahead at branch points with the aim to quickly discover a counterexample in fewer steps and less memory.
CHAPTER 1 INTRODUCTION

Reliability and security have been important concerns in the software development process. Traditionally, ensuring correctness of software relies on extensive testing and following rigorous software engineering principles. Model checking [CGP99], a technique to automatically verify properties of systems (hardware and/or software), has recently emerged as a promising approach to complement the existing techniques and provide a high degree of confidence on the quality of software developed. A typical model checker takes as input a model of the system to be verified and automatically determines whether the desired properties hold in the system. One of the major advantages of a model checker is its capability to provide counterexamples or error traces in the event a property is violated by the system. Such traces help developers to understand faults in the system and take appropriate corrective measures.

A number of model checkers [Hol91, McM92, BR02, HJMS03, BR00] have been developed each with its own ability of handling various aspects of behaviors (in terms of linear or branching time logic) and system state-space (e.g., explicit or symbolic). Most notable of them are Spin [Hol91] and SMV [McM92] which are developed in a university research environment and are also widely accepted in the industrial setting. These model checkers rely on the user to provide a high-level model of the system in terms of their respective specification language. These models are then verified against properties written in linear (Spin) or branching (SMV) time temporal logic.

In the recent past, model checking has also made significant inroads in analyzing the correctness of source code (e.g., Bandera [CDHR03], from Kansas State University,
Blast [HJMS03], from Berkeley, Verisoft [God03], from Bell Labs and Slam [BR02], from Microsoft Research, to name a few). The primary challenge in source-code verification is to efficiently reason about large or even infinite number of system behavioral patterns. For example, a function with a single infinite-domain input variable can potentially have infinitely many outputs. As such, all these model checkers, at their core, apply abstraction (on data and control [Lon93]) to make automatic analysis tractable.

In essence, the techniques can be classified into two classes: (a) one where programs are automatically translated into high-level models and traditional model checkers are used for verification (e.g., Bandera, Slam) and (b) the other which follows a well-known approach of abstraction-refinement and uses constraint solvers or theorem provers to aid model checkers. In this thesis, we will primarily focus on the later technique of abstraction refinement. Figure 1.1 illustrates the overview of modeling and model checking techniques as described.

**Counterexamples & Debugging.** Model checkers are often employed to detect errors rather than ensuring the absence of one. As such, generation of counterexamples is one
of the major uses of model checkers. Specifically, in program verification, presenting the user with counterexamples which can convey helpful, concise information for debugging is an extremely challenging task. A number of techniques [BNR03, GV03, Gro04, JRS02] have been proposed to analyze counterexamples and identify the root cause of errors. In spite of these advances, software verification is still in a state of infancy primarily due to complex data and control structures present in programming languages. We will present a detailed discussion of the existing techniques and their respective advantages and disadvantages in Chapter 2.

Solution Overview: FocusCheck. In this context, we present in this thesis, FocusCheck, a model checker and counterexample analyzer for sequential C programs. FocusCheck combines the power of various model checking techniques and constraint solving to verify programs which may have infinite-domain data. As alluded to before, FocusCheck makes use of the known techniques of abstraction-refinement and couples them with program analysis methodologies to help the user better understand the cause of errors in counterexamples. For the purpose of easy-usability and effective understanding of error-cause, FocusCheck is also equipped with an intuitive graphical user interface which provides various different hierarchical views of the counterexamples.

Outline. The remainder of the thesis is organized as follows. In Chapter 2, we present a brief overview of the existing techniques for program verification and counterexample analysis. Chapter 3 describes the salient features of our tool FocusCheck. In Chapter 4, we discuss future work into adding a heuristic to FocusCheck. We conclude in Chapter 5 by summarizing the contributions of this thesis followed by several future avenues of research.
CHAPTER 2 BACKGROUND

The major bottle-neck for applying model checking is caused by the state-space explosion problem. A number of approaches have been proposed, investigated and deployed with varying degree of success to combat this problem. They include abstraction, compositionality, symbolic manipulation, and pruning techniques. Abstraction \cite{Lon93} approximates the valuation of variables defining the behavior of the system. Compositionality \cite{Lon93} verifies individual components in a system and combines these results to reason about the global behavior. Symbolic verification \cite{BCM+92} alters the way state-space is represented (e.g. Binary Decision Diagrams). Pruning techniques, e.g. partial order reduction \cite{God91}, aims at reducing the search space by not exploring unnecessary/replicated patterns in system behavior. Many of these techniques are combined in various tools in the hopes of mitigating the problem \cite{God01}.

As background for the description of the research results presented in this report, in the following sections we will focus on the current state-of-art abstraction techniques applied in model checking.

2.1 Abstraction

Consider a system whose behavior depends on user inputs, e.g., a program whose loop-control parameter is determined by the user. To reason about a property of such a system, the model checker is required to analyze the program behavior for all possible (infinite) valuations of the loop-control parameter. The property may be violated after
the first-run of the loop or after one thousand runs of the loop. So it may be necessary to make certain assumptions about the loop itself in order for the property to be checked in a reasonable amount of time and memory.

A widely-used method of abstraction is to abstract the large finite domain or infinite domain data variables in the system into boolean variables [Sai00, PHR01, BR00, BR02]. This is referred to as data abstraction. [PDV01] proposed a less restricting data abstraction technique where infinite integer domains are abstracted to the domain of zero, positive, negative, and unknown. Data abstraction also requires that basic data operations be abstracted. For example, in the abstracted domain of zero, positive, negative, and unknown abstract addition is defined as: sum of two positives is positive, sum of two negatives is negative while sum of positive and negative is unknown.

The main aim of abstracting data domain and defining abstract data-operations is to generate a finite state system, $P'$, from a large, potentially infinite state system, $P$. Model checking is then done on the abstract system, $P'$. The abstraction is normally done conservatively such that if the property holds in $P'$ it will also hold in $P$. If the model checker finds a counterexample in $P'$, it is checked with respect to $P$ to ensure that the counterexample trace in $P'$ corresponds to a real trace in $P$.

Other effective techniques can be combined with abstraction to further enhance the memory management. For example, [DD02] describes the use of abstraction with binary decision diagrams. [DD02, HJMS02, CGJ+00] use formulas or transitions to map the concrete model to an abstraction and vice versa. This is known as predicate abstraction [SH97]. A set of predicates is either defined by the user or provided by automatic means. The predicates define how the model will be converted into a finite-state model of the system. The states in the abstracted model contain truth values of the predicates (true or false). For example, assume there exists a model representation of a system. Each state in the model represents a possible configuration of the system, the valuations of the local and global variables and where in the system the node currently is. The transitions
represent possible paths from the current state to a new state that adhere to the system specifications. Assume at some state in the model, x, it contains a predicate that the variable $i$ must be greater than 0. If the variable $i$ is not greater than 0 at state x then the path is spurious. The possible evaluations of the state x have been changed from infinite to two ($i$ is greater than 0 or not). Now that we have shown how the abstract system $P'$ can be constructed we can show how $P'$ can be refined to increase the chance at finding feasible traces through the concrete system $P$.

2.1.1 Abstraction-Refinement

The most traditional method for model checking with abstractions uses the abstraction-verification-refinement cycle [CGJ+00, Sai00, GPY02, PDV01, BR00, BR02, HJMS02, DD02]. Figure 2.1 illustrates this cycle. The first step of the cycle is to construct an abstraction $P'$ from $P$ where $P$ is the original abstraction. In the second step, $P'$ is verified against a desired property and if $P'$ satisfies the property the process is terminated. However, if there exists a counterexample that violates the property it needs to be determined if the counterexample is feasible in the concrete system. If it is not feasible, modify or add predicates to the system with the goal of ensuring further searches will not discover the same infeasible path. Refer to the small example from the previous section where there exists a predicate that at state x in a model that the variable $i$ must be greater than 0. Now assume that a counterexample was discovered and it is
infeasible. It was found that if \( i \) is equal to 1 at state \( x \) the counterexample is spurious. The predicate can be added or changed in order to refine the abstraction as needed. Thus, the predicate can be altered, at state \( x \) the variable \( i \) must be greater than 1.

[PHR01, PDV01] propose using simulation to discover if the counterexample is feasible or not. Others [CGJ00, DD02, Sai00] use theorem provers and/or constraint solvers to determine feasibility. If the counterexample is feasible then the abstracted model \( P' \) does not satisfy the given property; otherwise, \( P' \) is updated or refined to avoid infeasible counterexamples creating a new abstraction, \( P'' \). The refinement is either realized automatically or with user-guidance. [GPY02] uses artificial intelligence (a machine learning algorithm) to refine the abstraction. In the recent past, using the spurious counterexample to automatically refine the abstraction [CGJ00, DD02, Sai00, PDV01] has become the most common technique. The basic idea is similar to the example given above. When the model checker finds a counterexample it checks to see if this counterexample corresponds to a concrete trace. If it does not then it creates new predicates (refining \( P'' \)) using that specific counterexample trace so at least that trace will no longer be considered during the search. Once the refinement phase is finished the loop starts over at step one with the new abstracted system \( P'' \).

One of the major problems with the abstraction refinement loop is the infinite abstraction loop problem. This is where the abstraction is continually refined. This is more clearly seen in systems with loops that are capable of creating an infinite domain. For example, consider the following example where there is an assignment to the variable \( x \) which is set to 0. Following this assignment is a while-loop which runs as long as \( x = 0 \). Outside of the while loop is the error statement. Assume there are no predicates to begin with. In the first search it will find the error node since there is no predicate that keeps the search within the while loop. The model checker will then determine the abstraction is not feasible and update the predicates for the trace with one pass through the while loop to include \( x = 0 \). The algorithm will restart the search but this time it
will execute two passes through the while loop. The predicate is only for the first pass through the while loop thus the search will exit the loop after the second pass of the loop. It will then discover the same error and again update the predicates (adding \( x = 0 \)) but this time for the second pass. This series of steps will continue forever.

### 2.1.2 Lazy Abstraction

In [HJMS02] they propose a new technique called Lazy Abstraction to optimize the three-phase abstraction refinement loop. They criticize the previous method claiming it does not scale well to large systems. This method has two phases which it repeats. It first computes the abstraction by creating a tree that represents a portion of the abstracted state space. The first phase begins by doing a forward search on this tree. If more of the state space is needed in the tree it is constructed on-the-fly. If it finds an error state reachable in the tree it continues on to the second phase.

The second phase checks to see if the error trace corresponds to a concrete error trace. If the abstraction is too coarse then additional predicates are needed. These new predicates are designed using the information of the counterexample itself. The second phase works backwards from the last statement in the counterexample. It collects the instructions that if executed will lead from the current state to the error state, this is called the *bad region*. The goal is to find the first node where the intersection of the bad region and the current predicates of a node are unsatisfiable. This node is called the *pivot node*. Whichever predicate is the underlying cause of the error is the new predicate added. In other words it searches for the predicates that caused the intersection to be unsatisfiable. This new abstraction predicate is added to the nodes proceeding the pivot node up to and including the error node in order to stop the previous spurious counterexample. Once this is done it starts over from the first phase. However, the search begins from the pivot point. This method only abstracts when necessary and only refines the pieces needed. This method has been implemented in the model checker.
Example() {
1: if(*) {
7:   do {
8:     gotlock=0;
9:     lock();
   } gotlock++;
10:   if(gotlock){
11:     unlock();
12:   } while(*)
2:   do {
3:     lock();
4:     old=new;
5:     if(*){
6:       unlock();
new++;
5: } while(new != old);
6:   unlock();
7:   return;
}}

lock(){
   if (LOCK == 0){
      LOCK = 1;
   } else {
      ERROR
   }
}
unlock(){
   if (LOCK == 1){
      LOCK = 0;
   } else {
      ERROR
   }
}

Figure 2.2 Small lazy abstraction example [HJMS02]

Blast [BR00] which takes C code as input. They have recently extended this algorithm to handle concurrent systems [HJMQ03].

2.1.3 Lazy Abstraction Example

Figure 2.2 provides a small example to illustrate how the lazy abstraction idea works. Assume there are two predicates at the start, \( \text{LOCK} = 1 \) and \( \text{LOCK} = 0 \) that are already present in the nodes and correctly represent the system specifications. \( \text{LOCK} \) should be 0 whenever \text{lock()} \) is called and \( \text{LOCK} \) should be 1 whenever \text{unlock()} \) is called. The predicates should be true before that statement/node is executed. The algorithm will create a tree representing a portion of this system.

**Forward Search** The algorithm begin its first phase with a forward search on the tree
Figure 2.3 (a) First forward Search (b) Backward Analysis (c) Searching with the new predicate

in a depth-first search manner. The search will go from line 1 with the predicate \( \text{LOCK} = 0 \) to line 2 with the predicate \( \text{LOCK} = 0 \). In line 2 there is a call to \( \text{lock}() \) and the predicate for line 3 is \( \text{LOCK} = 1 \) which holds. From line 3 the search continues to line 4 with \( \text{unlock}() \) and the predicate for line 4 is \( \text{LOCK} = 1 \). The predicate for line 5 is \( \text{LOCK} = 0 \) which holds going from line 4 to line 5 because line 4 calls \( \text{unlock}() \). Since there are no predicates for new and old the search can choose either to exit the while loop or do another pass. Assume the search continues with exiting the while loop. So the search goes from line 5 to line 6 where line 6 calls \( \text{unlock}() \). Due to line 4 calling \( \text{unlock}() \) the variable \( \text{LOCK} \) is already set to 0. Thus, when line 6 calls \( \text{unlock}() \) \( \text{LOCK} = 0 \) and the search finds an error. Figure 2.3(a) illustrates the first forward search done on the example program.

**Backwards Counterexample analysis** Now we need to make sure that the counterexample is a genuine trace. Working backwards from the error state in line 6 the algorithm constructs the bad region. Going from ERROR to the start of line 6 the bad region is \( \{ \text{LOCK} = 0 \} \). Moving backwards to the start of line 5 the algorithm adds new = old to the bad region, \( \{ \text{LOCK} = 0 \land \text{new} = \text{old} \} \). From line 5 to line 4 the statements run
are `new++` and `unlock()` making the bad region `{LOCK = 1 & new+1 = old}`. The bad region stays the same from line 4 to line 3 but changes from line 3 to line 2 to `{LOCK = 0 & new+1 = new}`. As we can see these predicates are unsatisfiable due to the statement `new+1 = new`. Using a theorem prover the predicate `new ≠ old` is found to be important.

The node representing line 2 is the pivot node. The algorithm adds `new = old` to the list of predicates and refines the abstraction for the nodes from the pivot node up to the error node, nodes representing lines 3, 4, and 5. So, if the search goes from lines 3 to 4 to 5 then the predicate starting at 5 should be `{LOCK = 0 and new = old}` but if the search takes the else branch of line 3 then it goes from 3 to 5 the predicate should be `{LOCK = 1 & new = old}`. Figure 2.3(b) shows this backward analysis. Now that the predicates have been refined the search can start again.

**Searching with the New Predicate** The forward search begins again from the pivot node, line 2 which has a new predicate going into line 3 `{LOCK = 1 & new = old}`. The search will take the same path from line 3 to line 4 and line 4 to line 5 with the new predicate. When executing line 5 it will rerun the while loop due to the new predicate that `new = old` (the search will not goto line 6). Thus, the spurious counterexample is no longer considered. The search begins to run the while loop again so in the tree a branch is created from line 5 to a new node representing line 2, call this node line 2'.

It is safe to include the predicate from line 5 into the node for line 2'. The while loop of line 5 requires `new = old` to restart the loop. The predicates for the line 2' node are then `{LOCK = 0 & new = old}`. The predicates for the old node representing line 2 is `{LOCK = 0}` because the node is the pivot node and the refinement takes place after the pivot node. The reachable region of a node n is defined as the states that are reachable along the path from the root to n. Since the set of states that satisfy the reachable region of line 2' with the new predicates `{LOCK = 0 & new = old}` is a subset of the states that satisfy the reachable region with the predicates of the old node `{LOCK = 0}` the search can stop. Any error from this point on would have been found from exploring
1: \( j = 2; \)
2: \( n = 1; \)
3: \( i = 3; \)
4: while \((n > 0) \) {
5:     \( i++; \)
6:     \( \text{if}(i==j) \) {
7:         \( \text{error}; \)
8:     }  
}

Figure 2.4  Small lazy abstraction example

the old line 2 node. Thus, the node representing line 2' is considered to be covered and the search does not continue the search from that point. The search backtracks to line 3 and searches again taking the else branch and reaching the return. Since no error node is found the search backtracks even further to line 1 and searches the then branch of the if-statement starting at line 7. The search will continue and find that no error statement is reachable. Figure 2.3(c) illustrates the new search with the predicates obtained from the previous phase.

2.1.4 Lazy Abstraction Limitation

While Lazy abstraction helped to stop some forms of creating the infinite abstraction loop it is still possible to create one. For example, we can have the following example in Figure 2.4. Since \( i \) is changing in every pass of the loop it could eventually be equal to \( j \) if \( j \) is a high number thus it is not conservative to terminate the search after the first pass, the second pass, etc.. The algorithm will not terminate with this example due to the infinite-state space (the while loop at line 4) and the variable \( i \) continually changing (line 5).

The reasons lazy abstraction is considered the best choice for abstraction-refinement is the memory management (it constructs the system on-the-fly and only refines pieces of the system when needed) and the ability to stop some infinite abstraction-refinement loop examples like the previous example shown in Figure 2.2. In Chapter 3, we will
describe a different approach to abstraction.

2.2 Analyzing counterexamples

One question that has started getting more attention is what do we do after the model has been checked? If the model is large the corresponding trace to the violating state can also be large. Discovering the actual cause of the error can prove to be a time-consuming task for the user. While this topic has gotten attention it is still relatively new.

Recently work has been done using the traces from the start to an error state and comparing these against correct traces to determine the cause of an error. [JRS02] uses a game theory approach. They separate the statements that push the trace towards the error state (forced) with the statements that try to avoid the error state (free). These sets are then presented to the user. Work done by Alex Groce [GV03] computes a set of error traces and compares that set to a set of correct traces, this technique was implemented in Java PathFinder (JPF) [VHBP00]. To obtain the set of correct (positive) traces the algorithm works backwards from the end of an error (negative) trace. Using these sets it compares the commonalities and the differences between the two. The cause of the error is computed by finding all the transitions that only appear in the negative set along with the transitions that only appear in the positive set. The user is given a subset of all the states that only appeared in negative traces and a subset of all the states that only appeared in positive traces.

2.2.1 SLAM

[BNR03] proposes another solution to this problem, similar to [GV03], in which the authors implemented in their model checker SLAM [BR02]. Their technique compares a single correct trace with an incorrect trace. The algorithm uses a model checker to search
until it finds a trace to the violating state. If there are no traces to a violating state then the property is satisfied. If there is a trace it works backwards to find transitions that do not violate the property.

The model checker uses the state space already discovered during the counterexample search. Their algorithm initializes a worklist that contains all the reachable states to the end state of the trace vertex v that satisfy a correctness function f. The correctness function represents what the property should evaluate to at each step in order to avoid the error statement. This creates a list of pairs \((v, \Omega)\) where \(\Omega = f(v)\). It works backwards through the discovered state space. If the current pair being analyzed is not already in the visited list it is removed from the worklist and added to the visited list. All the transitions from that state to another, if the targets evaluation of the correctness function is true (the property is not violated at these states), are added to the worklist and the correct trace list.

For procedure calls, it works backwards the same way but has to match the call locations properly. This works in two phases, the first phase ascends from the entry point of a procedure, p, to the points that call p. The second phase descends to the procedure exit from procedure calls. Whenever the algorithm reaches a call statement it runs the second phase and whenever it reaches the top of a procedure it runs the first phase. Each phase has their own worklist they maintain. The transitions are done using call-return graphs. The end result is a correct trace. The localization of the error consists of all the statements that belong to the error trace but do not belong in any correct trace.

To show how this works consider the following small example, see Figure 2.5(a). The property to be checked is that ReleaseLock() is never called two times in a row and AcquireLock() is never called two times in a row. To help show this property assume there is a variable L that is set to false whenever the lock is acquired (after AcquireLock() is called) and it must be false whenever ReleaseLock() is called. L is
main()
1: AcquireLock();
2: if(...)  
3: ReleaseLock(); 
else 
4: ...
5: AcquireLock();
6: if(...) 
7: ReleaseLock(); 
else 
8: ...;
9: return;

1: i=0; j=0;
2: if(i \leq j) 
3: trigger=true; 
else 
4: trigger=false; 
5: if(trigger==true) 
6: error=1;

(a) (b)

Figure 2.5 Examples of Counterexample Analysis

set to true whenever the lock is released (after \texttt{ReleaseLock()} is called) and remains true until \texttt{AcquireLock()} is called. Assume the model checker found an error trace $t_e = [1, 2, 4, 5]$. There is only one trace through the program that does not lead to a violation which is $[1, 2, 3, 5, 6, 7, 9]$. The worklist is initialized to $(5, \text{L=false})$. As it proceeds backwards ensuring to take paths that will not violate the property it will discover a correct trace; $(1, \text{L=false}) \leadsto (2, \text{L=true}) \leadsto (3, \text{L=true}) \leadsto (5, \text{L=false})$. $t_c = [1, 2, 3, 5]$. The intersection of the two traces $t_e$ and $t_c$ we find that line 4 is the only line that is in $t_e$ that is not in $t_c$. Suggesting that the error is in line 4, which in fact is missing a call to \texttt{ReleaseLock()}. A halt is added to line 4 to know that that path has been searched. Invoking the model checker again it finds another error trace $t_e = [1, 2, 3, 4, 6, 8, 9]$. The worklist is initialized to $(9, \text{L=false})$. Working backwards it will discover the correct trace $[1, 2, 3, 5, 6, 7, 9]$. Line 8 is the only line that is in the error trace that is not in the correct trace. It adds a halt to line 8 and calls the model checker again to find no more errors.

Consider another example in Figure 2.5(b). Assume during the reachability section the model checker finds that error=1 in line 6. So the trace to this error state is $[1,2,3,5,6]$. Working backwards from line 6 the worklist is initialized to $(6, \text{error}=1)$...
Working backwards the algorithm discovers a correct trace; \((6, \text{error} \neq 1) \sim (5, \text{trigger} \neq \text{true}) \sim (5, \text{trigger} = \text{false}) \sim (2, i \leq j) \sim (1, i = 0; j = 0)\). Comparing this list of transitions \{\((1,2), (2,4), (4,5), (5,6)\)\} to the error trace transitions \{\((1,2), (2,3), (3,5), (5,6)\)\} the difference is \{\((2,3), (3,5)\)\}. These two transitions are considered the cause of the error and the algorithm localizes the error to line 3. However, the systems fault could be with the if statement at line 2. Instead of using \(a \leq \) it could be < and the error would be fixed. We present a different approach to focus in on the cause of the error in Chapter 3.

2.2.2 Distance Metrics

More recent work done by Alex Groce is similar to the one presented in [BNR03]. Instead of finding all the possible counterexamples and corresponding traces it analyzes one at a time. It differs from [BNR03] in that it is more concerned with determining how closely related the correct trace is from the error trace measured by using distance metrics [CGS04, Gro04]. Distance metrics are designed to quantify how closely two things are related to each other. The goal of which is to create correct traces that are more closely related to bad traces so the localization is smaller and will hopefully clearly reveal the cause of the error. The group recently developed a tool called explain [GKL04]. The tool uses CBMC as its model checker and works on ANSI-C programs. When CBMC finds a counterexample it is given to explain. Explain then finds a trace that does not violate the property that is as similar as possible to the counterexample using distance metrics. The output is displayed to the user by highlighting the statements that differ between the two traces. Program slicing is also done to remove the superfluous statements from the output. Program slicing is discussed in more detail in Chapter 3.
CHAPTER 3  FOCUSCHECK: A MODEL CHECKER FOR C PROGRAMS

We describe in this chapter, a tool called FocusCheck, for verification of sequential C programs and analysis of counterexamples. At its core, FocusCheck, is based on the abstraction-verification-refinement paradigm described in Section 2.1.1 with some differences. Below we present a brief outline of the salient features of FocusCheck.

The architecture of FocusCheck can be seen in Figure 3.1. The tool takes as input programs written in the C programming language and a property written in extended transition relations (Xtra) (Section 3.2). Xtra can be used to define sequential properties over control and data behaviors of the C program. The C program is translated into a push-down system that is readable by the XSB logic-programming language [aSBU03] (Section 3.1). FocusCheck then performs forward reachability analysis of the given program using on-the-fly abstraction of the infinite-domain variables (Section 3.3). If a counterexample is obtained, the model checker then slices (Section 3.4) the counterexample to generate a Focus Statement Sequence (FSS). The feasibility of data operations in a FSS ensures the existence of a feasible counterexample (Section 3.5). A constraint solver is therefore tightly coupled with the model checker to detect feasibility of FSS data operations. If indeed a feasible FSS is obtained, a number of post-mortem techniques (Section 3.7) are deployed to zoom in on the cause of the error which can greatly reduce the debugging effort. Finally, the model checker is equipped with an intuitive graphical user interface (Section 3.8).
Running Example. To help illustrate how FocusCheck works an example from the Blast model checker [HJMS02] will be used throughout this section, see Figure 3.2. The code represents a simple locking program. The desired property is to ensure strict alternation between invocation of the lock and unlock procedure calls. In other words, the property to be checked is the condition that the variable error is never set equal to 1. Unlock should always read the variable lock to be 1 and be able to set it to 0 and vice versa for the lock procedure. If this is not true, the error variable is set to 1 and the property is violated.
1: int error = 0;
2: Example() {
3:     lock = 0;
4:     if (*){
5:         do {
6:             if (*){
7:                 lock();
8:                 goto_lock++;
9:             }
10:         } if (goto_lock == 1){
11:             unlock();
12:         }
13:     } while (*)
14:     do {
15:         lock();
16:         old = new;
17:         if (*){
18:             unlock();
19:             new++;
20:         }
21:     } while(new != old);
22: } unlock();
23: } return;
24: }
25: lock() {
26: if (lock == 0){
27:     lock = 1;
28: } else {
29:     error = 1;
30: } }
31: unlock() {
32: if (lock == 1){
33:     lock = 0;
34: } else {
35:     error = 1;
36: } }
37: }
38: Figure 3.2 Blast Example Revisited [HJMS02]

3.1 The Program Model

The core of FocusCheck is developed in the XSB tabled logic programming environment [aSBU03]. As such the first step is to translate the programs written in C into XSB-readable terms. For this purpose we rely on CIL (C Intermediate Language) developed at UC, Berkeley [UC05] to obtain control flow graph models for C programs. The control flow graphs are then represented as XSB terms. Each procedure is defined by a term decl with four arguments: name, list of formal parameters, list of local variables and the start state of the procedure body:

\text{decl}(ProcName, Formals, Locals, StartState).
Each state in the procedure body is defined by a tuple of line number and label number and interstate transitions are represented by the term:

\[
\text{ctrans} (\text{StartLine-StartLabel}, \text{Stmt}, \text{EndLine-EndLabel}, \text{Cond})
\]

The second argument above represents the statement at the source state and the last argument captures whether the \text{Stmt} is evaluated to \text{true} or \text{false}. This is specifically useful when the \text{Stmt} is a conditional statement. For example: if the last argument is \text{true}, then the constraint in the conditional statement is assumed to be true and the program flow goes along the then-branch of the conditional block.

Variables are given a name that includes what procedure they appear in and their name. This ensures that variables with the same name but in separate procedures do not have the same name in the representation. For example, assume procedure P has a local variable i. This becomes local_P_i. \text{Stmt} are separated by their type; assignment, conditional, goto, return, etc. The \text{Stmt} contains all the information needed to analyze it properly. For an example, Figure 3.3 presents the model for the procedure \text{unlock} in Figure 3.2. \text{decl} provides the name of the function which has one local variable \text{local_unlock_error} and the first statement starts at line 35 with the label number of s40. The first statement in the program is a conditional one. If the conditional constraint is satisfied then the control flow chooses the \text{true} path to 36-s39; otherwise it goes to 38-s38. Some program statements are needed to be separated into multiple
statements. For example, assume there is a call to procedure \( p_1 \) and the return of \( p_1 \) is set to some variable \( x \) (\( x = \text{p1()} \)). Each property is given a global variable to hold the values of their returns. The procedure \( p_1 \) has a global variable, \( \text{return\_val\_fn}\_p_1 \), so the main procedure can access it and assign the variable \( x \) to the return of \( p_1 \). There is the statement to call \( p_1 \) which is followed by an assignment statement setting \( x \) to global variable representing the return of \( p_1 \).

3.2 XTra: Representing Properties

We introduce the notion of extended transition system (XTra) to represent sequential control and/or data oriented properties of C programs.

Definition 1 (XTra) Extended transition system \( E = (P, \rightarrow, S, C, \sigma, F) \) where \( P \) is the set of states, \( F \subseteq P \) is the set of final (bad) states and transition relation \( \rightarrow \subseteq P \times S \times C \times \sigma \times P \) with \( S \) is the set of program statements, \( C \) is the set of constraints over (program/property) variables and \( \sigma \) is the transfer function relating the source-state and program statements to the destination-state variables.
**Example Property.** All files opened are eventually closed. As the properties are written in terms of the translated program it takes into consideration the way file open and close statements are represented in the translation (see Section 3.1). Recall that, every open call of the form \texttt{fd=fn\_OpenCall(FileName)} is translated into two statements: \texttt{call(fn\_OpenCall, [FileName])} and \texttt{assign(fd, return\_val\_fn\_OpenCall)}. Note that we have used a generic name \texttt{fn\_OpenCall} to represent all open functions; e.g. \texttt{fn\_open, fn\_fopen, fn\_freopen} etc.

The property transition system is shown in Figure 3.4. There are 4 states \(p_1, p_2, p_3\) and \(p_4\) and each transition is labeled by a program state to be matched, the guard over program and property variables (true is the default guard) and the assignment statements on destination state variable (\(\epsilon\) represents an empty assignment).

\(p_1\) moves to \(p_2\) such that the state variable \(X\) records the type of the \texttt{OpenCall} seen. The transition from \(p_2\) to \(p_3\) is invoked if the statement assigning the return value of the call \texttt{OpenCall} (recorded as the \(p_2\)'s state variable) is visited. The return value is captured as a state variable at the state \(p_3\); note that this is the file handler of the opened file. Both \(p_1\) and \(p_2\) have self-loop transitions which are unguarded, i.e., the transitions are taken for any statement in the program and does not update any state variables. \(p_3\) moves back to \(p_1\) if \texttt{CloseCall} on the saved file handler is seen. \(p_3\) makes self-loops if the statements seen are (a) not \texttt{CloseCalls} or (b) \texttt{CloseCall} on some other file handler. Finally, \(p_3\) goes to \(p_4\) if the return statement of the main procedure is visited; this implies the program has exited and has not closed at least one of its opened files. \(p_4\) is referred to as the final state or the bad state. If the program has an execution sequence leading the \texttt{XTra} to the state \(p_4\), then that execution sequence is a counterexample witnessing a violation of the property: every opened file is closed eventually.

**Encoding \texttt{XTra} in XSB:** The transition relations for \texttt{XTra} is represented as 5-ary predicate in XSB: \texttt{ptrans}. The final or bad state is represented using the predicate \texttt{bad}. The XSB encoding of the open-close property (Figure 3.4) is presented in Figure 3.5. Each
%%% transition relations
\begin{verbatim}
pttrans(p1,call(OpenCall,-),1,[],p2(OpenCall)): opencall(OpenCall).
pttrans(p1,-,1,[],p1).
pttrans(p2(X),assign(_X1,var(return_val(X))),1,[assign(Y,var(return_val(X)))],p3(Y)).
pttrans(p2(X),_,1,[],p2(X)).
pttrans(p3(X),call(CloseCall,[F]),eq(F,X),[],p1):- closecall(CloseCall).
pttrans(p3(X),call(CloseCall,[F]),neq(F,X),[],p3(X)):- closecall(CloseCall).
pttrans(p3(_,X),assign(return_val(frunain),_),1,[],p4).
\end{verbatim}

%%% auxiliary predicates
\begin{verbatim}
opencall(fn_fopen).
opencall(fn_open).
opencall(fn_freopen).
opencall(fn_fdopen).
closecall(fn_fclose).
closecall(fn_close).
closestmt(call(Call,-)): closecall(Call).
\end{verbatim}

%%% final state
bad(p4).

Figure 3.5  Open-Close Property in XSB

\texttt{pttrans} has five arguments. The first is the current node and the fifth argument is the destination node. The second argument is the specific \texttt{Stmt} that needs to be made in order to go to the destination node, \_ is used to represent any \texttt{Stmt}. For example, it can be a call to open a file. The third argument is a set of constraints/stipulations on the variables in order for it to move to the destination node. 1 represents there is no guard to be checked. In the example, we want the property to ensure that the arguments in the close called matches the file opened. The fourth argument contains the transfer functions. It contains any information that the property needs to remember at its destination node, [] is used if there are none. This is seen from the property transition from $p_2$ to $p_3$. This records the file handler for the file opened to be ensure the file handler closed is the same.
Looking at the example in Figure 3.2 the property to check is if error is ever set to 1. This property has two states and is shown in Figure 3.6. The property remains at state p1 until the local variable error is set equal to 1. If the property goes to p2 it reaches the bad state and a counterexample has been found.

### 3.3 Reachability Analyzer

Reachability analysis involves on-the-fly construction of the product of the program and the property under consideration. It generates all possible counterexamples, i.e., sequences of program statements that lead the property to its bad or final states. Note that programs in our context may contain (recursive) procedure calls and returns. As such the reachability analyzer must keep track of an execution stack in terms of return location in order to match correct calls and returns along the analysis path. We use the techniques of push-down system model checking to meet this requirement. Below we present the definition of push-down system used in the reachability analysis of C programs. [BSLS03]

**Definition 2 (Push-Down System Representation)** A push-down system representation of a program is a tuple \( PDS = (G, \Gamma, \rightarrow) \) where \( G \) is the set of global variable valuations, \( \Gamma \) is the set of program statements and the local variable valuations in the scope of the statement and finally \( \rightarrow \) is the transition relation of the form:

1. \( (g, \gamma) \rightarrow (g, \epsilon) \) if \( \gamma \) corresponds to a return statement: returning from the procedure destroys the local store.
2. \( \langle g, \gamma \rangle \leftarrow \langle g, \gamma_1, \gamma_2 \rangle \) if \( \gamma \) corresponds to call to a procedure \text{procName}, \( \gamma_1 \) represents the start statement of \text{procName} and \( \gamma_2 \) represents the return location in the caller.

3. \( \langle g, \gamma \rangle \leftarrow \langle g', \gamma' \rangle \) if \( \gamma \) is not a call or return statement and \( g' \) and \( \gamma' \) represents the new valuation of global variables and next statement with new local variables respectively.

Every state in a push-down system is a tuple of global variable valuations and a stack where each transition is invoked based on the current top-of-stack content.

**Push-down System Example.** Referring to the blast example in Figure 3.2 we can see how a push-down system can represent a program. The control location will include the global integer variable error, \( G = \{ \text{error} \} \). The stack alphabet includes all the states which in our context would be the tuple of line numbers and the local variable valuations \( (\text{lock}, \text{got.lock}, \text{old}, \text{new}) \). For example, the transition from statement at Line 3 to Line 4 will be represented as the push-down transition rule:

\[
\langle \{0\}, (3, \{ \text{unk}_1, \text{unk}_2, \text{unk}_3, \text{unk}_4 \}) \rangle \leftarrow \langle \{0\}, (4, \{0, \text{unk}_2, \text{unk}_3, \text{unk}_4 \}) \rangle
\]

In the above, the valuation of all the local variables at Line 3 are “unknown.” Furthermore, each of the variables have different unknown valuations (\text{lock} is equal to \text{unk}_1, \text{got.lock} is \text{unk}_2, while \text{old} and \text{new} are equal to \text{unk}_3 and \text{unk}_4 respectively. After the transition, at Line 4, the valuation of global variable \text{error} remains unchanged along with local variables \text{got.lock}, \text{new} and \text{old}. The valuation of \text{lock} gets updated to 0.

Note that in the example, we have used line numbers instead of program statements as described in the definition assuming that there is a one-to-one mapping between the two. In the actual setting, we generate a new number - \textit{labels} - which are distinct for each program statement.

Given the property transition relations \( \longrightarrow \) and push-down transition relation \( \leftarrow \), the transitions \( \longrightarrow_p \) for the product of push-down representation and property is defined
as follows:

\( \langle p, g, \gamma \rangle \xrightarrow{p} \langle p', g', \omega \rangle \iff \exists p^{m,c,\sigma} p'. \exists \langle g, \gamma \rangle \xrightarrow{\langle g', \omega \rangle}. \text{match}(m, \gamma) \land \text{eval}(c, g, \gamma, p) \)

where \( \omega \) is either \( \epsilon, \gamma_1 \gamma_2 \) or \( \gamma' \) as per Definition 2, \text{match} returns true only when \( m \) can be unified with the statement represented by \( \gamma \) and \text{eval} return true if the constraint \( c \) can be satisfied under the valuations in \( g, \gamma \) and \( p \).

**XSB Encoding of the Reachability Analysis:** The above transition relation is realized as Prolog-rules in XSB. Below we present one such rule corresponding to the assignment statement.

\[
\text{systrans}((L1-La1, \text{assign}(X, \text{Exp}), L2-La2):S1-S2-C-no,
\quad \langle(L2-La2, \text{Stmt}, L3-La3):\text{Stemp}-S2-C-T], P1, P2) :\]

\[
\text{eval}\_exp(\text{Exp}, S1, \text{ExpVal}),
\]

\[
\text{assign}(S1, X, \text{ExpVal}, \text{Stemp}),
\]

\[
\text{move}\_prop(P1, S1, P2),
\]

\[
\text{ctrans}(L2-La2, \text{Stmt}, L3-La3, T).
\]

Above, \( S1 \) and \( S2 \) represent the global and local variable valuations before an assignment statement has been executed. The predicate \text{eval}\_exp evaluates the valuation of the expression \text{Exp} in the context of the current variable store and the resultant value \text{ExpVal} is assigned to \( X \) via the \text{assign} predicate. A new store \text{Stemp} is obtained after the update to \( X \). The property state change is obtained using the predicate \text{move}\_prop. Finally the next statement in line is obtained by expanding the \text{ctrans} relation from \( (L2, La2) \) (tuple of line and label number).

**Data Abstraction:** Programs may contain infinite domain input variables. Reachability analysis is performed statically, and, as such, the run-time valuations of these variables will be unknown. In other words, the analysis must present results for all possible valuations of the input variables. To handle such a situation we use abstraction [Lon93] (see Chapter 2). Reachability analysis is performed without interpreting the unknown valuations of the infinite domain variables and as such expressions over these uninterpreted variables are also left uninterpreted. This leads to an abstract model
which has more behavior than the original program. If a counterexample is generated in the abstract program, the sequence of operations can be fed to a constraint solver (e.g., CLP(R) [aSBU03]) to check whether it is feasible in the concrete program. We will further present in the subsequent sections optimizations to check for feasibility of counterexamples.

**Example of data abstraction.** Returning to the blast example shown in Figure 3.2, the variables new and old are unknown. There is a while loop at line 22 which relies on the valuations of these variables. The analyzer leaves this statement uninterpreted and the search chooses a path nondeterministically. Based on the path chosen there is a new constraint on the valuations of new and old (either new==old or new != old).

### 3.4 Creating the Focus Statement Sequence

Once the reachability analyzer has found a counterexample, it is given to the slicer. The goal of the slicer is to include only those program statements that directly or indirectly lead to the cause of the counterexample. Slicing can reduce the size of the counterexample trace for the user to examine [BSS04]. Furthermore, the constraint solver is able to check for feasibility of this smaller trace. It is also possible to remove some counterexample traces if their slices end up being identical.

#### 3.4.1 Data and Control Dependencies

Typically, program statements are not independent. A program statement evaluation and how the program got to a certain point is usually dependent on previous statements. Data dependencies consist of the variables in assignment statements that directly or indirectly effected the variables in the goal statement. Control dependencies are the conditional statements that effected the path to the goal statement. An example of a data dependency would be: the statement $a = b + c$ is data dependent on the assignment
statements of $b$ and $c$. On the other hand, if the statement $a = b + c$ is nested within an if statement $(d > e)$ then the statement $a = b + c$ is dependent on the outcome of $d > e$.

### 3.4.2 Program Slicing

Program slicing [Wei84] is a widely-used program analysis technique used in compilers and debuggers. The central theme of slicing is to obtain a subsequence of a given sequence using certain slicing criteria. In the current context, a counterexample sequence is retraced in reverse direction from the last statement that lead the property to a bad state. Statements are identified as members of a slice if they directly or indirectly affect the last statement of the counterexample or any property transition. Such a slice is referred to as a *Focus Statement Sequence* (FSS). To identify the dependencies and focus statements, we maintain a worklist of control and data dependencies while performing backward reachability from the last statement.

The data dependency worklist maintains a set of variables whose valuations affect the focus statements already identified in backward traversal, while the control dependency worklist maintains a set of conditional statement labels which affect the control flow leading to one or more previously-classified focus statements. The criteria for classifying a statement as a focus statement is as follows:

1. A statement is a focus statement if it is responsible for moving the property from one state to another.

2. An assignment/call statement is a focus statement if it sets a variable which is present in the data dependency worklist.

3. A conditional statement is a focus statement if its statement label is present in the control dependency worklist.
4. A conditional statement is a focus statement if it is reached (via backward traversal) from its then(else) branch where the else(then) branch includes an assignment to a variable present in the data dependency worklist.

The corresponding updates to control and data dependency list:

1. If the statement is classified as a focus statement because of data dependency, then the variable set at the statement is removed from the data dependency worklist. The control and data dependencies of this statement are added to corresponding work lists.

2. If the statement is classified as a focus statement because of control dependency, then its label number is removed from the control dependency worklist. The control and data dependencies of this statement are added to corresponding work lists.

The backward traversal terminates when either the search has analyzed all of the statements within the counterexample or when the data and control worklists are empty.

**Example of Slicing.** Assume that a counterexample was found in the unlock procedure in the blast example in Figure 3.2. Assume the counterexample runs `lock()` at line 7 and then exits the while loop to run `lock()` again at line 16. The counterexample is \(1, 3, 4, 6, 7, 28, 29, 8, 10, 13, 16, 28, 31\). Line 31, `error = 1`, is the last statement in the counterexample found. This statement will initiate the slicing criteria. This statement moved the property from \(p_1\) to the bad state, \(p_2\), thus Line 31 is added to the FSS and `error` is added to the data worklist. Since Line 31 is within the if statement at Line 28 (`lock == 1`) it is added to the control worklist. Line 28 is analyzed and as it appears in the control worklist it is added to the FSS, removed from the control worklist, and the variable `lock` is added to the data worklist. Line 10 is an if statement which does not appear in the control worklist however the then-branch not taken has an assignment to a variable in the data worklist (see item 4 above) so Line 10 is added to the FSS and
got_lock is added to the data worklist. Line 8 is included in the FSS and got_lock remains in the data worklist, got_lock is on both sides of the assignment. Line 29 is added to the FSS due to lock being present in the data worklist. Line 28 (lock == 1) is again added to the FSS, removed from the control worklist, and lock is inserted back to the data worklist. Lines 7 refers to the calls to the procedure lock(). Since Lines 28 and 29 are classified as focus statements, the if statement at line 4, if(*), is added to the control worklist and the FSS. Finally, line 3 removes lock from the data worklist and is itself added to the FSS. The variable got_lock is left in the data worklist. Lines 1 and 2 do not update got_lock. This results in all the statements in the counterexample being analyzed and the slicing terminates. The slice of the counterexample, FSS, is ⟨3, 4, 6, 28, 29, 8, 10, 28, 31⟩.

Analyzing multiple FSSs can also help reduce the number of counterexamples. If, after slicing multiple counterexamples, some of the FSSs are the same only one needs to be kept. For example, there can be two counterexamples that have different paths but the statements that differentiate them do not directly or indirectly effect the last statement. This will result in the same FSS and therefore one of the counterexamples can be discarded.

3.5 Checking for Feasibility

We stated in Section 3.3 that reachability analysis and counterexample (FSS) generation is performed on the abstract program model, i.e., a model where all the infinite domain input variables and their associated operations are left uninterpreted. As a result, a counterexample obtained in the abstract model may not be a valid or feasible counterexample in the concrete model. In other words, the sequence of counterexample operations is not feasible for any valuation of the uninterpreted variables. It can be easily shown that feasibility of FSS operations ensures the presence of a feasible coun-
terexample. As the length of the FSS is potentially smaller than the corresponding
counterexample, the sequence of operations to be checked for feasibility is also less, thus
reducing the feasibility checking overhead.

For the purpose of identifying feasibility/infeasibility of counterexamples/FSS, we use
a built-in constraint solver CLP(R) in XSB. The sequence of operations are restructured
as a conjunction of constraints over variables and fed to CLP(R). If it returns true,
the sequence of operations is feasible; otherwise, the counterexample/FSS is discarded.
For example: if the sequence of operation is \( x>y; x=z; z=y; z>x \), the corresponding
constraints will be

\[
X > Y \land X_1 = Z \land Z_1 = Y \land Z_1 > X_1
\]

where \( X \) represents the initial value of \( x \) and \( X_1 \) represents its valuation after the
assignment statement \( x=z \). Similarly for \( y \) and \( z \). The example constraint is not satisfiable
proving that the sequence of operations is infeasible.

**Assumptions in a Feasible FSS.** In Section 3.3 we show that when searching for an
error some conditionals are left uninterpreted. They are based on unknown or input
variables. A counterexample may rely on these unknown variables and if so there can
be assumptions/constraints placed on these variables. Each counterexample is given a
set of assumptions on the unknown variables that are assumed to be true in order for
the counterexample to be feasible. To show this refer to the blast example, the variable
\texttt{got_lock} is unknown. Consider a FSS \( \langle 1, 3, 10, 11, 35, 38 \rangle \). This trace is dependent on
the unknown variable \texttt{got_lock} set to 1 because of the conditional at Line 10. Thus,
the assumption set for this trace includes \texttt{got_lock} = 1.

### 3.6 Experimental Results

We experimented with freely available Linux Core utility programs. The property
used is the open-close property discussed in Section 3.2: every open file is eventually
Table 3.1 Verification of Linux Core Utilities using the openclose property: single counterexample

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>CFG</th>
<th>Cex</th>
<th>FSS</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>871</td>
<td>786</td>
<td>200</td>
<td>13</td>
<td>4.22</td>
</tr>
<tr>
<td>*cksum</td>
<td>344</td>
<td>255</td>
<td>49</td>
<td>7</td>
<td>1.00</td>
</tr>
<tr>
<td>cut</td>
<td>755</td>
<td>966</td>
<td>66</td>
<td>8</td>
<td>0.87</td>
</tr>
<tr>
<td>*comm</td>
<td>271</td>
<td>344</td>
<td>36</td>
<td>9</td>
<td>54.09</td>
</tr>
<tr>
<td>md5sum</td>
<td>687</td>
<td>941</td>
<td>78</td>
<td>7</td>
<td>0.71</td>
</tr>
<tr>
<td>od</td>
<td>2007</td>
<td>2558</td>
<td>240</td>
<td>8</td>
<td>3.82</td>
</tr>
<tr>
<td>tac</td>
<td>718</td>
<td>631</td>
<td>107</td>
<td>9</td>
<td>20.79</td>
</tr>
<tr>
<td>*tsort</td>
<td>591</td>
<td>614</td>
<td>127</td>
<td>8</td>
<td>3.46</td>
</tr>
</tbody>
</table>

closed. Table 3.1 corresponds to the case, where FocusCheck generated a single feasible focus statement sequence. The first column presents the program name, the second gives the total lines of code in the program while the third provides the size of the corresponding control flow graph. The counterexample length and the corresponding focus statement sequence length are presented in the fourth and fifth columns respectively. Finally, the time required to generate the feasible FSS is shown in the last column. Note that the FSS length is significantly less than the corresponding counterexample showing the usefulness of slicing. The ratio of FSS length to counterexample length is lowest for the program od: 1/30.

Table 3.2 presents the results for programs for which FocusCheck was able to generate all the feasible FSSs. Note that this is a significantly more expensive task as the entire program needs to be explored. The second column shows the ratio of the number of feasible counterexamples to the number of feasible FSSs. It can be seen that the number of FSSs is much less than that of counterexamples (e.g., 5% for cksum). We also present in the third and fourth column the FSS length for the maximum size counterexample and the counterexample length for the maximum length FSS respectively.

We have also verified a portion of the Traffic Alert and Collision Avoidance System (TCAS) [RTC90] which is used in most commercial aircrafts to issue traffic advisories
to pilots when one or more aircrafts come in close proximity. We concentrated on the
Resolution advisory (RA) module of TCAS which is used to identify the safest maneuver
for the controlled aircraft in the context of various parameters: relative position of the
intruder aircraft, flight trajectory, minimum protected zone of each aircraft, etc. The
RA module sets a variable altsep to either UPWARD_RA or DOWNWARD_RA depending on
whether the safety action of the controlled aircraft is to move to a higher or lower altitude
respectively. [BSS04].

FocusCheck is used to analyze TCAS using different valuations of altsep as the error
condition. FocusCheck automatically identifies the pre-conditions on input parameters
necessary (as assumptions) for specific valuations of altsep. The assumptions generated
exactly match the pre-conditions necessary for the correct functioning of the RA module.
How FocusCheck generates these assumptions will be discussed in the following section.
Specifically the pre-conditions for altsep=UPWARD_RA are

otherTrackedAlt > ownTrackedAlt,
upSeparation > downSeparation,
downSeparation < positiveRAAltThresh

On the other hand, the pre-conditions for altsep=DOWNWARD_RA are

otherTrackedAlt ≤ ownTrackedAlt,
upSeparation ≤ downSeparation,
downSeparation ≥ positiveRAAltThresh

Table 3.3 summarizes the results of FocusCheck on TCAS.
Table 3.3 Verification of the Resolution Advisory Module of TCAS

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>CFG</th>
<th>No. of Cex/FSS</th>
<th>Max. Cex Vs. Max FSS</th>
<th>Len</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>193</td>
<td>282</td>
<td>16/8</td>
<td>114-63</td>
<td>46.84</td>
<td>40.75</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16/8</td>
<td>119-66</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.7 Analyzing the Focus Statement Sequence

We further order and analyze multiple FSSs to help the user understand the cause of errors in the program. Two specific methods are described in this section: Ranking FSS according to their complexity (Section 3.7.1) and identifying program blocks that are most likely to harbor the error in the program (Section 3.7.2).

3.7.1 Ranking the FSSs

A single bug in a program may generate multiple counterexamples/FSSs. The FSSs are ranked on the basis of their length and the number of variables in their respective assumptions sets. The shorter FSSs are given higher priority. If two or more FSSs have the same length their rank is then determined by the number of variables in their assumption set.

Definition 3 (Rank) Given two FSSs $F_1$ and $F_2$, $F_1$ is said to be at higher rank than $F_2$, denoted by $F_1 \geq F_2$, if:

1. the length of $F_1$ is less than that of $F_2$ or

2. the length of $F_1$ is equal to the length of $F_2$ and the number of variables in the assumptions of $F_1$ is less than the number of variables in the assumptions of $F_2$.

Ranking FSSs facilitate rapid debugging of the code in an iterative manner. FSSs at higher ranks are more likely to be conceptually easier to understand and debug.
Debugging a higher ranked (simpler) FSS may solve all the problems and save a large amount of time and effort.

### 3.7.2 Localizing the Program Errors

FocusCheck can also localize the error into a specific program segment included within the FSSs. This subset of the FSS is called the *neighborhood of error statements* (NEST). The goal is to zoom in on a subsequence of the FSS in order to find the exact cause of the error. The approach is based on analyzing the common aspects of multiple FSSs and creating a reduced set of FSSs.

**Definition 4 (Reduced Set of Focus Statement Sequences)** A set of focus-statement sequences \(\{F_1, F_2, \ldots, F_n\}\), where each \(F_i\) is paired with assumption set \(A_i\), is said to be reduced if the following conditions hold:

1. If \(c\) is a constraint in \(A_i\), then \(\neg c\) is either not present in any \(A_j\) or is present in at least two \(A_j\) (\(j \neq i\))

2. \(\forall i, j(i \neq j) \Rightarrow (F_i \neq F_j \lor A_i \neq A_j)\)

3. A sequence of statements \((s_{i1}, s_{i2}, \ldots, s_{in})\) is marked in each FSS \(F_i\) such that the outer-most conditional expression over input variables in \(F_i\) cannot be evaluated using the constraints present in \(A_i\).

The first item eliminates complementary assumptions from sets using the observation that “if a constraint \(c\) and its negation \(\neg c\) appear in exactly two distinct assumption sets, then \(c\) and \(\neg c\) are most likely generated from the same conditional statement which has exactly one FSS for each of its branches,” [BSS04]. There are three possible ways with which it can hold.

1. **The error statement is followed by a conditional.** That is the faulty code happens before the conditional statement. Assuming this causes the assertion
violation, the assignment may affect statements in both branches of the condition. This will result in two FSSs being generated. The constraint and its negation appear in two separate assumption sets.

2. **The error statement is in a conditional expression.** The faulty code is within the condition expression. For example, if (faulty code). Again for the same reasons as in 1 above this will create 2 FSSs.

3. **The error appears in both branches of a conditional.** There are error statements contained within both branches of a conditional block.

   If there are two assumption sets that contain a constraint c and its negation that satisfy the above cases then it is possible to remove c and its negation from the sets. The constraint pair must appear in exactly two assumption sets. This is because if there is more than one constraint c or \( \neg c \) then it is most likely the errors are appearing in both branches of the conditional block c/\( \neg c \), item 3 above. We wish to localize the error inside the conditional block. These constraints are needed to correctly create the NEST which is described below. Removing the constraints may result in not localizing the error to include the conditional block. Therefore, in this case we do not remove the constraints c and \( \neg c \).

   Removing a constraint and its negation from two separate assumption sets can lead to those FSSs being identical. Thus, one of them may be removed. This technique can reduce the total number of FSSs.

   **FocusCheck** uses the algorithm Reduce shown in Figure 3.7. The algorithm removes complementary assumptions and then from this result removes the duplicate FSSs. With the assumption sets **FocusCheck** projects the assumption sets onto their corresponding FSS. This resulting subsequence is the *neighborhood of error statements* (NEST). The projection is done by a forward analysis of the FSS. Each statement in the FSS is analyzed with the assumption set obtained from the previous statement. The assumption
Input: A set $S$ of FSSs $F_1, F_2, \ldots, F_n$ and their corresponding assumption sets $A_1, A_2, \ldots, A_n$.
Output: $\text{Reduce}(S)$

1. Initially $\text{Reduce}(S) = S$. Repeat steps 1 and 3 till no change in $\text{Reduce}(S)$.
2. If there exists a constraint $c$ in a unique $A_i$ and its negation $\neg c$ in a unique $A_j$, $i \neq j$, then delete $c$ from $A_i$ and $\neg c$ from $A_j$. Iterate this step until no such $c$ is found.
   (reduction by eliminating complementary assumptions)
3. If there exists in $\text{Reduce}(S)$ identical FSSs $F_i$ and $F_j$ with identical assumption sets $A_i$ and $A_j$, $i \neq j$, remove any one of those FSS-assumption set pairs from $\text{Reduce}(S)$.
   (reduction by eliminating identical FSS-assumption pairs)
4. Project each $A_i$ in $\text{Reduce}(S)$ to its corresponding FSS $F_i$ as follows.
5. Start with statement $s_k$, $k = 1$, the first statement in $F_i$ and repeat the following cases.
   (a) $s_k$ is a conditional statement with conditional expression $c$:
      i. if $c \notin A_i$ or $\neg c \notin A_i$ then mark all the focus statements in the block containing $s_k$ and go to step 4
      ii. else $k++$
   (b) $s_k$ is an assignment statement $x = y$ (call statements are considered as assignments of actual parameters to the corresponding formal parameters)
      i. if $\exists c \in A_i$ involving $y$ then add the new constraint over $x$ in $A_i$ by replicating constraints over $y$ and replacing $y$ by $x$ in the replication. $k++$
      ii. if $\exists c \in A_i$ involving $x$ then delete $c$ from $A_i$. $k++$
   (c) If $s_k$ is the last statement of $F_i$, mark the entire $F_i$; go to step 4.

Figure 3.7 Reduce Algorithm [BSS04]

set is updated if the current statement being analyzed affects any of the constraints in the set.

The first statement is analyzed with the assumption set obtained from when it was checking for complementary constraints mentioned above, the original assumption set. The algorithm then identifies the outermost conditional statement where the condition generated unimportant constraints (the condition that does not update any constraints in the assumption set). Then, it marks all the focus statements in the current scope of that conditional statement as the NEST. In the worst case the NEST will be the entire FSS but in the best case it will point to a single line that if changed will completely remove the error. Projecting the NEST onto multiple FSSs will usually result in a
providing good insight into the cause of the error.

**Localization Example.** Consider the example shown in Figure 3.8. The program takes as input five integers. The program is to sort the five integers in descending order, \( a_1 \geq a_2 \geq a_3 \geq a_4 \geq a_5 \). The program consists of a number of if statement blocks. At the end of the program stating at line 62 is an if statement that checks if they do satisfy the property and if not set error equal to 1. Running this code in FocusCheck with the property that error should never be 1 will return two FSSs. \( FSS_1 = \langle 6, 9, 41, 53, 56, 62, 63 \rangle \). \( FSS_2 = \langle 6, 9, 41, 53, 54, 62, 63 \rangle \). The assumption set for \( FSS_1 \) is \( A_1 = \{ a_1 > a_2, a_3 > a_4, a_1 > a_3, a_5 \geq a_3, a_3 \geq a_2, a_5 \geq a_1 \} \) and the assumption set for \( FSS_2 \) is \( A_2 = \{ a_1 > a_2, a_3 > a_4, a_1 > a_3, a_5 \geq a_3, a_3 \geq a_2, a_1 > a_5 \} \). \( FSS_1 \) has \( a_5 \geq a_1 \) in its assumption set and \( FSS_2 \) has \( a_1 > a_5 \) in its assumption set. The analyzer will remove these two assumptions from each set, \( A_1 = \{ a_1 > a_2, a_3 > a_4, a_1 > a_3, a_5 \geq a_3, a_3 \geq a_2 \} \) and \( A_2 = \{ a_1 > a_2, a_3 > a_4, a_1 > a_3, a_5 \geq a_3, a_3 \geq a_2 \} \).

The algorithm then projects \( A_1 \) onto \( FSS_1 \). The algorithm starts at the start of \( FSS_1 \) in line 6. Line 6 contains the first 3 assumptions found in \( A_1 \) so the algorithm moves to the next statement in the trace. The conditional statements in lines 9 and 41 are also included in \( A_1 \) but the conditional for line 53 is not. All of the focus statements that lie within this conditional (line 53) are marked as the NEST, lines 53 and 56 in this case. \( FSS_2 \) works in the same manner but the NEST for \( FSS_2 \) includes lines 53 and 54. The algorithm suggests that there is something wrong within the \( \langle 53, 54, 56 \rangle \) block. The error is in fact in line 53 the if-statement should read \( a_1 > a_5 \) instead of \( a_1 < a_5 \). Changing this line removes the error and the program is verified.

We can see from the example that each FSS was 7 lines in length and the localization for each FSS was only 2 lines. The localization greatly reduced the amount of information needed for the user to work through in order to find the bug in the program.
1: int main(){
2: int a1,a2,a3,a4,a5;
3: int o1,o2,o3,o4,o5;
4: int error=0;
5: //input a1,a2,a3,a4,a5
6: if(!(a1 > a2)\&\&(a3 > a4) \&\&(a1 > a3)) {
7:   exit(0);
8: }
9: if(a3 > a5){
10:   o1=a1;
11: if(a4 > a5){
12:   if(a2 > a3){
13:     o4=a4,o5=a5;
14:   }else{
15:     o2=a2,o3=a3;
16: }else{
17:     o2=a3,o3=a2;
18: }
19: o2=a3,o3=a4;
20: if(a2 > a5){
21:   o4=a2,o5=a5;
22:   else
23:     o4=a5,o5=a2;
24:   }
25: }/* line 11 */
26: if(a2 > a5){
27:   o4=a5,o5=a4;
28:   if(a2 > a3){
29:     o2=a2,o3=a3;
30:   else
31:     o2=a3,o3=a5;
32: }else{
33:   o2=a3,o3=a5;
34: if(a2 > a4){
35:     o4=a2,o5=a4;
36:   else
37:     o4=a4,o5=a2;
38: }else{
39:   o4=a4,o5=a2;
40: }else{
41:   if(a2 > a3){
42:     o4=a3,o5=a4;
43:   if(a5 > a2){
44:     o3=a2;
45:   }if(a5 > a1)
46:     o1=a5,o2=a1;
47:   else
48:     o1=a1,o2=a5;
49: }else{
50:   o1=a1,o2=a2,o3=a5;
51: }else{
52:   o3=a3;
53:   if(a1 > a5)
54:     o1=a1,o2=a5;
55:   else
56:     o1=a5,o2=a1;
57:   if(a2 > a4)
58:     o4=a2,o5=a4;
59: else{
60:   o4=a4,o5=a2;
61: }endif((o1 < o2)||((o2 < o3)|| (o3 < o4)||(o4 < o5)){
62:   error=1;
63: }
64: }
Figure 3.8  Sorting numbers

3.8 Graphical User Interface of FocusCheck

The goal of the GUI was to make a simple and intuitive program in order for the user to debug errors efficiently. The application window is split into two major panes. The left most pane is the text editor. This is where the source code can be edited. Line numbers also accompany this code and are seen in their own pane to the left of the text editor. The right most pane is where the counterexamples are displayed. This was developed using Java Swing [Mic05].

3.8.1 Text Editor

The text editor consists of two tabs. The first one is for the C source file, the program to be checked. This will be converted automatically into the XSB readable predicates as shown in Section 3.1. The second file is the property file. This file contains the property
that is to be checked. This property file must be written as a series of predicates as shown in Section 3.2. The property file should follow the naming convention. For example, if the C file is called blast.c the property file should be called blast_prop.P.

The normal file options common to most text editors can be used. FocusCheck can open, save, save as, or create a new file. When opening the file blast.c FocusCheck will also attempt to open blast_prop.P. When new is chosen it will leave two blank text tabs. If save as is chosen it will only ask for one name and automatically save the property file, this way it is easier to abide by the naming rules. The text editor has some simple commands such as copy, cut, and paste. The line numbers left of the text editor will change accordingly to the length of the file. These options can be found in edit drop down menu in the main tool bar of FocusCheck. See Figure 3.9 for a screen shot of FocusCheck.

3.8.2 Checker

Now that FocusCheck has a program source file and property file, it can check the program. In the main tool bar there is a checker drop down menu with two options, Run and Remove Highlights. The Remove Highlights will remove all highlights that appear in both the line number and text editor panes. When the Run option is chosen a new window will appear called Run Checker Options.

Run Checker Options. This window has three text boxes and a Run button, see Figure 3.10. The first box is for the maximum number of FSSs to be displayed. If there are many to display the user can choose a smaller, more manageable number. It will pick the FSSs with the best rank first, the ranking is described in Section 3.7.1. If it is left blank it will display all of the FSSs available. The second and third boxes are for the filename and directory in which the C file and property file are contained. This will automatically be filled in if the user has saved or opened an existing file. The run button will run the script 'verify' with the arguments filename and file directory, for example
verify blast.c blast-ex will run the FocusCheck on the file blast.c which is located in the blast-ex directory.

Once the script has finished running, the FSSs discovered by FocusCheck will be displayed in the right most pane of the FocusCheck display. A tab is created for each FSS, the total number of which was either chosen by the user or all of the FSS found by FocusCheck. If the property passes, there are no counterexamples, then the pane will create a tab that says “verified.”

FSS. Every FSS has a number and color associated with it. The color is shown in the
Some colors are reused if there are many FSSs, the colors yellow, red, and gray are reserved. The FSSs are presented in order of their rank (see Section 3.7.1). Each FSS consists of a list of line numbers and an assumption set as described in Section 3.7.2. The line numbers are displayed in a list menu on each tab. By selecting a line number it will highlight both the line in the list menu as well as the corresponding line in the C source file in yellow. By pressing the up and down keys it will move from one line to the next line. This allows for easy navigation of the FSS with regards to the actual source code file. Yellow is reserved for navigating the FSSs. In Figure 3.11 the second line of the FSS is selected and the corresponding line, 22, is highlighted in the source file.

There are also four buttons along with the line number list in the tabbed window: assumptions, localize, show all, and remove highlights. The assumptions button opens a new window displaying the assumptions or constraints necessary for the feasibility of the particular FSS. This button will run the script called ‘post’ if it has not already been run, the work in ‘post’ is described in Section 3.7.2. This is separate from the ‘verify’ script in order to help split up the work. ‘Post’ is only run once and does the analysis for all of the FSSs present. Figure 3.12 shows the blast example’s assumption set for the first FSS.

The localize button will also run the ‘post’ script if it has not already been run.
Figure 3.11 Traversing the FSS

The localize button highlights the line numbers to the left of the source code red where the localization takes place. The 'post' script identifies specific areas in an FSS in which the user should concentrate their attention. This is the neighborhood of error statements (NEST) as described in Section 3.7.2. These highlights can only be removed from the remove highlights option in the Checker drop down menu. In Figure 3.13 the line numbers on the right most side are highlighted red to represent the NEST.

The showall button will highlight all the lines of the particular FSS in the source code window the FSS color. The highlights will stay on the source code as the user
moves from one FSS to another. When another FSS is chosen to ‘showall’ FocusCheck will highlight that FSSs lines with its FSS color on top of the previous FSSs highlights. Any lines that they share will be changed to a gray color highlight. For example, assume there is an FSS which has a green color and has been selected to show all highlights on lines 1,2,3,and 4. Assume there is another FSS with the color blue and its trace is 1,2,5,and 6. When both of these are selected to show all lines 1 and 2 will be colored gray, lines 3 and 4 will be colored green and lines 5 and 6 will be colored blue. This allows the user to view multiple FSS’s at the same time with regards to the source code. It also shows lines that are common to multiple FSS’s which may aid the user in finding the cause of the violation. Figure 3.14 shows the source code when both FSS 1 and 2 have both chosen the showall button. Lines 22 and 23 are highlighted gray because both FSSs share those common lines. Line 8 is highlighted yellow because that is where the current focus of the user is at. Line 10 is highlight green because it belongs in the FSS for 2 but is not shared with FSS 1.

Finally, the remove highlights button will remove all the highlights that were created by that FSS when the showall button was pressed. That is it will remove the highlights in the source code of the lines of the trace for that specific FSS. If two FSSs share the same line that is colored gray and one of the FSSs is removed, the color will change
back to the FSS that is to remain highlighted. If there are more than two FSSs sharing the same line and one is removed it will stay gray until only one FSS has that line highlighted. This button does not remove the highlights created by the localize button.

3.8.3 Help Options

There are two help options. The first is option help and the second is a quick guide. The option help will spawn a new window with what each button and menu choice does, similar to the information presented in this section. The quick guide option has a list of
five easy steps to quickly begin using FocusCheck and to get an understanding of the tool. The steps include opening a file, selecting the run option, ensuring that the options are correct and clicking run, the panel on the right displays the multiple FSS that the user can navigate, and finally the various options that each FSS can employ.

3.8.4 Web Applet

To give an example on how FocusCheck works the program was converted to an applet and placed on the web at the following address:
http://www.cs.iastate.edu/~cwkeller/applet/focusexamples.html

Due to issues with XSB not working on the web server it was not possible to run the scripts correctly. Thus, the examples had to be separated and have the results preloaded. There are about five examples with most of the functionality of FocusCheck. However, when Checker then Run is selected the run checker options box does not appear but instead just shows the results of all the FSSs. The results are not generated by the model checker but rather are loaded from the directory which already has all the results. The user is not able to edit the source code. The options described in Section 3.8.2 and in Section 3.8.3 above works the same as described.
CHAPTER 4 IMPROVING THE SEARCH WITH HEURISTICS

The aim of FocusCheck is to model check programs with infinite-domain data and to analyze counterexamples such that errors can be easily and efficiently debugged. The main techniques applied in FocusCheck are (a) on-the-fly abstraction, (b) slicing, and (c) error localization. However, the simple depth-first search procedure of FocusCheck may still be expensive owing to the presence of a large number of paths to explore. In this chapter, we describe a road-map to develop a heuristic-based search procedure which will guide the model checker to explore the relevant paths, with respect to the property, before examining the paths that are less likely to lead to an error. The primary objective is to find shorter counterexamples efficiently.

4.1 Existing Heuristics

Heuristic based model checking is traditionally referred to as directed or guided model checking. [YD98] provides four possible heuristics to help find errors in a system. The first heuristic is target enlargement. The error states are made larger by including adjacent states that are within one step of the error state constructing a larger preimage of the error state. The goal is that the search will have more of a chance to find a path to the error state and decrease the searching time. The second heuristic is the implementation of Hamming Distance [Ham50]. The states that are the most similar to the error states are explored first. Tracks creates a set of preimages that are based
on a subset of the variables in the system. Using this set of tracks a smaller version of the system is searched for the error. Finally, *guideposts* are user defined tags that help lead the search to the violation state. The guideposts are conditions the user feels are necessary for the assertion to be violated. They claim using a combination of target enlargement, tracks, and guideposts provide the best results. However, [RE99] claims that this method’s heuristic is too weak and requires too much manual guidance to provide much of a benefit. [ELL01, RE99] describe a scenario in which the heuristic estimates the distance from the current state to an error state within binary decision diagrams. The states that will most likely lead to an error are expanded first with the hope of discovering the error states quickly. The user can provide a formula which the heuristic uses to search the model otherwise one is automatically generated. The approach is implemented in the SPIN model checker [Hol91]. However, the algorithm relies on knowing the “BDD goal of the erroneous states” [RE99] as input. The user can specify which states in the promela code are dangerous with a tag. [BRS00] uses the guidepost technique in CTL model checking, they require the user to provide hints in order for the guided search to work on binary decision diagrams. The search works backwards from the error state and uses the hints to guide it.

The problem with these approaches is that they deal strictly with models and not with the actual program structure which does not match the ideas presented in Chapter 3. The goal of the heuristic presented in this work is to focus primarily on an actual program. In addition, the approach presented in this chapter does not rely on knowing which states in the program are the erroneous states or having user-defined guideposts before searching. The user is not required to give any additional information that is not already presented to **FocusCheck**, i.e., the property and the program to be checked.
4.2 Heuristic Approach

The heuristic requires four pieces of information that can be automatically extracted from the program and the property to be verified: a call graph [Ryd79], an assignment graph, the property transition weights, and the domain of the conditionals.

A call graph of a program captures the relationship between procedures, e.g., procedure \( P \) calls procedure \( Q \) at line number 24. An assignment graph is a subset of the program control flow graph identifying only the statements that assign/set a pre-specified set of variables. The domain of conditionals in a program defines the boundary of if-then-else statements. Finally, property transitions are weighted on the basis of their respective distance from the bad or final state. In the following Section 4.2.1, we describe each of these features in details.

4.2.1 Automatically Generating Heuristic-Information

The call graph consists of what procedures call other procedures. For example, assume the \texttt{main} procedure calls procedures \( P \), \( Q \), and \( R \). The graph will include \texttt{main} as the root node with branches to nodes annotated by \( P \), \( Q \) and \( R \). The nodes also include the line/label numbers in which the procedures are called.

The assignment graph includes all the statements that may effect a subset of variables. In the current context, the variables are the ones present in the property. In other words, we extract all the statements from the program that set a variable present in the property. Each node in the graph represents one of these assignment statements and the inter-node transitions are the abstraction of the program path to another assignment statement. The assignment graph starts at the beginning of the program, in most cases within the main procedure. Each node includes the statement and the line/label number. Consider the example program in Figure 4.1(a) and the variables of interest are \( a \) and \( b \). The corresponding assignment graph is shown in Figure 4.1(b).
1: \( x = 0; \)
2: \( a = 2; \)
3: \( b = 3; \)
4: if\( (a < b) \) {
5: \( b = a + 1; \)
6: \( x = b + a; \)
7: while \( (a < 10) \) {
8: \( a = a + 1; \)
9: \( x = x + 1; \)
10: }
11: }
12: else {
13: \( x = a; \)
14: \( b = x + b; \)
15: }
16: \( a = 10; \)

(a)

Figure 4.1  (a) Example Program.  (b) Assignment Graph

The domain of conditional statements are the line/label numbers from the start of a conditional to the end and can be easily identified. For example in Figure 4.1(a), the domain of conditional statement at Line 4 is 4–11 for the then-branch and 12–15 for the else-branch. If there is a conditional with a then-branch but no else-branch the domain of the else-branch is nothing. Therefore, the then-branch will be the path chosen unless it will result in the property following an infinite transition.

Property transition weights, as stated before, are the shortest distance of each transition from the bad or final state of the property. Consider the weighted property transition graph in Figure 4.2. Assume node 1 is the start state of the property and node 6 is the bad state. The branches are labeled by how far they are away from node 6. The branches that either backtrack to a previous node (for example, node 5 to node 1) or are self-loop transitions are given a weight of infinity. These branches are the least ideal as they do not progress the property towards the bad state. The search gives priority to the transitions with lower weights. Transitions inherit the lowest possible number of
steps to the bad state (excluding the two infinity cases). From node 1 to node 5 the weight is 2. However, from node 5 to node 3 the weights is 3. There is still a possible path from node 1 to node 6 in 2 steps thus the weight from node 1 to node 5 is 2.

4.2.2 How the heuristic works

The forward search of the program is done recursively with the program, property and the associated heuristic graphs being initialized to their respective start states. With every forward move of the program and the property (see Section 3.3, Chapter 3), the corresponding moves of the associated assignment/call graph states are computed; specifically, an assignment statement may move the assignment graph to a new state, while a call statement may move the call graph to a new next state. In short, the product of the program, property and the heuristic graphs is computed on-the-fly.

The most important aspect of the heuristic is the handling of conditional statements - note that conditional branching is the primary cause for making the search procedure inefficient. For each conditional statement, we quantitatively analyze the likelihood of each branch to lead to an error statement.
\( H1 = \text{checkbranch}(\text{thenBlock}, \text{assigngraph}, \text{callgraph}, \text{property}); \)

\( H2 = \text{checkbranch}(\text{elseBlock}, \text{assigngraph}, \text{callgraph}, \text{property}); \)

The heuristic function \text{checkbranch}, Figure 4.3, searches in the order of the property weights. From the current state of the property, the transition with the lowest weight is considered first. \text{checkbranch} determines if a transition in the property matches any node in the call graph or assignment graph whose linenumber/label number lies within the branch domain (inside the \text{thenBlock} or the \text{elseBlock}). \text{checkbranch} returns the distance of line/label numbers of the corresponding statements that lie within the then and else blocks from the start of their respective conditional blocks. The search proceeds via the then (else) branch if the \text{checkbranch} for \text{thenBlock} (\text{elseBlock}) is smaller than that for \text{elseBlock} (\text{thenBlock}). It is worth mentioning here that, in some cases, the heuristic is capable of avoiding unnecessary searching along program paths which will never progress the property towards the error state. If the search procedure reaches a dead end, the procedure backtracks to branch points it did not explore and proceeds from that point. Note that, any movements within the assignment, call, and property graphs are undone as needed when backtracking.

Recall that the example code in Figure 4.4 requires strict alternation in the invocation of procedures \text{lock()} and \text{unlock()}. Therefore, the negation of the property represents the case where there are at least two calls to \text{lock()} (\text{unlock()}) without any intermediate calls to \text{unlock()} (\text{lock()}); see Figure 4.5(a). There are no variables the property is dependent on therefor an assignment graph is not needed.

The search begins at line 1. At line 4 is an if statement with a then-branch (lines 5-14) and no else-block. Referring to the property either a call to \text{lock()} or \text{unlock()} is needed to progress the property. Using the call graph shown in Figure 4.5(b) \text{lock()} is called at lines 7 and 16 and calls to \text{unlock} are called in lines 11, 19, and 23. The difference from the start of the first block, 5, to the call \text{lock}, 7, is 2. Since the result of
int best = ∞;
for each transition, j, in the property in order of increasing weight
excluding weights that are ∞
{
    For each transition, i, from the current node in the assign. graph that satisfies j
    {
        if (line number of i < best and i is within the branch domain)
        {
            best = transitions line number;
        }
    }
    For each transition, i, from the current node in the call graph that satisfies j
    {
        if (line number of i < best and i is within the branch domain)
        {
            best = transitions line number;
        }
    }
    if (best ≠ ∞)
        return (best - start of the branch domain);
}
return ∞;

Figure 4.3 The checkbranch procedure

checkbranch on the first block did not return infinity it is the path chosen. The search
continues and discovers another conditional at line 6 without an else-branch. The same
call to lock() is seen which lies within the domain (7-9) and the then-branch is taken.
The search continues with the call to lock() which moves the current position of the
property to node 2. The search continues into and back out the lock() procedure, we
will skip the this and other procedure calls as nothing of interest happens in regards to
this example. The search continues at line 8 which does not move the position of any
graphs. Line 10 is another conditional without a else-branch and a then-branch that
contains a call to unlock(). The checkbranch procedure will be run on the then-branch
to discover that this will take a property path with infinity weight, the path back to
node 1. Thus, the then-branch is not taken. The while loop is analyzed next. While
loops can be split into two statements, for the example there is an if statement with a
goto line 5 as its then body and a goto line 15 as its else body. Since, the then and
```c
int error = 0;
Example() {
    lock = 0;
    if (*){
        do {
            if (*){
                lock();
                got_lock++;
            }
            if (got_lock == 1){
                unlock();
            }
            } while (*)
            do {
                lock();
                old = new;
                if (*){
                    unlock();
                    new++;
                }
            } while(new != old);
            unlock();
        return;
    }
}
```

Figure 4.4 Small lazy abstraction example [HJMS02]

else of this conditional statement does not contain any current property transitions the choice is done nondeterministically. In either case the search will discover the bad state. If it takes the else-branch there is a call to `lock()` at line 16. If it takes the then-branch it will analyze the conditional at line 4 again. `checkbranch` will return 1 as the call to `lock()` (line 5) matches a property transition from node 2 to node 4. Thus, in either case the property will move to the bad state and a counterexample is found.

### 4.3 Discussion

The basic premise of the heuristic is to peek ahead at branch points in order to discover an optimal path that will cause the property to progress further toward the bad state in the least amount of steps with minimal work. By giving priority to property
transitions that have a shorter distance to the violating state, the search is likely to reach a bad state in fewer steps consuming minimal time and memory.
CHAPTER 5  CONCLUSION

We discussed in this report a model checker, FocusCheck, for sequential C programs. The model checker combines the strengths of logic programming (backtracking) and program slicing and is equipped with a smart graphical user interface which enables easy viewing of counterexamples. The experimental results presented in the report provide a strong testimony of the practical applicability of the model checker. Finally, we have also investigated a heuristic-based approach to further optimize the search mechanism of the model checker and presented a detailed description of the algorithm and the associated data structures.

Some of the future avenues of research include: (a) incorporating the proposed techniques in the domain of object oriented languages like C++, Java, (b) investigating the role of slicing in analyzing counterexamples for multi-threaded programs, and (c) enhancing the GUI to navigate multiple C programs.

FocusCheck can be downloaded from

BIBLIOGRAPHY


[ELL01] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed explicit model checking with HSF-SPIN. In SPIN ’01: Proceedings of the 8th


