CMC: A MODEL CHECKER FOR NETWORK PROTOCOL IMPLEMENTATIONS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Madanlal Musuvathi
February 2004

© Copyright by Madanlal Musuvathi 2004
All Rights Reserved

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David L. Dill
(Principal Advisor)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dawson Engler

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mihalis Yannakakis

Approved for the University Committee on Graduate Studies:

To my parents

Abstract

Complex systems have complex errors. Real systems have a variety of mishandled corner cases triggered by intricate sequences of events. In practice, this leaves a residue of errors that cause system crashes but only after days or weeks of continuous execution. When detected, such problems are often very difficult to diagnose because the errors are not reproducible and the sequence of events leading to them cannot be reconstructed.

Formal verification methods are a possible way to find and diagnose such deep errors. One option is explicit model checking, which systematically enumerates the possible states of the system. A basic model checker starts from an initial state and recursively generates successive system states by executing the nondeterministic events of the system. States are stored in a hash table to ensure that each state is explored at most once. This process continues either until the whole state space is explored, or until the model checker runs out of resources. When it works, this style of state graph exploration can achieve the effect of impractically massive testing by avoiding the redundancy that would occur in conventional testing.

Conventional model checkers usually assume that the design is described at a high level that abstracts away many details of the actual implementation. Verifying actual code using such a tool requires reconstructing this abstract description from the code. This process requires a great deal of manual effort, hampering the use of model checking in actual system design. Moreover, human errors in the manual abstraction process result in missing bugs and cause false alarms during verification, further increasing the cost and reducing the usefulness of model checking. Such errors can be introduced both when constructing the model and as a result of *drift* as the

actual system evolves.

This thesis proposes CMC (C Model Checker) to address these issues. CMC works on unmodified C or C++ implementations without resorting to any intermediate description. Like traditional model checkers, CMC explores large state spaces efficiently by storing states and achieves the equivalent of executing astronomical numbers of tests in reasonable time. By not requiring any high-level model of the code, CMC considerably reduces the manual effort involved in model checking real systems. More importantly, it finds the bugs that are actually *in* the implementation: it does not miss implementation bugs that would be omitted from a model, nor does it create false reports of errors that appear in the model but *not* in the implementation.

The primary goals of CMC are to *scale* to large systems and to *effectively* find errors in such systems. However, CMC is not able to provide any correctness guarantees about the input system. While providing such guarantees is desirable, doing so for even moderately sized systems is prohibitively difficult. Instead, CMC focuses its resources to enhance its scalability and effectiveness. For instance, CMC stores only a small signature of each state in the hash table rather than storing the entire state. This enables CMC to handle many orders of magnitude larger states but at the risk of missing errors due to hash collisions. Similarly, CMC employs many approximate, yet powerful state transformations techniques to improve its state space search. Such techniques are not available to model checkers that focus on absolute correctness.

While the approach of CMC is applicable to system software in general, it has been validated only on network protocol implementations. Network protocols are especially suited to model checking as they are mostly self-contained systems where few external, well defined inputs trigger many interleaving behaviors. The thesis applies CMC to two different network protocols of varying complexity: AODV, a loop free routing protocol for ad-hoc networks and TCP, the most widely used transport protocol in the Internet today.

CMC checked three different publicly available implementations of the AODV protocol and found 34 unique errors at a rate of roughly one bug per 300 lines of code. CMC also found an error in the underlying AODV specification.

For the TCP protocol, CMC checked the Linux TCP implementation from the

stable kernel version 2.4.19. This implementation is around $50,000$ lines of code. It is mature, thoroughly tested by its actual deployment in the Internet, and is visually inspected by many in the open source community. CMC found 4 errors in this implementation where it fails to satisfy the TCP specification. Also, CMC achieved over 90% protocol coverage while model checking this implementation. These results demonstrate that CMC is able to scale to systems as large and complex as the TCP implementation and successfully find errors in them.

Acknowledgments

The work that led to this thesis would not have been possible without the guidance and support from many teachers, colleagues, friends, and family. First and foremost, I like to thank David L. Dill for being a wonderful adviser. Over these years, Dave has been very patient while I tried different things to figure out my research interests. I am deeply indebted for his constant guidance and encouragement throughout my Ph.D.

During my thesis work, I received a lot of guidance from Dawson R. Engler. If not for his eagerness and enthusiasm, CMC would not have come this far. His insights and timely sparks have helped the design and implementation of CMC in significant ways.

I like to specially thank Mihalis Yannakakis for his valuable discussions regarding CMC and for providing useful feedback on my thesis on a very short notice. I also like to thank Jerome H. Friedman and John Mitchell for participating in my Oral Examination Committee.

As with all large projects, CMC involved the work of many people. David Park contributed a lot during the many discussions I had with him and he implemented some key modules in CMC. The core idea of directly model checking C code came during a discussion with Andy Chou while working together on a different model checking effort. Both David Park and Andy Chou helped me write the papers on CMC. Rafael Hernandez, Priyank Garg, Rushabh Doshi, Adam Simpkins, Russell Greene, and Anthony Hui willingly endured CMC in its prototype form.

On the personal front, I like to thank Kavitha for her constant support. She sweetly kept tab on me during the painstaking period of writing this thesis.

Finally, my research was partially supported by Gigascale Silicon Research Center (GSRC) MARCO Grant No: SA3276JB and Stanford Network Research Center (SNRC) Grant No: 127S018.

Contents

	v
Abstract	**vii**
Acknowledgments	**xi**
1 Introduction	**1**
1.1 Introduction	1
1.1.1 Network Protocols	1
1.1.2 Model Checking	3
1.1.3 Model Checking Implementations	4
1.1.4 Challenges	5
1.2 Related Work	6
1.2.1 Traditional Model Checking	6
1.2.2 Software Model Checking	7
1.2.3 Automatic Model Extraction	8
1.2.4 Static Analysis	8
1.2.5 Correct Protocol Synthesis	9
1.2.6 Testing Network Protocols	9
1.3 Contributions of the Thesis	10
1.4 Overview of the Thesis	11
2 CMC Design	**13**
2.1 Introduction	13

2.1.1 Chapter Overview 14
2.2 Basic Design Principles 15
2.2.1 Emphasis on Scalability 15
2.2.2 Focus on Error Detection 15
2.2.3 Minimize False Behaviors 15
2.2.4 Minimize Manual Interaction 16
2.3 Building a Model from the Implementation 16
2.3.1 A Running Example 16
2.3.2 Processes and Threads 17
2.3.3 Transitions 19
2.3.4 The Environment Model 19
2.3.5 Modeling Nondeterminism 20
2.3.6 Model Building Summary 22
2.4 Model Checking Algorithm 23
2.4.1 Capturing States 23
2.4.2 Generating the Initial State 24
2.4.3 Generating a Successor State 24
2.4.4 Generating all Successor States 25
2.4.5 State Space Search Algorithm 25
2.5 Checking Correctness Properties 27
2.5.1 Generic Properties 29
2.5.2 Model Specific Properties 29
2.5.3 Producing Error Trace 30

3 Effective State Space Search **33**
3.1 Introduction 33
3.1.1 Chapter Overview 33
3.2 Managing Large States 34
3.2.1 Resource Constraints During Model Checking 34
3.2.2 Hash Compaction 36
3.2.3 Incremental States 36

3.2.3.1 The Basic Idea 37
3.2.3.2 Monitoring Writes 39
3.2.3.3 Managing State Object Granularity 40
3.3 Managing Large State Spaces 41
3.3.1 A Good Model Design 42
3.3.1.1 Down-scaling 42
3.3.1.2 Avoiding Model Redundancy 43
3.3.1.3 Avoiding Extraneous Information from the Environment 43
3.4 Heap Canonicalization 44
3.4.1 The Heap Graph 45
3.4.2 Previous Approaches 46
3.4.2.1 Lerda and Visser's Algorithm 46
3.4.2.2 Iosif's Algorithm 47
3.4.3 Incremental Heap Canonicalization 49
3.4.3.1 Access Chains 50
3.4.3.2 BFS Access Chain 51
3.4.3.3 Relocating Objects in the Canonical Heap 51
3.4.4 Implementation Details 55
3.4.5 Some Extensions 55
3.4.6 Extracting the Heap Graph 56
3.5 Automatic Data Slicing 58
3.6 Heuristic Search 59

4 The AODV Case Study 61
4.1 Introduction 61
4.1.1 Chapter Overview 62
4.2 Description of the AODV Protocol 62
4.3 The AODV Implementations 65
4.4 The AODV model 66
4.4.1 Processes and Transitions 66
4.4.2 The Environment Model 67

4.4.3 Correctness Properties . . . 67
4.5 Dealing with State Space Explosion . . . 68
4.6 Results . . . 69
4.6.1 Memory errors. . . . 70
4.6.2 Unexpected messages . . . 72
4.6.3 Invalid messages . . . 72
4.6.4 Routing loops . . . 74
4.6.5 The Specification Bug . . . 74
4.7 Comparison with Static Analysis . . . 76
4.7.1 Overview of MC . . . 76
4.7.2 Results . . . 78
4.7.2.1 Generic Properties . . . 78
4.7.2.2 Protocol Specific Properties . . . 79
4.7.3 Summary . . . 80

5 Environment Modeling - The TCP Case Study 83
5.1 Introduction . . . 83
5.1.1 Chapter Overview . . . 84
5.2 Difficulties in Environment Modeling . . . 84
5.2.1 Environment Modeling Overview . . . 84
5.2.2 Defining the System Boundary . . . 85
5.2.3 Closing the System . . . 86
5.2.4 Providing Input Triggers . . . 87
5.3 Lessons: Use Well-Defined Interfaces . . . 87
5.4 The Linux TCP Model . . . 89
5.4.1 Running the Entire Kernel in User Space . . . 89
5.4.2 Processes, Threads and Transitions . . . 89
5.4.3 Environment Model . . . 90
5.4.4 Correctness Properties . . . 91
5.4.4.1 TCP-specific Assertions . . . 91
5.4.4.2 Kernel Resource Leaks . . . 91

5.4.4.3 TCP Specification Conformance 92
5.5 Refining the Environment Triggers 93
5.5.1 The Search Effectiveness Metrics 93
5.5.2 Iterative Environment Refinement 94
5.6 Results 95

6 Conclusion 99
6.1 Summary 99
6.2 Future Work 100
6.2.1 Increasing the Scope of CMC 100
6.2.2 Making CMC More "Push Button" 100
6.2.2.1 Reusable Environment Models 100
6.2.2.2 Automated Environment Refinement 101
6.2.3 Applying Static and Dynamic Analysis 101
6.2.4 Heuristic Search 101

Bibliography 103

List of Tables

4.1 The set of event handlers used in AODV model checking. 66
4.2 Properties checked in AODV. 67
4.3 Lines of implementation code vs. CMC modelling code. 68
4.4 Number of bugs of each type in the three implementations of AODV. The figures in parenthesis show the number of bugs that are instances of the same bug in the mad-hoc implementation. 69
4.5 Comparing MC and CMC 78

5.1 Coverage achieved during model refinement. The branching factor is a measure of the state space size. 94

List of Figures

2.1 A simple routing protocol implementation. 18
2.2 Implementation of `malloc()` in CMC. `malloc()` nondeterministically fails to allocate memory. 21
2.3 Multiple calls to `cmc_choose` not necessarily lead to exponential repetition of a transition 26
2.4 Pseudocode for the CMC model checking algorithm. 28

3.1 The start state and the successor state of a transition can share objects not modified in a transition 38
3.2 Two heap states that are behaviorally equivalent, and their common heap graph 44
3.3 Iosif's heap canonicalization algorithm. 48
3.4 The incremental heap canonicalization algorithm 54
3.5 The improved heap canonicalization algorithm 57

4.1 Node a initiates an RREQ for destination d. 64
4.2 Node b responds with an RREP. 64
4.3 State after Node a receives an RREP. 64
4.4 Node b sends an RERR message when it detects a link failure to Node d. 65
4.5 Mishandled `malloc` failure: if `malloc` fails, the loop will exit after allocating less than `rerrhdr_msg.dst_cnt` buffers. The two errors are in code that assumes `rerrhdr_msg.dst_cnt` buffers were allocated. Both lead to segmentation faults. 71

4.6 Two bugs: an unexpected message and an invalid route response. (a) An unexpected route-response (RREP) message causes `getentry` to return null, crashing the machine. (b) If a route returned by `getentry` has been invalidated the hopcount will be 255. However, the code does not check for this and sends the message. 73
4.7 The mad-hoc implementation compares the hop counts of the route received in RREP and its existing route in the routing table. AODV specification requires that this comparison be done *after* an increment 74
4.8 The specification bug: the sequence number from an incoming message is used without validation, causing "time" to go backwards when messages are reordered. Fortunately, while the error was not obvious (surviving 6 rounds of specification revisions) the fix is trivial. 75

5.1 The Linux TCP implementation (version 2.4.19) does not honor the RST flag in the SYN_RECV state unless the ACK bit is set. The bug was inadvertently introduced while fixing a "harmless and rare" case. The code prematurely returns when the ACK field is not set on an incoming packet. 96

Chapter 1

Introduction

1.1 Introduction

Ensuring the correctness of large and complex systems is a prime goal of computer science research. With the increase in raw computing power, our inability to build systems that are reliable and robust, and provide sufficient confidence about their correctness is the key obstacle in making computer systems more prevalent. As a result, a large body of work has focused on techniques that aid in building systems that function correctly.

1.1.1 Network Protocols

This dissertation specifically deals with the correctness of a special but important class of systems involving a network protocol. A network protocol is a set of rules commonly agreed by two or more nodes when communicating with each other over a network. This communication usually happens to achieve some goal, such as a reliable transfer of a file from one node to another, or the detection of a route to some node in the network. A network protocol is commonly specified as a finite state machine that the communicating nodes should implement. At each state of the state machine, the protocol specifies the actions necessary to respond to various external events, such as network packets from other nodes in the network, inputs from the users of the

protocol, and timeouts. The ultimate purpose of such a protocol specification is to guarantee that any node seamlessly communicates with other nodes in the network and achieves the desired goal, as far as all nodes involved implement the specification.

There are two correctness requirements on a network protocol. First, the protocol specification should be correct. Specifically, the rules provided by the protocol should be sufficient to guarantee that *any* system implementing these rules achieves the desired goal of the protocol under *all* well-behaved circumstances. The task of determining if a protocol is correct in this sense is known as *protocol validation*. Second, the particular protocol implementation used should correctly implement the specification. Implementing a protocol typically requires figuring out some details left open in the specification, interacting with other protocols or modules coexisting in the implementation, handling various implementation-specific error scenarios, and finally optimizing the code for good performance. In the end, it is necessary to ensure that the implementation still meets all the requirements of the protocol specification. Determining if a given implementation does so is known as *protocol conformance* checking.

The correctness of networking protocol specifications and their implementations is important. Network protocols are beginning to be used at the core of many critical services where failure is unacceptable. Also, protocol implementations are the first target of external security attacks. An error in a network protocol implementation can have wide ranging effects; it can reduce the performance of a network service, compromise or crash a machine, and in the worst case bring down the entire Internet.

Unfortunately, network protocols are difficult to design and implement correctly. They involve complex interactions among multiple nodes across a network and deal with many nondeterministic events. This makes the protocol behavior very difficult to reason about. The protocol designers and implementors must carefully handle all possible events in all possible protocol and network states. To complicate matters further, most network protocols have to handle various network failures such as packet losses or link failures, and be robust against malfunctioning or rogue nodes in the network.

The currently practiced methods for protocol validation and conformance checking

are various forms of *simulation* and *testing*. While these are effective in the very early stages of protocol development, their effectiveness drops very quickly as the easy bugs are detected and fixed. Network protocols inherently have a variety of corner cases triggered only by intricate sequences of events. But, both simulation and testing can only explore a minute fraction of all possible event sequences. As a result, even heavily-tested systems can have a residue of errors that take days or even weeks to arise, making them all but impossible to replicate. When detected, such problems are often very difficult to diagnose because the errors are not reproducible and the sequence of events leading to them cannot be reconstructed.

1.1.2 Model Checking

Formal verification methods are a possible way to find and diagnose deep errors [64, 67, 76] in systems. One attractive option is model checking, a fully *automatic* verification technique that searches for error states in a given system by systematically enumerating the possible states of the system. A basic model checker starts from an initial state and recursively generates successive system states by executing the nondeterministic events of the system. Model checkers employ various mechanisms to avoid the inherent redundancy in such an exhaustive search. For example, states can be stored in a hash table to ensure that each state is explored at most once. The search continues either until the model checker finds a state that violates some correctness property, or the whole state space is explored, or until the model checker runs out of resources. In the case when an error state is found, the model checker reports the error along with a trace that can later be used to reproduce the error.

Model checking is sometimes used to prove that a system satisfies a specified property. However, it is often more practical to use it as a bug-finding method. When model checking is applicable, it can be more effective than conventional testing methods in discovering bugs because of its thoroughness at exploring the state space of the system, including rare corner cases that might otherwise be overlooked. Model checking can be more efficient than random testing because the former searches each state at most once.

Model checking works well for mostly self-contained systems where few external inputs can trigger many event interleavings. Network protocols fit this model very well: they are driven primarily by internal protocol state, are typically control-dependent and respond to a small but well defined set of nondeterministic events from its environment. Not surprisingly, a huge body of work deals with model checking network protocols (such as [47]).

1.1.3 Model Checking Implementations

Despite their advantage, model checkers are rarely used in practice. Conventional model checkers usually assume that the design is described at a high level that abstracts away many details of the actual implementation. Verifying actual code using such a tool requires reconstructing this abstract description from the code. This process requires a great deal of manual effort, hampering the use of model checking in actual system design. Moreover, human errors in the manual abstraction process result in missing bugs and cause false alarms during verification, further increasing the cost and reducing the usefulness of model checking. Such errors can be introduced both when constructing the model and as a result of *drift* as the actual system evolves [23]. For these reasons, it is a notable curiosity when software is model checked rather than an everyday occurrence.

These problems can be overcome if model checkers check the system implementation directly. This thesis presents CMC, one such model checker. CMC (C Model Checker) works on unmodified C or C++ implementations and explores the state space of the *implementation* without resorting to any intermediate description. Like traditional model checkers, CMC avoids redundancy by storing states and achieves the equivalent of executing astronomical numbers of tests in reasonable time. However, CMC does not require writing a separate high-level model of the code, nor extracting such a model from an implementation. More importantly, it finds the bugs that are actually *in* the implementation: it does not miss implementation bugs that would be omitted from a model, nor does it waste the user's time with bugs that appear in the model but *not* in the implementation. Also, a subtle advantage of model checking

a network protocol implementation directly is that protocol validation can be done *simultaneously* while checking for protocol conformance.

While the approach of CMC is applicable to system software in general, it has been validated only on network protocol implementations. For reasons mentioned earlier, network protocols are especially suited to model checking. This thesis applies CMC to two different network protocols of varying complexity: AODV, a loop free routing protocol for ad-hoc networks and TCP, the most widely used transport protocol in the Internet today. CMC found many errors in all the implementations of the two protocols checked, some of which are non-trivial and difficult to find by any other method. Also, CMC found an error in the AODV *specification* that escaped ten revisions of the specification and prior formal attempts [6] to verify the protocol, clearly demonstrating the benefit of using an implementation for protocol validation.

1.1.4 Challenges

Model checking implementations definitely comes with its challenges. The most apparent problem is the *state explosion problem*; the state space of an implementation might be extremely large or even infinite. While all model checkers have to address this problem, CMC faces a bigger challenge. An implementation has a lot more detail than necessary to check the properties of the system. This extraneous detail unnecessarily increases the size of the state space. With no access to an abstract description of the system, CMC has to manage such large state spaces with effective techniques and meaningful trade-offs.

Another problem is that CMC has very little control over the implementation that it checks. Most protocol implementations are never written with model checking in mind. So, the formalism that CMC supports should be general enough to encompass the different styles in which network protocols are implemented. Specifically, well known formal models, such as *guarded transitions* might not be applicable. Moreover, an implementation might use arbitrary data structures for efficiency, which might not be amenable for formal treatment. Also, CMC has to support the well-known idiosyncrasies of the C language, the most popular language for implementing network

protocols.

While CMC reduces the amount of work required by checking the implementation directly, the user is still required to provide a model of the environment. The environment model contains all the relevant aspects of the network and the operating system that the protocol being checked interacts with. Such a model is necessary to avoid false error reports resulting from illegal inputs or state changes that would never occur in actual system execution. Providing an environment model for a large and complex system can require considerable effort.

By addressing the challenges described above, this thesis presents an approach to pragmatically apply model checking to find complex errors in actual protocol implementations.

1.2 Related Work

This dissertation touches a large number of fields. It borrows many successful ideas and extends previously known techniques from a wide range of research areas. An attempt has been made to quote related work at relevant parts of the thesis. This section surveys only the work related to the central topic of this thesis: the correctness of network protocols in particular and computer systems in general.

1.2.1 Traditional Model Checking

The basic idea of using state graph search to verify network and communication protocols is quite old, dating back to at least 1978 [40, 82]. In recent decades, model checking has made significant progress in tackling the verification of complex, concurrent systems [20]. Tools such as SMV [54], SPIN [48], and Murphi [29] have been used to verify hardware and software protocols by exhaustively searching the state space. By caching states and employing sound state reduction techniques, these tools can detect non-trivial bugs.

The drawback of traditional model checkers is that the system to be verified must be modeled in a particular description language, requiring a significant amount of

manual effort that can easily be error prone. CMC was specifically designed with the goal of reducing the amount of work that is required to go from software development to systematic verification.

1.2.2 Software Model Checking

Some recent formal verification tools have already used the idea of executing and checking systems at the implementation level. Verisoft [37], for instance, systematically executes and verifies actual code and has been used to successfully check communication protocols written in C.

However, Verisoft does not store states and can thus potentially explore a state more than once. This problem is alleviated to some degree by partial order reduction, a sound state space reduction technique implemented in Verisoft that eliminates the exploration of redundant interleavings of transitions created by commutative operations. Nevertheless, this technique requires hints to be provided by the user and/or some static analysis of the code to determine dependencies between transitions; indeed, when the set of possible transitions in a system have a high degree of interdependence, as is the case with the handlers in typical protocol implementations, partial order methods become less effective. Finally, interesting systems almost always have state spaces with cycles and in such cases Verisoft is limited to checking only up to a fixed depth.

Java PathFinder [13] uses model checking to verify concurrent Java programs for deadlock and assertion failures. It relies on a specialized virtual machine that is tailored to automatically extract the current state of a Java program. Much like CMC, Java PathFinder compresses and stores states in a table to prevent redundant searches and relies on various abstraction techniques to curb the state space explosion problem. The infrastructure on which JPF relies, however, can not be applied to software written in C or C++, which are still the predominant languages used in system software development.

1.2.3 Automatic Model Extraction

Some tools avoid the cost of specifying a high-level description of a system by automatically extracting such a description from the implementation of the system. For instance, Bandera [23] is a sophisticated model extractor for Java programs. It uses a given temporal property as a slicing criteria to extract relevant parts of the system. Also, Bandera accepts user provided annotations to abstract data values to specific subranges. Along similar lines, FeaVer [45] uses a set of user defined mappings to extract abstract models from C code. Both these tools generate descriptions that can be checked using the SPIN [48] model checker.

Model extraction tools still require the user to select and automatically mark stand-alone subparts of the system. While some of this process can be automated [23], the user still needs to have an intimate understanding of the system implementation, making it difficult to scale this approach to large systems. Also, most modeling languages lack many C constructs such as pointers, dynamic allocation, and bit pointers. These omissions further complicate the task of extracting high-level descriptions from C implementations.

1.2.4 Static Analysis

Static analysis has also gained ground in recent years in detecting bugs in software. Tools such as ESC [28], LCLint [35], ESP [26], and the MC Checker [33] have been used to check source code for errors that can be statically detected with minimal manual effort. While static techniques are good for finding a specific set of errors, the CMC approach can find deep conceptual errors in the code such as emergent routing loops that are difficult to find statically. In addition, CMC does not suffer from too many false positives since every scenario checked is a valid execution path.

As an interesting combination, SLAM [2] uses model checking to statically prove invariants about C programs. It converts C code into abstracted skeletons that contain only Boolean types. SLAM then model checks the abstracted program to see if all paths in the program satisfy given invariants. However, SLAM does not deal with concurrent environments that contain multiple processes, queues, etc.

1.2.5 Correct Protocol Synthesis

One way to ensure the correctness of a protocol implementation is to make it *correct by construction*. Many domain-specific languages [10, 16, 57] support the automatic generation of a protocol implementation from a high level description of the protocol. This high level description can independently be verified using conventional formal verification tools. Correctness of the implementation follows from the correctness of both the starting description and the translation process. The drawback of these language-based approaches is that they are domain-specific and are not general enough for implementing different kinds of protocols. Also, the networking community rarely adopts them. To our knowledge, almost all widely-used protocol implementations are written in C or C++.

A different but related method aims to reduce errors by providing a more natural infrastructure for networking implementations. Examples include the x-kernel [50], Scout [65] and, to a degree, the Fox project [7], which uses standard ML and interfaces to help protocol construction. This approach appears easier to deploy than a new language, however these methods mainly help protocol construction, rather than focusing on ways to validate protocol correctness.

1.2.6 Testing Network Protocols

There is a large body of work both in research [43, 49, 4] and in industry [1] that focuses on testing network protocols. While it is not possible to cite all the work done in this area, [58] provides a survey of this field.

Due to the importance of TCP, some interesting testing approaches have been applied specific to this protocol. One approach involves transmitting carefully designed packets to the implementation and observing its response [22, 27]. In contrast, x-sim [12] executes an unmodified TCP implementation in a simulator to test for performance related problems. Typically, testing touches only a small number of executed paths and thus might lead to undetected errors, especially in a complex system such as TCP. Complementary to the testing approaches, tcpanaly [68] *passively* analyzes packet traces to detect abnormal behavior of TCP implementations. While this

relies on large trace sets to achieve coverage of TCP behavior, this approach has a particularly attractive low up-front cost and scales well to large number of instances.

1.3 Contributions of the Thesis

In general, this thesis present a pragmatic approach to model check and find errors in large and complex network protocol implementations. This approach is validated on two real world examples: AODV and TCP.

Specifically, the contributions of this thesis are:

1. The design and implementation of CMC, an explicit state model checker for C and C++ programs.

2. Applying CMC to three implementations of the AODV protocol. CMC found errors in every few hundred lines of code and an error in the protocol specification.

3. Applying CMC to the Linux TCP Implementation. This demonstrates that CMC can scale to such large, complex systems. CMC found 4 errors in this well tested code.

4. Novel state transformation techniques to improve state space search effectiveness. Many of the techniques are previously known but require considerable re-engineering to work in the context of CMC.

5. Design techniques to efficiently save and restore large states, by processing the difference between states.

6. An incremental algorithm for heap canonicalization that does not require a traversal of the entire heap for small changes in the heap structure.

7. General methods for building environmental models for large systems.

8. Metrics to measure partial state spaces and using them to iteratively refine environment models and heuristic searches.

9. Comparing the approaches of model checking and static analysis for finding bugs.

1.4 Overview of the Thesis

Chapter 2 describes the design of CMC and explains the various steps necessary to model check a protocol implementation from a user's perspective. This chapter also describes how CMC captures the state of a given implementation and explores the state space of the implementation.

Chapter 3 explains the various techniques CMC employs to effectively model check a given system. This involves techniques to manage large state sizes and large state spaces of implementations. CMC extracts the incremental differences between states and uses these differences to efficiently save and restore large states. CMC employs a novel heap canonicalization algorithm that does not require a costly traversal of the entire heap. During model checking, CMC dynamically infers various unessential variables and automatically slices them from the state.

Chapter 4 discusses the first case study, the application of CMC to AODV, a routing protocol for ad-hoc networks. This chapter reports the results from checking three publicly available implementations. To further evaluate the effectiveness of CMC, this chapter describes a comparison study between the static analysis and the model checking approaches for finding bugs in the AODV implementations involved in this case study. While CMC is successful in finding many complex errors not found by the static analysis method, CMC misses many errors all of which arise due to imperfections in the environment model.

Chapter 5 addresses the problem of building a comprehensive environment model for large network protocols. This chapter describes a failed attempt to generate a model for the Linux TCP implementation, a much larger and complex protocol than AODV, which clearly demonstrates the pitfalls in conventional approaches of building environment models. Then, the chapter provides some general guidelines for building correct environment models for large systems. Following these guidelines, model checking TCP involves an extreme approach of executing the *entire* Linux kernel in

CMC. This chapter discusses the results from the case study, the use of coverage metrics to evaluate partial state space searches, and using these metrics to iteratively improve the environment model.

Finally, Chapter 6 provides some discussion of future work.

Chapter 2

CMC Design

2.1 Introduction

CMC is a model checker specifically designed for *effectively* finding errors in *large* systems. It checks the system implementation directly and does not require an alternate description of the system. This chapter discusses the design of CMC, the guiding principles, and the various trade-offs required to scale to large, complex systems.

At a high level, a model checker works as follows. A user provides a description of the system and a *specification* that consists of a set of properties that should hold in a correctly functioning system. The model checker then systematically generates the different states of the system, and checks if every generated state satisfies the given specification.

The behavior of a system can be captured by the *state graph* of the system. The nodes in this graph represent the different system states and the directed edges represent the atomic execution steps of the system. A *transition* of the system is an execution of the system from one state to another along an edge in the state graph. The initial state of a transition is known as the *start state* of the transition and the final state is known as the *successor state*. Due to nondeterminism in the system, there can be multiple successor states for a particular state. Each such successor represents one of the many executions possible from that state.

Starting from a system description, model checkers perform a systematic search

for error states in the state graph of the system that is reachable from a given initial state. Model checkers typically generate the state graph *on the fly* by starting from the initial state and recursively generating all its successor states. Doing the search on the fly enables model checkers to report errors even if the state graph is too large to search completely. This is especially important since the state graphs of systems with errors are often much larger than those of correct systems. Obviously, a complete search of the state graph is required before certifying that the system indeed satisfies the specification. However, the size of the state graph grows exponentially with the size of the model, a situation commonly referred to as the "state explosion problem." As a result, a complete search is not always be possible except for very small system descriptions.

Model checking can either be *explicit* or *implicit*. Explicit model checkers generate representations of the individual system states. These model checkers store the generated states in a hash table which is required to avoid exploring the same state more than once. On the other hand, implicit model checkers naturally avoid this redundancy by dealing with symbolic representations of *sets* of states. Such symbolic representations can sometimes result in memory savings by compressing the redundant information shared by many states. However, dealing with symbolic states typically restricts the constructs (such as loops) available when specifying the transitions in the system.

CMC is an explicit model checker and directly uses the system *implementation*; the user is not required to provide any other description of the system. By doing so, CMC avoids the upfront cost typically associated with model checking. The transitions correspond to the actual execution of the code and CMC explores the state space of the implementation by systematically triggering these transitions.

2.1.1 Chapter Overview

The design of CMC is described as follows. Section 2.2 enumerates the key principles that guide the design of CMC. Section 2.3 describes the process of building a system model from an implementation and then, Section 2.4 explains how CMC explores

the state space of the system. Finally, Section 2.5 describes the various correctness properties that CMC checks during model checking.

2.2 Basic Design Principles

This section describes the fundamental principles that underly the design of CMC. As they differ from those employed in conventional model checkers, it is important to state them before discussing CMC design.

2.2.1 Emphasis on Scalability

The primary aim of CMC is to be applicable to large systems. CMC design is heavily optimized for scalability. The fundamental decision of checking the implementation directly is a direct consequence of this need. For large systems, it is extremely difficult to manually specify or automatically extract an alternate model that correctly represents the system. The basic premise behind CMC is that the only "description" possible for a large system is its implementation itself.

2.2.2 Focus on Error Detection

CMC is designed to be a bug-finding tool rather than a checker that certifies correctness. While proving correctness is desirable, the state explosion problem makes it dauntingly difficult to do so even for moderate sized systems. This has led many researchers [20, 17] to insist that model checkers should focus on error detection rather than proving absolute correctness. CMC employs powerful, approximate techniques (§3) to effectively search large state spaces for bugs at the risk of missing some errors in the system.

2.2.3 Minimize False Behaviors

CMC attempts to minimize behaviors in the model that are not possible in the real system. Such false behaviors can cause CMC to waste resources during model checking

and more importantly, can lead to false error reports. In order to minimize such false behaviors, CMC avoids modifying the implementation code as much as possible. This implies that powerful techniques such as abstraction and slicing cannot be applied in CMC.

2.2.4 Minimize Manual Interaction

Another goal of CMC is to minimize the user interaction as much as possible. Requiring human intervention typically makes a tool less usable. More importantly, relying on user input can drastically limit its scalability. User provided guides can be helpful when model checking small systems. As the system size increases, the users' understanding of the system and thus their ability to make good judgments decreases. Thus, user inputs are not of much help when checking large systems, and can sometimes have negative impact as these inputs are more error prone. CMC minimizes user interaction by employing techniques that automatically infer the necessary facts directly from the program execution.

2.3 Building a Model from the Implementation

This section describes the process of building a CMC model of a system from its implementation. Briefly, this process consists of identifying the different execution threads in the system, closing the implementation by providing an appropriate environment, and specifying the correctness properties that need to be checked. This section uses the running example given below to effectively describe this process.

2.3.1 A Running Example

Figure 2.1 shows a skeleton implementation of a routing protocol, very similar to the AODV protocol described in Chapter 4. The system consists of a set of nodes in a network each running the protocol code described in Figure 2.1.

The `main()` function of the implementation calls the initialization function (line 34), and then enters an event dispatch loop (line 35). Depending on the input event, it

calls one of four event handlers defined in lines 1 through 26. Each handler processes an event: a user request for a route to a destination, a request from another node, a response from another node to one of its previous requests, and a timer event requiring the protocol to invalidate its old routes.

2.3.2 Processes and Threads

All systems consist of a collection of interacting entities. In CMC, each such entity is modeled as a *process* executing unmodified C or C++ implementation of that entity. It is possible for two or more processes to execute the same code and thus behave as different instances of the same entity. In order to model multi-threaded entities, each process can have more than one *thread* of execution.

Processes and threads in CMC behave as conventional processes and threads. Each CMC process has its own copy of the global variables and allocates memory from its own heap. Threads can access the global variables and the heap of the process they belong to, but execute in their respective stacks and register contexts. While the processes do not share any state, they can communicate with each other through a shared memory region provided by CMC.

The notions of processes and threads enable a user to easily map the execution of a large system to its model. For instance, when modeling the system in the running example, each routing node can be mapped to a CMC process executing the protocol implementation in Figure 2.1. To emulate the single-threaded execution of the implementation, each process should consist of one CMC thread that appropriately executes the event dispatch loop to handle input events.

CMC along with all the processes comprising the system run as a single operating system process in user space. Internally, each CMC process is implemented as a collection of threads. CMC achieves isolation between the processes by restricting access to appropriate memory regions. The shared memory region is implemented in a straightforward manner by allowing access to all CMC processes. CMC is responsible for scheduling the threads of the processes comprising the system. Different behaviors of the system are explored by appropriately scheduling these threads.

```
1 /* event handlers */
2 on_user_request (dest_ip) {
3   if(route_table has a route for dest_ip)
4     return route for dest_ip
5   else
6     broadcast_request(dest_ip)
7 }
8
9 on_recv_request (dest_ip) {
10   if(route_table has a route for dest_ip)
11     send_response (route for dest_ip)
12   else
13      broadcast_request(dest_ip)
14 }
15
16 on_recv_response (route) {
17   install route in route_table
18   if(route needs to be forwarded)
19     send_response (route)
20 }
21
22 on_timeout(){
23   for each route in route_table
24     if(route too old)
25       remove route from route_table
26 }
27
28 init(){
29   route_table = null
30   insert self route for my_ip
31 }
32
33 main(){
34   init()
35   while(true){
36     event = select(...);
37     depending on event, call one of
38        on_user_request(...)
39        on_recv_request(...)
40        on_recv_response(...)
41        on_timeout(...)
42   }
43 }
44
```

Figure 2.1: A simple routing protocol implementation.

2.3.3 Transitions

A transition is an atomic execution step of the system. In CMC, it corresponds to the execution of a set of instructions by a thread in the model. A transition starts when CMC schedules and transfers control to a thread. The thread explicitly notifies the end of a transition by calling the special CMC function `cmc_yield()`.

Defining what corresponds to a transition is left to the model designer and depends on the system being modeled. As transitions are executed atomically, they determine the concurrency or the degree of interleaving achieved during model checking. The model designer has to decide on a concurrency model suitable for the properties being checked in the system.

Natural execution boundaries exist for most systems, which can be used to define transitions. For instance, many systems are *event-driven* where the implementation consists of a set of handlers for each event the system responds to. The example in Figure 2.1 is one such instance. Such systems can be modeled by mapping each event handler to a transition. The `cmc_yield()` function can be inserted at the end of each event handler (for instance, by inserting it in the `select()` call). Insertions of the `cmc_yield()` function happens at specific program points in the environment or in the appropriate blocking functions in the thread library. Thus, in general, it is not necessary to modify the core protocol implementation.

2.3.4 The Environment Model

Once the execution of the implementation is mapped into processes and threads, and the transition boundaries are appropriately defined, the implementation must then be closed by providing an environment model.

All nontrivial systems interact with external entities such as a human user, a network etc. The environment should contain models for all such entities. As an example, the nodes running the routing protocol in Figure 2.1 interact through a network. A simple model of the network can be implemented in the shared memory region as an unordered queue of bounded length. Any packet sent to the network is queued in this network model. Also, whenever the queue is not empty, a packet is

nondeterministically chosen and sent to the receiving process. To seamlessly communicate with the processes, this network model should implement its own versions of the interface functions, such as the `broadcast_request()` function.

The system to be model checked is typically embedded with other modules in a larger execution context. For instance, most TCP implementations execute in the kernel along with other modules such as the file system. The system being checked can depend on some of these external modules, which need to be adequately modeled in the environment. These can be modeled by providing simple *stubs* or alternate implementations for the interface functions of these modules. For example, `gettimeofday()` might return a constant or the value of an internal counter.

The environment model has to be designed with a lot of care. The decision as to which part of the system is to be checked and which is the environment is decided by the user. Also, the environment should be modeled in as little detail as possible – otherwise, many superfluous states will be generated to model environmental behavior irrelevant to checking the protocol. An important role of the environment is to provide input triggers to the system. If these inputs are not adequately constrained, the environment can trigger executions not possible in a real system, leading to false errors that can be very hard to debug. On the other hand, over-constrained inputs might not trigger adequate behaviors in the system, leading to poor fault coverage during model checking.

Providing a coherent environment model can be time consuming. It is therefore important to reduce the modeling effort required to apply CMC to a previously unchecked protocol. A first obvious step is to engineer the models so that they are as re-usable as possible, reducing the incremental effort of checking a new protocol. This is especially beneficial when related protocols are checked. Chapter 5 addresses the issues related to generating environment models for large systems.

2.3.5 Modeling Nondeterminism

Almost all systems exhibit nondeterministic behavior. Nondeterminism can arise due to many reasons: the different orders in which a system receives input events, the

```
void* malloc(size_t n){
  if(cmc_choose(2) == 0)
     return 0; //nondeterministic failure

  // alloc n bytes from heap and return
}
```

Figure 2.2: Implementation of `malloc()` in CMC. `malloc()` nondeterministically fails to allocate memory.

different values these inputs take, and the order in which different threads in the system are scheduled. Modeling the nondeterminism present in a system is crucial to any model checker.

One important cause of nondeterministic behavior is the concurrency present in the system. The different threads in the system can arbitrarily interleave their execution. CMC captures this nondeterminism by controlling the scheduling of the various threads in the model. CMC explores multiple interleavings by trying different thread schedules from a particular state.

Nondeterminism is also inherently present in the environment. For instance, the inputs generated by a human user of the system are nondeterministic and cannot be predicted beforehand. To represent such nondeterminism in the environment, CMC provides a `cmc_choose()` function (similar to VS_toss in Verisoft[37]). This function takes an integer argument n and returns an integer in the range $[0 \ldots n-1]$. An environment model can use the return value to chose one out of n different choices present at an instant. CMC will attempt to try all possible return values for each call to `cmc_choose()`.

As an example, Figure 2.2 shows the implementation for `malloc()` function that allocates memory from the CMC heap. This function nondeterministically choses between two possibilities: it either allocates the requested memory or simulates a memory overflow by returning NULL. Note that calls to `cmc_choose()` appear in the environment code or in implementations of standard system functions (such as `malloc() and select()`). Thus, it is generally not necessary to modify the actual implementation.

Nondeterminism can also arise when making simplifying assumptions or abstracting away inessential detail in the environment model. For instance, consider a state in which the routing protocol (Figure 2.1) has two pending events: a timeout event and packet reception event. In reality, the order in which these events happen is determined by the complex timing dependencies across multiple events. Modeling the environment to accurately capture these dependencies is neither possible nor necessary. A simple modeling solution is to consider these two events as nondeterministic such that CMC explores both possible execution orders of these events.

A model designer has to be careful when making such approximations in the environment as they can potentially lead to behaviors not possible in a real system. In the example above, if it is never possible for the timeout event to occur before the packet reception, treating them as nondeterministic events can cause CMC to explore invalid behaviors and thus potentially report false errors. However, many system implementations are designed to be robust against such timing dependencies and the use of nondeterminism here can effectively check for these assumptions.

2.3.6 Model Building Summary

To summarize, the process of building a model includes the following steps. First, a model designer identifies the code that needs to be checked and extracts it from other modules in the implementation, if any. Then, the execution of the system is emulated in the model by appropriately creating CMC processes and threads to run the code. The designer also determines the concurrency required in the model by defining the transition boundaries. Finally, a suitable environment model that includes the necessary nondeterminism is provided to close the system. This model along with environment should be able to link with CMC and execute in the context of CMC.

Most network protocols are implemented as non-preemptive systems and CMC is specifically designed to make the modeling of such systems straightforward. In these systems, threads yield control either voluntarily or by blocking on certain events. While modeling such systems, the designer has to merely insert `cmc_yield()` call

on all functions that block or in the special *yield* function supported by the thread library. Once this is done, the model automatically mimics the behavior of the real system.

2.4 Model Checking Algorithm

Once a system model is built as described above, CMC systematically generates the reachable state graph of the system by triggering the various transitions in the system. On generating a state that fails to satisfy the given specification, CMC reports the error and terminates. Otherwise, CMC continues the state space search till it runs out of resources.

2.4.1 Capturing States

Before the state space can be explored, it is necessary to capture the states of the model. At any instant, the model state consists of the states of all the processes in the model along with the shared memory state. The state of a process consists of the state of all its global variables, the contents of its heap and all its thread states. The state of a thread consists of its stack and the register contents.

Given a system, CMC is able to automatically capture its state. Also, CMC is capable of restoring the system to a previously seen state. This enables CMC to backtrack to a state and explore a different execution.

In order to avoid generating false (and nonsensical) behavior, saving and subsequent restoration of states should be invisible to the implementation. Specifically, CMC should ensure that *all* of the state visible to the program is captured. To capture the global variables, CMC saves and restores the entire *data* and *bss* segments of the program implementation. To capture all dynamically allocated objects, CMC traps all memory allocation functions (such as `malloc`, `new`, and `new[]`) in the program and allocates memory from the CMC heap. Capturing the stack and register states is done in a straightforward manner by copying their entire contents.

The system state automatically captured by CMC is sufficient for most systems,

including the ones discussed in this thesis. However, it is possible for programs to maintain some of their state outside their memory context. For instance, they can create files in the disk or access state maintained by the operating system such as lock variables or interrupt masks. When model checking such programs, it is necessary to capture these *external* states also.

Currently, CMC transfers this burden to the user and requires that the environment model adequately accounts for these external states. For instance, the file system model in the environment (implemented in the shared memory) should capture the state of all files that are created, deleted or modified during a transition. CMC then automatically saves and restores the file system state along with the process state, and thus ensures transparency. While this adds extra burden on the user, these models once constructed can be reused for checking other programs. Ideally, such reusable models will be part of an "environment library" that a user can use to build the environment model.

Once the state of the system can be captured, searching the state space is straightforward. A single step of a model checker consists of executing a transition, which amounts to restoring the system to the desired state and scheduling an appropriate thread. The following sections describe this in detail.

2.4.2 Generating the Initial State

CMC computes the initial state of the model from the initial state of its components. Initially, the shared memory is empty. When the process starts the global variables contain values as initialized by the linker and the heap is empty. The stack and register contents of the threads is set up such that the thread starts executing its thread function when scheduled next.

2.4.3 Generating a Successor State

At any given state, every process in the system can have one or more enabled threads. Assuming that a process and one of its enabled threads are selected (as described in the next section), CMC generates a successor state as follows. First, CMC restores

the system to the current state. This involves copying the shared memory contents, the contents of the heap and the global variables of the process, along with the stack and the register states of the thread. After restoring the system state, CMC executes the transition by transferring control to the thread. The thread eventually yields control back to CMC by calling the `cmc_yield()` function. At this point, the contents of the thread state (stack, registers), the process state (global variables, heap) and the shared memory state reflect the changes caused by the execution of the current transition. The successor state is obtained by storing the states of the current thread and process, and the state of the shared memory. The states of other threads in the process, and the states of other processes in the model do not change during this transition.

2.4.4 Generating all Successor States

As described earlier, a state can have several successors due to nondeterminism in the system model. Nondeterminism in a CMC model can arise because of following: the choice of which thread of a process to schedule next, and the choice of the values returned by calls to `cmc_choose()` during this particular thread execution. CMC generates all successors of a state by exploring all possible transitions: it consecutively schedules all enabled threads in that state. For each thread execution, CMC repeats the execution for all possible return values of `cmc_choose()`.

If a thread makes multiple calls to `cmc_choose()`, it is quite possible that CMC repeats the execution exponential number of times. A model designer has to be aware of this crucial fact while designing the model. As an interesting example, Figure 2.3 shows a typical program which calls `malloc()` multiple times. As described in Figure 2.2, each of these calls can nondeterministically fail. However, this does not result in an exponential repetition as the program exits on each `malloc()` failure.

2.4.5 State Space Search Algorithm

Once CMC is able to generate all successors of a given state, the state space exploration simply consists of successively repeating this process starting from the initial

```
void thread_function(){
  char *a, *b, *c;

  // malloc can nondeterministically fail
  // In such a case, it returns 0

  a = (char *) malloc(...);
  if(a == 0){
    return failure;
  }
  b = (char *) malloc(...);
  if(b == 0){
    return failure;
  }
  c = (char *) malloc(...);
  if(c == 0){
    return failure;
  }

  // do interesting work

}
```

Figure 2.3: Multiple calls to `cmc_choose` not necessarily lead to exponential repetition of a transition

state. The pseudocode for the search algorithm is shown in Figure 2.4.

The algorithm is a typical graph search algorithm and maintains two data structures: a hash table of states seen during the search, and a queue of states seen but whose successors are yet to be generated. The hash table guarantees that the algorithm explores the subgraph rooted at a state at most once. The algorithm starts with the initial state in the queue and an empty hash table. During each iteration, it extracts a state from the queue and generates all its successors. Any successor not seen before is reinserted into the queue.

The state graph can be searched depth-first or breadth-first by using appropriate queuing disciplines for the search queue. An interesting possibility is to prioritize the states in the search queue and heuristically select the state that has the maximum likelihood of leading to an error state. Alternately, heuristics can also be used to maximize coverage achieved during model checking.

2.5 Checking Correctness Properties

During model checking, CMC checks for a range of correctness properties, from simple pointer access violation errors to complex protocol bugs.

Currently, CMC can only check for *safety* properties. These properties can be specified as assertions over variables of a *single* state or as assertions that have to be satisfied during the execution of a transition. The current CMC implementation cannot check for liveness properties. This is not as severe a restriction as it might appear at first. Many interesting class of properties can be modeled as safety properties. As safety properties are easier to reason about, many liveness constraints are approximated in practice by checking more conservative safety properties. For instance, to ensure that no TCP node starves in a network, the TCP specification requires that all implementations perform congestion control (i.e. reduce their send window in response to a packet loss), which can be checked as a safety constraint. Finally, for finite systems liveness properties can indeed be checked as safety properties [8].

```
void modelCheck(){
   System State = Process States + shared memory
   Process State = Thread States + global variables + heap
   Thread State = stack + registers

   Queue StateQ;
   Hash VisitedStates;

  Build initial state.
  StateQ.insert(initial);

  while(current = StateQ.pop()) {
    //Repeat forall nondeterministic choices
    forall processes (0 <= pid < N)
    forall threads thid of pid
    forall return values of cmc_choose calls{

      restore shared memory state;
      restore global variables and heap of pid;
      restore stack and registers of thid;

      schedule thread thid;
      //thread yields control by calling cmc_yield;

      // Build successor state
      store stack and register state of thid
      states of all other threads do not change

      store global variables and heap state of pid;
      states of all other processes do not change

      store shared memory state

      if(successor in VisitedStates)
        continue;
      if(successor fails assertions)
        generate error

      VisitedStates.add(successor);
      StateQ.insert(successor);
    }
    if no successor of current state
       generate deadlock error
  }
}
```

Figure 2.4: Pseudocode for the CMC model checking algorithm.

2.5.1 Generic Properties

Given a model, CMC automatically checks for certain generic properties that should be satisfied by all correct C programs. Examples include properties that the program should never crash, that it should never leak memory, etc. As CMC executes the implementation directly, failures to satisfy these generic properties automatically show up as an error during the execution of a transition. For instance, any invalid pointer access results in a segmentation violation that can be detected by CMC.

The memory manager in CMC is designed to catch a variety of memory related errors. When an allocated memory block is released, CMC overwrites the block with a random value. This helps in detecting use-after-free bugs, as such accesses are likely to result in invalid pointer accesses. Similarly, to detect uninitialized accesses in heap allocated memory, CMC initializes a memory block with a random value during allocation.

After each transition, CMC checks for memory leaks. CMC uses the traditional mark-and-sweep algorithm to determine if all the allocated memory blocks in the heap can be reached from the global, stack and register variables of the processes. As the type of the variables cannot be determined in C (or C++), CMC uses techniques from Boehm's conservative collection [9] to infer pointers.

CMC also checks for deadlocks in the given model. Deadlock states are the states from which the system can make no progress. They can be identified in the state graph as those nodes from which the only transitions loop back to themselves. It is possible, however rare, that a transition from a state might lead to a different state, but still result in a self loop due to hash collisions. CMC avoids this problem by checking against the actual representation of the state, rather than its hash, while determining deadlock states.

2.5.2 Model Specific Properties

Apart from the generic properties, CMC can check for model specific properties that are provided by the user. By executing the code, CMC automatically checks for the assertions (such as `assert()`) present in the code. Ideally, these assertions will be

inserted in the code by the implementor well before CMC is applied.

Additionally, the model designer can provide assertions in the form of boolean functions (written in C). CMC evaluates these functions in each state generated and reports an error when any of these assertions fail. As CMC has a global view of all the processes in the model, it is able to check for *system-level* invariants involving states of multiple processes. For example, the model in Figure 2.1 can include an assertion requiring that the routing tables of all the nodes in the network should be loop free at all instants.

In the future, the CMC approach could easily be coupled with other dynamic debugging tools such as Purify[73] or StackGuard[24]. These tools can catch run-time errors such as uses of uninitialized memory, stack overflows, etc. Such tools would be more effective when used with CMC than with ordinary testing, because CMC would achieve greater effective test coverage for a given level of user effort than conventional software testing methods.

Also, CMC provides limited support for checking general safety properties specified as deterministic finite state automata. During model checking, CMC executes the specification automata simultaneously with the implementation providing the same inputs to both of them. CMC reports a violation if the automata reaches an error state. Section 5.4.4.3 provides an example of such an approach.

2.5.3 Producing Error Trace

When any of the properties mentioned above fails during model checking, CMC produces a trace containing a list of transitions that generate the error state from the initial state of the model. To facilitate this, CMC remembers the trace for every state in the search queue. A successor state inherits this trace from the parent state and appends information about the transition that generated it. By doing so, all successor states of a state share the trace information of that state. Also, the states in the hash table do not maintain any trace information. Thus, once the state graph rooted at a particular node is completely searched, the trace information for the node can be deleted.

The error trace produced can later be used to deterministically replay the model execution to reproduce the error. This is useful in debugging the error in the model, and specifically so for complex errors which occur many instructions or transitions away from their actual cause.

The trace produced for an error depends on the specific search discipline used by CMC, and in general will not be the shortest possible trace, unless CMC performs a breadth first search. Also, CMC exits after detecting the first error. It is desirable to run model checkers in batch mode so that a single execution can detect multiple errors. This requires effective pruning of multiple error traces caused by the same error. [39, 3] are interesting approaches to solve this problem.

Chapter 3

Effective State Space Search

3.1 Introduction

CMC exhaustively searches for errors states in the state space of a system implementation. Performing this state space exploration effectively requires a careful management of the limited resources of memory and time.

There are two key challenges in model checking implementations. First, system implementations can have large states. During model checking, CMC needs to save and restore any state visible to the implementation, which necessarily includes all the global variables, the heap and the stack. For practical systems, this can be tens or even hundreds of kilobytes. Second, system implementations can have very large state spaces. When dealing with such large spaces, CMC should ensure that it searches as many system behaviors as possible before running out of resources.

3.1.1 Chapter Overview

This chapter describes how CMC addresses the two challenges mentioned above. Section 3.2 describes how CMC handles large state sizes. The key idea is to process only the *difference* between states. By doing so, states can be effectively compressed. The remaining sections of this chapter, starting from Section 3.3, explain the different techniques CMC employs to manage large state spaces. Section 3.4 describes the

heap canonicalization algorithm that CMC uses to identify equivalent heap states. Section 3.5 explains how CMC automatically slices inessential information from the state. Finally, Section 3.6 describes some search heuristics that guide CMC to interesting parts of the state space.

3.2 Managing Large States

When model checking implementations explicitly, the sheer size of the implementation state can stress both memory and time resources. The following section enumerates the various resources required when model checking systems with large states. Then, Sections 3.2.2 and 3.2.3 discuss two techniques that CMC uses to conserve these resources.

3.2.1 Resource Constraints During Model Checking

For model checkers, memory is more critical a resource than time; typically, model checkers run out of memory before they complete their computation. The memory resources available ultimately determine the amount of state space a model checker is able to explore and thus, determine the number of errors the model checker finds.

Most of the memory resources during model checking is consumed by the two data structures CMC maintains: a hash table and a queue of states (§2.4.5). The hash table contains an entry for every generated state, and thus its size is proportional to both the size of each state and the size of the state space explored by CMC. The queue contains the states that are visited but whose successors are yet to be generated. The size of the queue depends on the particular search discipline used. For instance, in a depth-first traversal of the state space graph the size of the queue is bounded by the depth of the state space, which is typically small for practical state spaces [44]. However, the queue size can get orders of magnitude larger for most other search disciplines including breadth-first search. As a rough estimate, with states as large as hundreds of kilobytes and with practical memory resources available in modern computers, CMC can maintain only a few thousand states in memory at any instant.

This puts the severe constraints on the size of the hash table and the size of the queue during model checking.

While time resources are not as important as memory resources, slow model checkers are not usable in practice. The speed at which a model checker performs the state space exploration is determined by time taken to execute a transition. For each transition, CMC requires at least three traversals of the entire program memory contents, including all the global variables and the entire heap.

1. **Restoring the Start State:** At the beginning of a transition, CMC restores the system to the desired state by copying the contents of the state to the program memory.

2. **Computing the Hash Value:** Once the system is restored to a particular state and the transition is completed, CMC makes a second traversal to compute a hash value for the successor state. The hash value is necessary to determine if the successor is already present in the hash table or not.

3. **Saving the Successor State:** If the successor is not present in the hash table, CMC makes a third traversal of the memory contents to save the successor state in a separate buffer.

Performing these memory traversals on very large states is costly. Accessing large amounts of data results in very poor memory cache performance, a phenomenon commonly known as *data cache poisoning*. Saving and restoring of states involve a direct copy of the state and can be optimized by using the efficient `memcpy()` function available in most operating system environments. However, computing the hash value is more costly; it requires at least an extra memory lookup and a few arithmetic operations for every byte in the state. For states as large as hundred kilobytes, it is possible that these three traversals collectively take as much as tens of milliseconds.

In summary, model checking systems with large states strains three resources: memory resources required for the hash table, memory resources required for the queue, and the time required to make a transition. To scale to systems with large states, CMC should effectively manage these three resource requirements.

3.2.2 Hash Compaction

A technique known as *hash compaction* [75] can be used to reduce the memory resources required for the hash table by several orders of magnitude. Hash compaction is based on the insight that states need not be stored in their entirety in the hash table; compacted state signatures suffice to determine if a particular state is present in the hash table or not.

Hash compaction works as follows. It computes a signature and a hash value for every generate state, each of which is typically around 4 to 8 bytes in size. Hash compaction stores the signature, instead of the state, in the hash table at a location determined by the hash value. As the signature is computed independently from the hash value, it is possible for two states to have the same signature but still be stored at different locations in the hash table as their hash values can be different.

By compacting a large state into a small signature, hash compaction provides several orders of magnitude memory savings. In effect, the memory requirements for the hash table is no longer dependent on the size of each state. With states as large as hundreds of kilobytes, explicit model checking is just not practical without hash compaction.

Using hash compaction involves a small risk of hash collisions, due to which CMC can consider two different states as the same and explore only one of them. However, for state spaces on the order of hundred million states with practical hash table sizes of several hundred megabytes, the probability of missing even a single state due to a signature conflict can be reduced to 0.1% or lower [75]. Also, the risk of collisions can be further reduced by rerunning CMC using different hash functions.

3.2.3 Incremental States

While hash compaction reduces the memory requirements for the hash table, the same approach cannot be used for the states in the queue: all the information in a state is necessary to generate the successors of that state. Moreover, the time resources required to execute a transition still need to be addressed.

This section describes an incremental scheme to represent states, which can conserve both the memory requirements for the queue and significantly reduce the time taken for a transition.

3.2.3.1 The Basic Idea

The incremental scheme is based on a crucial observation that a transition changes only a small fraction of the entire state. The basic idea is to identify the portions of the state that change in a transition and only process these differences when generating a successor state. To manage the state differences effectively, CMC splits the system state into a collection of smaller *state objects* of appropriate granularity. Each system state essentially consists of an *object table* that contains pointers to the state objects that comprise the state. This level of indirection allows different states to share the same state object. CMC tracks the state objects that are modified in a transition and constructs the successor state from the start state of the transition by appropriately updating the pointers to the modified objects. Figure 3.1 shows an example. Starting from a state (obj_1, obj_2, obj_3) that consists of three state objects, a transition modifies only obj_2. The successor state (obj_1, obj'_2, obj_3) can be constructed from the modified object obj'_2, while sharing obj_1 and obj_3 with the start state.

As state objects can be shared between multiple objects, memory requirements for the queue can be reduced, because the queue typically contains states with a lot of commonality, such as all the successors of a particular state. Thus, it is very likely that the states in the queue share a large number of state objects, resulting in significant state savings.

Also, it is no longer necessary to traverse the entire state during a transition. Recall that CMC needs to make three state traversals (§3.2) for every transition: to restore the system to the start state, to compute the hash value of the successor state, and to generate a copy of the successor state. As shown in Figure 3.1, the successor state can be generated by copying the object table from the start state and appropriately updating the pointers to the modified state objects. Specifically, CMC need not traverse unmodified state objects. Also, restoring the system to a desired state requires updating only those state objects that differ between the current system

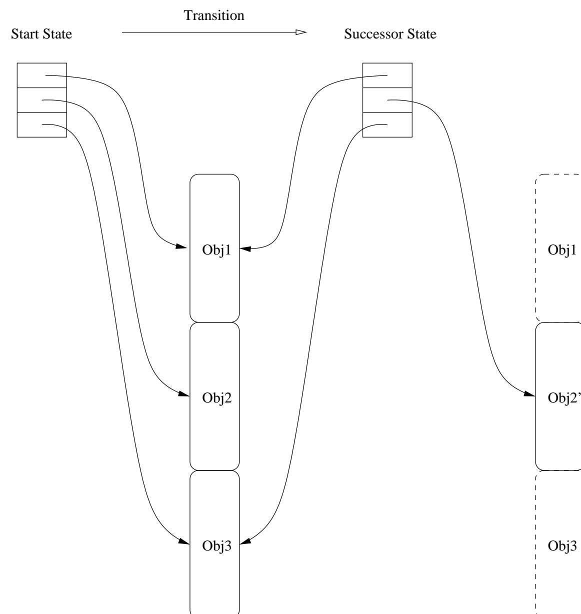

Figure 3.1: The start state and the successor state of a transition can share objects not modified in a transition

state and the desired state. For instance, after executing the transition in Figure 3.1, the system is in the successor state. To restore the system back to the start state, CMC only needs to copy the contents of obj_2 to the system memory. Finally, the traversal to compute the hash value can also be optimized. If the hash function is associative, CMC can compute the hash value for each state object separately and *cache* the value in the object table. If a state object is not modified in a transition, then its cached hash value can be used to compute the hash value for the successor state. Thus, CMC only needs to recompute the hash value for the modified objects. By avoiding altogether the need to access any unchanged state objects in a transition, CMC can achieve a good cache performance. The implementation of the incremental scheme described below speeds up CMC as much as 5 times.

Implementing the incremental scheme in CMC requires addressing two issues. First, to determine which state objects have changed in a transition, CMC needs a mechanism to monitor writes. Second, the performance improvement of the incremental scheme depends crucially on an intelligent choice of the granularity of the

state objects. The following sections discuss these issues.

3.2.3.2 Monitoring Writes

During a transition, CMC needs to determine which of the state objects are modified. In effect, this requires CMC to monitor all the writes performed in a transition. Doing so is non-trivial as CMC executes the implementation code directly and has no control over the execution once the transition is scheduled. One option is to statically instrument either the implementation code [31, 66] or the implementation binary [73, 74] such that CMC gains control on every memory write. Apart from requiring a modification to the implementation, which CMC tries to avoid (§2.2), the challenge is in ensuring that *all* the writes are caught. Inability to do so, can result in inconsistent states that involve partially modified objects. This can totally confuse the implementation or worse, can silently lead to false errors that are hard to debug.

CMC uses a simple mechanism that is easy to implement and is guaranteed to catch all writes performed by a transition. CMC uses the virtual memory protection provided by the underlying operating system to detect writes. Most operating systems allow a user process to mark some of its virtual memory pages as read-only, thereby disabling writes to these pages. When a process attempts to write to a read-only page, the operating system notifies the process using a signal. CMC uses this signal notification to detect any writes performed by the system during a transition.

Virtual memory protection can only be done at the granularity of a virtual memory page, which is typically four kilobytes in modern operating systems. Assuming that the state objects are mapped into virtual memory pages (§3.2.3.3), the incremental scheme works as follows. Each system state consists of a *page table* that contains pointers to the virtual memory pages of the state. The process of generating a successor state is similar to the *copy-on-write* mechanism used by the operating system to fork user-processes. At the beginning of a transition, CMC restores the system to the desired start state and marks all the pages in the state as read-only. During the transition, a first write to some virtual memory page results in a signal. On receiving this signal, CMC makes a copy of the page currently being written to, and grants write accesses to this page. All future writes to this page do not generate a signal.

The copy generated on the first write to a page represents the unmodified contents of the start state of the transition.

Using the virtual memory protection to detect writes comes with additional costs. Modifying the write access permissions to a virtual memory page requires a system call. Also, the handling of the signal generated on the first write to a read-only page is costly and can take tens of microseconds in modern operating systems. These costs are more than compensated by the cost of entire state traversals that are otherwise needed.

3.2.3.3 Managing State Object Granularity

Choosing an appropriate granularity for the state objects is essential. For every state object, CMC needs to maintain a pointer in the object table. This extra overhead is not justified if the state objects are very small. On the other hand, a large state object can lead to *false sharing*; if a state object contains two unrelated variables, CMC unnecessarily saves, restores and computes the hash value for one variable, when only the other variable is modified in a transition. Moreover, the incremental scheme for storing states relies on the crucial fact that only a small percentage of state objects change in a transition. Thus, it is necessary to localize all related variables that are likely to be modified together into as few state objects as possible. Finally, it should be easy to map each state object to a virtual memory page as required by the write monitoring mechanism discussed in Section 3.2.3.2

For ease of implementation, CMC uses simple heuristics to determine the granularity of objects. CMC maps each dynamically created heap object as a separate state object. In most cases, each call to an allocation routine (`malloc()` or `new`) allocates memory for a single structure or an array. Thus each heap object logically constitutes a set of variables that are likely to change together. Also, as heap objects can be allocated at arbitrary memory locations, CMC is free to allocate each such object in a separate virtual memory page.

CMC maps global variables into state objects at link time. Large programs are linked together from different object files. Each object file typically implements a separate module and thus is likely to contain related global variables. Accordingly,

CMC allocates the variables from an object file into one or more virtual memory pages at link time. The stack and the register contents are small enough and can be allocated in a separate state object.

The scheme of incremental differences discussed above can be viewed as essentially the same as that used by distributed shared memory systems (such as Treadmarks [55]) to reduce the cost of remote page migration. Many optimizations proposed (such as [52, 53]) in the context of distributed shared memory are also applicable in CMC. Using these techniques to improve the performance of the incremental scheme for storing states is an interesting future work.

3.3 Managing Large State Spaces

The state explosion problem is the most serious problem with model checking in practice. The state space of a system grows exponentially with the system description and can be very large, or even infinite for practical systems. Model checkers apply various state space reduction techniques such as symmetry reduction [21, 30, 19] and partial-order reduction [38, 81] to address the state explosion problem. These techniques seek to identify a set of *equivalent* states such that the model checker needs to explore at most one of them. By doing so, the hope is that the resulting reduced state space is small enough for the model checker to completely explore and thus guarantee the correctness of the system.

There are several problems in directly applying these techniques to CMC. First and foremost, as CMC checks the implementation directly, it might not be possible to completely search the state space *even* after applying these reduction techniques. Also, most of the reduction techniques require that the input system be described in a suitable formalism. For instance, using symmetry reduction [21] requires that the user identify some variables as belonging to a special data type *scalarsets*. Such formalisms are not readily applicable for checking arbitrary C programs.

The main goal of CMC is to be an effective bug-finding tool, and not to prove correctness. This allows CMC to use approximate, yet powerful reduction techniques.

These techniques allow CMC to identify a set of *similar*, but not necessarily equivalent, states and to avoid redundantly exploring such similar states more than once. This enables CMC to explore as many system behaviors as possible before running out of resources, thereby increasing the chances of finding errors in the system.

Obviously, the use of approximate techniques comes with a risk of missing errors in the system. However, this is a meaningful trade-off for a practical tool like CMC. As will be evident in the forthcoming sections, the freedom to use approximate techniques allows CMC to scale to large systems, be applicable to arbitrary C programs, and to avoid user annotations by automatically inferring the required information.

Sections 3.4 to 3.6 describes the various reduction techniques that CMC employs. Section 3.4 describes a heap canonicalization algorithm that enables CMC to identify two heap configurations that are identical but differ in the placement of objects in the heap. The main challenge is in performing this canonicalization incrementally. Section 3.5 describes a technique to automatically slice inessential variables from the state. Finally, Section 3.6 explains some heuristic search techniques to maximize the chances of finding errors.

While these techniques alleviate the state explosion problem, they only help if the system model provided by the user is carefully designed in the first place.

3.3.1 A Good Model Design

A large responsibility of minimizing the state explosion problem is on the user of CMC. A well designed system model adequately expresses the desired behaviors of the system without causing unnecessary state space blowup. Unfortunately, building such a good system model remains an art, and requires a lot of trial and error. The following discussion provides some general guidelines.

3.3.1.1 Down-scaling

A very useful and necessary technique is to limit the scale of the system [29]. For instance, when model checking a network protocol, one can restrict the number of nodes in the network to, say, three or four. Usually, there is a tendency to make

the system more general by increasing its scale. While this might be acceptable in a testing or a simulation framework, increasing the scale of a system can adversely affect the effectiveness of model checkers; the state space is typically exponential on most scale parameters of the system. Model checkers are good in exploring *complex* interactions in a system of *small* scale. Most hard-to-find bugs involve such complex interactions among a small number of processes, and are therefore preserved after down-scaling. Of course, this may miss bugs that only occur for larger instances of the system.

3.3.1.2 Avoiding Model Redundancy

When designing a system model, it is necessary to carefully avoid redundancy in the system. As a simple example, corrupting the same packet twice can increase the state space without exploring new protocol behaviors. Sometimes, two seemingly different transitions can produce the same effect on the protocol behavior. For instance, it is not necessary to model a link failure in the network, as it is equivalently modeled as a loss of all packets sent on that link.

3.3.1.3 Avoiding Extraneous Information from the Environment

A model designer should carefully avoid any extraneous information in the system state. In theory and sometimes in practice, a single bit of unnecessary information can double the state space. As CMC requires unmodified implementations, the user has no control over any inessential information already present in the implementation. CMC has mechanisms to slice such information from the state (§3.5). However, the user can aid CMC in ensuring that no extraneous information enters the system from the environment.

During model checking, the implementation makes calls to various functions (such as `malloc()`, `select()`) in the environment. When such functions return, the environment should ensure that the return value is functionally dependent on the input arguments provided by the system. There are many *hidden* means by which information from the environment can corrupt the system state. For instance, a local variable

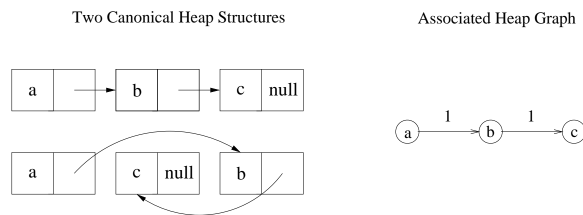

Figure 3.2: Two heap states that are behaviorally equivalent, and their common heap graph

in an environment function can potentially corrupt the stack of the calling thread. All such variables have to be cleared before returning from the function. Similarly, the function should clear any registers that are not caller-saved.

3.4 Heap Canonicalization

Most programs dynamically allocate objects in the heap. The exact memory location of these objects is decided at run time by some internal heap allocation algorithm and typically depends on the ordering of all previous object allocations and deletions. From the perspective of a program, however, the object locations and the contents of pointers to these objects is completely arbitrary.

Correctly functioning programs should not make any assumptions about the object locations and are required to behave the same irrespective of these locations. Specifically, the program should not make any *unsafe* pointer manipulations such as comparing the pointers of two different heap objects or creating pointers from arbitrary non-pointer values. To enforce this discipline, many type safe languages such as Java do not allow programs to access or manipulate pointer values.

Allocation of objects at arbitrary locations in the heap poses an important problem for explicit state model checkers. While exploring the state space, a model checker can produce two heap states that are identical except for the locations of the objects. For example, Figure 3.2 shows two heap states that contain the same linked list of three elements but differ in the location of the different objects. Assuming that the program being model checked does not make any unsafe pointer manipulations, such

heap states are *behaviorally equivalent*, as the program cannot differentiate between them. A model checker should identify such behaviorally equivalent states and explore at most one of them.

The behavioral equivalence of heap states can be captured by their *heap graph*, which is discussed in the following section. A heap canonicalization algorithm extracts the heap graph of a heap state and produces a canonical representation from the heap graph. Using such an algorithm, a model checker explores only one of all the heap states that share the same heap graph. The notion of the heap graph and the first heap canonicalization algorithm are due to Iosif [51]. The main contribution of this thesis is an improved heap canonicalization algorithm that scales to large heaps by avoiding extensive memory traversals.

3.4.1 The Heap Graph

A heap graph *abstracts* the memory locations of the heap objects, while maintaining information about the pointers to these objects. This graph contains a node for each object in the heap. Whenever there is no confusion, the node corresponding to a heap object will simply be referred to as the *object* in the heap graph. There is a directed edge from object a to object b in the heap graph if and only if a contains a pointer to b. This edge is labeled by the offset of the field in object a that contains the pointer to b. For instance, the heap graph of the linked list is shown in Figure 3.2. Obviously, no two outgoing edges of an object have the same label. Each node in the heap graph is labeled by the values of all the non-pointer fields in the corresponding object as shown in Figure 3.2.

A program can access heap objects only through pointers from global variables or local variables in the stack. It is necessary to capture the state of these variables in the heap graph as well. Conceptually, all global and local variables can be considered as fields in a *global* object that represents the statically allocated region in the memory. The heap graph contains a global node that corresponds to the global object. If a global or a local variable contains a pointer to a heap object, the heap graph contains an edge from the global node to the object. This edge is labeled by the offset of the

variable in the global object. Like other nodes in the heap graph, the global node is labeled by the values of all non-pointer global and local variables.

Basically, a heap graph captures the entire state of the heap along with all the global and local variables, but abstracts the memory addresses contained in the pointer variables. Provided a program does not make any unsafe pointer manipulations, it cannot differentiate between two heap states that have the same heap graph.

A *heap canonicalization* algorithm is an algorithm that generates a single canonical representation for all heap states that have the same heap graph. A model checker can use such an algorithm to generate a canonical representation of the heap state and store this representation in the hash table. This ensures that the model checker does not redundantly explore two heap states that are behaviorally equivalent.

3.4.2 Previous Approaches

The heap canonicalization algorithm is necessary for any software model checker that models the state of the heap. There are two algorithms proposed in the model checking literature.

3.4.2.1 Lerda and Visser's Algorithm

The need for canonicalizing the heap is (to the author's knowledge) first mentioned by Lerda and Visser [59] in the context of Java PathFinder [13], a model checker for multi-threaded Java programs. The algorithm tries to avoid the heap canonicalization problem altogether by attempting to allocate an object at the same location in the heap. The basic idea is to create a mapping between an object and the memory location when the object is *first* allocated. This mapping is then preserved while backtracking, and is used to determine the location of objects when allocated in a different path during model checking.

The algorithm in [59] assumes that two objects allocated at the same *dynamic context* are the same. This context consists of the Java bytecode instruction that allocates the object, along with a counter that represents the number of times the

instruction has previously executed. The counter is necessary as the same instruction can execute multiple times (say, in a loop). The algorithm attempts to assign the same memory location for two objects allocated in the same dynamic context. [59] reports substantial state reductions in specific examples using this scheme.

However, this algorithm is a heuristic at best and cannot guarantee heap canonicalization. The main problem is that the dynamic context in which an object is allocated does not correlate with the behavioral equivalence as determined by the heap graph. Looking at Figure 3.2, the list element with the label c should be identified as the "third element in the linked list" irrespective of when the element is allocated. Suppose this linked list represents an *ordered* queue of received packets, then it is quite possible for the system to receive the same packet at different contexts (say, at different protocol states) but still insert the packet at the same location in the linked list. Also, when packets can get reordered, then the system can receive different packets in the same context but insert them at different locations in the linked list.

In summary, the behavior of the heap is determined by the heap graph and not by the dynamic context in which objects are allocated. Thus, a heap canonicalization algorithm should determine the canonical representation from the heap graph. The Lerda and Visser's algorithm can work for simple cases, but does not extent to large, complex programs.

3.4.2.2 Iosif's Algorithm

A comprehensive solution to the heap canonicalization problem is provided by Iosif in [51]. The notion of the heap graph and the behavioral equivalence of two heaps with the same heap graph are due to Iosif. The Iosif's algorithm is guaranteed to produce the same canonical representation for behaviorally equivalent heap states.

At first sight, the heap canonicalization problem appears to be an instance of the graph isomorphism problem, which is intractable [36]. However, the simple but crucial insight is that the outgoing edges of a node in the heap graph can be *deterministically* ordered by the labels on these edges. As mentioned in Section 3.4.1, these labels represent the offsets of various pointer fields in an object and can simply be ordered

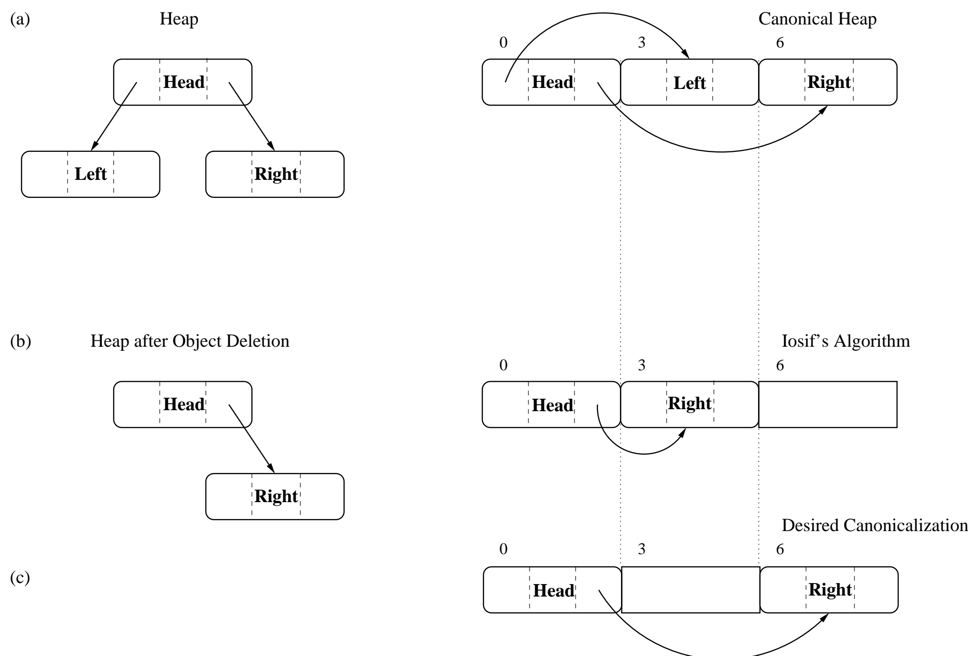

Figure 3.3: Iosif's heap canonicalization algorithm.

by their values.

The Iosif's algorithm essentially involves a deterministic traversal of the heap graph. ([51] specifically uses a depth first traversal of the heap graph.) During the traversal, the algorithm relocates the heap objects in a separate *canonical heap*, which at the end of the traversal contains the canonical representation of the original heap. The algorithm starts the traversal from the global node of the heap graph and an empty canonical heap. For every node visited, the algorithm relocates the corresponding heap object in the next available memory location in the canonical heap. After relocating an object, the algorithm modifies the values of all pointers to that object to reflect the change in the location. Figure 3.3(a) shows an example. The heap consists of three objects in a binary tree. The *head* node is global and contains pointers to a *left* node and a *right* node. Assuming that the size of these nodes is 3 word lengths, the algorithm relocates the head, left and right nodes at offsets 0, 3 and 6 respectively in the canonical heap.

The correctness of the Iosif's algorithm follows. If two heap states have the same

heap graph, then the algorithm will visit the heap objects in the same order and relocate them at the same offsets in the canonical heap. This produces the same canonical representation.

3.4.3 Incremental Heap Canonicalization

As discussed in Section 3.2.3, CMC processes only the incremental differences between the start and the successor states of a transition. One reason to do so is to avoid the costly traversal of the entire state. However, Iosif's algorithm might require traversing huge portions of the heap even for a small change in the heap.

To illustrate this, consider the heap in Figure 3.3(a) and a transition that deletes the left node in the binary tree. Figure 3.3(b) shows the resulting heap and the canonical representation generated by the Iosif's algorithm. The algorithm locates the right node at offset 3 in the canonical heap, while the node was previously located at offset 6. This change in the location invalidates the precomputed hash value for the right node, forcing CMC to traverse the node to compute the hash value even though the node did not change in the current transition.

In general, as the Iosif's algorithm relocates objects in the graph traversal order, an incremental change in the heap graph can change the locations of a large number of heap objects. Specifically, a deletion or an insertion of an object can change the locations of all objects that follow this particular object in the traversal order. The hash value needs to be recomputed for any object whose location is modified. Also, when the location of an object changes, the contents of all the pointers to the object change. Thus, any object that contains such a pointer needs to be updated and its hash value recomputed. In the worst case, the algorithm might require recomputing the hash value for all objects in the heap.

This section describes an incremental heap canonicalization algorithm implemented in CMC. The basic idea is to minimize changes in the object location so that their precomputed hash values can be used. For instance, Figure 3.3(c) shows the desired canonical representation of the heap in Figure 3.3(b). Specifically, the location of the right node in the canonical heap does not change when the left node is deleted from

the binary tree.

Like the Iosif's algorithm, the incremental algorithm relocates the heap objects into a canonical heap to generate a canonical representation. However, the location of an object in the canonical heap is independent of the graph traversal order. Instead, the location of an object is determined by a global mapping that the algorithm maintains. In this respect, the incremental algorithm is similar to the Lerda and Visser's algorithm. But, the crucial difference is in the way the objects are *identified* by the algorithm. Instead of using the dynamic context of object allocations, the incremental algorithm uses an object-specific property in the heap graph to identify a heap object. Specifically, the algorithm uses the *bfs access chain* of an object, which corresponds to the shortest path between the object and the global node in the heap graph. As will be explained below, this guarantees that the algorithm produces the same canonical representation for behaviorally equivalent heaps. The use of the global mapping ensures that the location of an object in the canonical heap does not change unless the bfs access chain of the object changes in a transition.

A detailed explanation of the algorithm follows. The next section defines the notions of access chains of an object, which are crucial to the incremental algorithm.

3.4.3.1 Access Chains

Heap objects can only be accessed through pointers from global or local variables. For instance, a C program can access the right node in Figure 3.3(a) with an expression `head->right`. In general, a heap object *obj* can be accessed through a chain of pointers. This access can be represented by $\langle obj_0, \mathit{offset}_0, obj_1, \mathit{offset}_1, \ldots, obj_n = obj \rangle$, where obj_0 is the global object, and for $0 \leq i < n$, offset_i is the offset of a field in obj_i that contains a pointer to obj_{i+1}. In the heap graph, this chain forms a path starting from the global node to the node corresponding to *obj*. This path is uniquely defined by the offset labels on the path edges.

Formally, an *access chain* of a heap object is a path in the heap graph from the global node to the object and is denoted by $\langle \mathit{offset}_0, \mathit{offset}_1, \ldots, \mathit{offset}_{n-1} \rangle$, the list of offset labels on the path edges. For instance, the access chain of the third element in the linked list of Figure 3.2 is $\langle 1, 1 \rangle$. Also, the global node in the heap graph has an

empty access chain $\langle\rangle$.

In general, there can be multiple access chains for the same object, corresponding to the different ways a program can access that object. This is typically the case for objects that are present in multiple data structures. The set of access chains of an object uniquely identifies the object in the heap. Also, if two objects in two different heaps have the same set of access chains, then the two objects are *equivalent*, since no program can differentiate between these two objects.

3.4.3.2 BFS Access Chain

A breadth first traversal of the heap graph naturally defines an access chain for objects in the graph. During a breadth first traversal, the edges used to traverse the graph form a spanning tree of the graph rooted at the global node. For any object in the graph, this spanning tree provides a shortest path from the global node to that object. Obviously, the access chain that corresponds to this path is a shortest of all access chains for the object. Additionally, if the breadth first traversal traverses the edges from an object in the increasing order of their offset labels, then the access chain constructed above is guaranteed to be the lexicographically smallest of all shortest access chains of the object.

Formally, a *bfs access chain* of an object is the access chain $\langle a_0, a_1, \ldots, a_n\rangle$ such that for any other access chain $\langle b_0, b_1, \ldots, b_m\rangle$ of the same object the following holds: either $m > n$; or $m = n$ and there is an $i < n$ such that for all $0 \leq j < i$, $a_j = b_j$ and $a_i < b_i$. The bfs access chain of all objects in the heap graph can be constructed by performing a breadth first traversal of the graph. Given a heap graph, the *bfs access chain* of an object is unique. The incremental heap canonicalization algorithm uses this chain to determine the location of objects in the canonical heap.

3.4.3.3 Relocating Objects in the Canonical Heap

A heap object can be characterized by its start address and its length. The term *obj* will be used synonymously to refer to both a heap object and its start address. Also, the term $len(obj)$ represents the length of the object *obj*. The incremental heap

canonicalization algorithm involves a relocation of the heap objects in a canonical heap. Let $reloc(obj)$ represent the location to which a heap object obj is relocated in the canonical heap. The ultimate aim of the canonicalization algorithm is to define the $reloc$ function such that the relocation guarantees the same canonical representation for two heaps with the same heap graph.

The $reloc$ function should have some desirable properties. First, only heap objects should be relocated and it is not necessary to relocate the global object obj_0. Thus,

$$reloc(obj_0) = obj_0 \tag{3.1}$$

Also, the relocation function should relocate objects in their entirety. In other words, the offset of fields in an object should be invariant during relocation.

$$reloc(p) = reloc(obj) + p - obj, \quad \forall p : obj \leq p < obj + len(obj) \tag{3.2}$$

Finally, the $reloc$ function should not overlap objects in the canonical heap. That is, for all valid heap addresses p, q

$$reloc(p) = reloc(q) \iff p = q \tag{3.3}$$

For a heap object obj, the incremental algorithm seeks to define the $reloc(obj)$ from the bfs access chain of the object. Let $\langle a_0, a_1, \ldots, a_n \rangle$ be the bfs access chain of a heap object obj. Let $parent(obj)$ be the parent of the object obj in the breadth first traversal. Obviously, the bfs access chain of $parent(obj)$ is $\langle a_0, a_1, \ldots, a_{n-1} \rangle$. Assume there exists a special function $canon()$ that maps memory addresses to memory addresses. Consider the following relocation.

$$reloc(obj) = canon(reloc(parent(obj) + a_n)) \tag{3.4}$$

Note that a_n is the offset of a field in $parent(obj)$ that points to obj, and thus $parent(obj) + a_n$ represents the address of that field. The following simple theorem follows.

Theorem 1 *For an arbitrary function canon(), the relocation function defined by Equation 3.4 along with Equations 3.1 and 3.2 relocates two objects with the same bfs access chain to the same location in the canonical heap.*

Proof: The proof follows from a simple induction on the depth of the bfs access chain of an object. The base step trivially follows from Equation 3.1. Assume the theorem holds for all objects with a bfs access chain of length $\leq n$. Consider two objects obj_1, obj_2 in two different heaps that have the same bfs access chain $\langle a_0, a_1, \ldots, a_n \rangle$. Obviously, $parent(obj_1)$ and $parent(obj_2)$ have the same bfs access chain $\langle a_0, a_1, \ldots, a_{n-1} \rangle$, which is of length n. Thus, by induction $reloc(parent(obj_1)) = reloc(parent(obj_2))$. This implies that $reloc(obj_1) = reloc(obj_2)$ from Equation 3.2. □

Theorem 1 allows the following interpretation for Equation 3.4. Reasoning inductively, if the term $reloc(parent(obj))$ captures the bfs access chain of the $parent(obj)$ $\langle a_0, a_1, \ldots, a_{n-1} \rangle$, then the term $reloc(parent(obj)) + a_n$ captures the bfs access chain of obj $\langle a_0, a_1, \ldots, a_n \rangle$. Thus, Equation 3.4 is just specifying that the $reloc(obj)$ is a function of the bfs access chain of the obj. The function *canon* simply provides this mapping.

Now, with the additional constraint of Equation 3.3, the following theorem shows that the relocation defined above provides a canonicalization of the heap.

Theorem 2 *For an arbitrary function canon(), the relocation function defined by Equation 3.4 along with Equations 3.1 and 3.2 generates the same canonical representation for two heaps with the same heap graph, provided Equation 3.3 holds.*

Proof: Given two heaps with the same heap graph, equivalent objects in the two heaps have the same bfs access chain. The proof follows from Theorem 1. The constraint in Equation 3.3 can be satisfied by appropriately designing the *canon* function as shown in Section 3.4.4. □

The advantage of this algorithm over the Iosif's algorithm comes from the fact that the relocation address depends *only* on the bfs access chain of the object. So, any changes to the heap graph such as object allocations, deletions or pointer assignments that do not change the bfs access chain of an object do not affect the relocation address of the object. This has a huge impact when canonicalizing the heap of nontrivial

```
//globals
canon_table; //implemented as a hash table
unalloc_address = start_heap_address;

canon(parent_ptr, len){
  if(canon_table[parent_ptr] is not defined){
    // incrementally define canon for (parent_ptr)
    canon_table[parent_ptr] = unalloc_address;
    unalloc_address += len;
  }
  return canon_table[parent_ptr];
}

bfs_visit(obj){
  for(each field f of obj){
    a = offset of field f
    if(f is not a valid pointer)
      continue;

    child_obj = object pointed by f;
    if(child_obj seen before)
      continue; // child_obj is already canonicalized

    // obj is the parent of child_obj
    reloc[child_obj] = canon(reloc[obj + a], len(child_obj));
    queue child_obj for bfs visit
  }
}
```

Figure 3.4: The incremental heap canonicalization algorithm

programs which typically consist of large collection of data structures that are only loosely interconnected. Modifications to some of the data structures do not affect the canonicalization of other data structures.

3.4.4 Implementation Details

The only remaining constraint is to define a *canon* function such that the relocation defined in Equation 3.4 follows Equation 3.3. In fact, it is fairly straightforward to do so. The trick is to define *canon* incrementally in such a way to avoid object overlaps. Figure 3.4 describes one such implementation. The *canon* function is implemented as a lookup table that is incrementally filled.

The incremental algorithm can be efficiently implemented as shown in Figure 3.4. To avoid the breadth first traversal of the entire heap, the algorithm can maintain a compact representation of the heap graph along with the heap. Typically, the compact representation is many orders of magnitude smaller than the actual heap. After a transition, the algorithm traverses this representation to determine the objects that are modified in this transition and the objects for which their bfs access chain has changed. The hash value needs to be computed only for these objects.

3.4.5 Some Extensions

While the algorithm described above, ensures that an $reloc(obj)$ does not change unless its bfs access chain changes, some optimizations are possible to handle the common cases when the bfs access chain *does* change.

During the program execution, an object might get an extra reference. If this new reference precedes the previous parent of the object, the bfs access chain of the object will change, though the object itself remains unchanged. Another common occurrence is for heap objects to be transferred from one data structure to another. For instance, when the program starts a connection object can be in the list of unallocated connections and subsequently, the object might be transferred to a connected list after connection establishment. This modifies the bfs access chain of the object.

Surprisingly, a simple modification to the algorithm shown in Figure 3.5 can handle all such cases. The key idea is to reuse the old relocation address of objects whenever possible. When traversing the heap, objects that are allocated in the current transition are processed as before. All other objects were present before the transition and thus have an old relocation address. Consider one such object *obj* which has an bfs access chain *bac* and a canonical heap location *loc* before the transition. Obviously, the function *canon* is defined such that $loc = canon(bac)$. Suppose a transition changes bfs access chain of *obj* to bac' and $canon(bac')$ is not defined, then the improved algorithm defines $canon(bac')$ to be *loc*. By doing so, the algorithm identifies that the two bfs access chains *bac* and bac' correspond to the same object and thus modifies the *canon* function to return the same relocation address. This handles the most common cases of bfs access chain changes.

Having the canonicalization function to return the same address for two or more bfs access chains comes with a risk; if the program subsequently creates a heap with two *different* objects at these bfs access chains then the canonicalization function would try to relocate these objects to the *same* address. Such an overlap invalidates Equation 3.3. If the algorithm detects such a conflict, then it mutates the *canon* function to assign different relocation addresses. Whenever the *canon* function mutates, Theorems 1 and 2 no longer hold and thus unique signatures for canonical heaps are no longer guaranteed. Mutation of the *canon* function can cause CMC to redundantly explore two equivalent heap states. However in practice, such mutations occur only during the initial phases of model checking. After traversing the first few hundred states, the algorithm progressively *learns* about the bfs access chains that can potentially point to different objects.

3.4.6 Extracting the Heap Graph

CMC does not have type information of C objects at runtime. Thus, CMC has to *infer* pointers to objects while performing the heap traversal. As in [9], CMC treats any bit pattern that looks like a pointer as one. While this works in most cases, it is quite possible, however rare, for CMC to make wrong guesses and follow invalid

```
//globals
canon_table; //implemented as a hash table
unalloc_address = start_heap_address;

canon(parent, len, old_value){
  if(canon_table[parent, len] is not defined){
    if(old_value is valid){
      // try using the old_value
      canon_table[parent, len] = old_value;
    }
    else{
      //relocate to new memory address
      canon_table[parent, len] := unalloc_address;
      unalloc_address += len;
    }
  }
  if(canon_table[parent, len] conflicts){
    mutate the canon function by
    modifying the canon_table to avoid the conflict
  }

  return canon_table[parent_ptr, len];
}

bfs_visit(obj){
  for(each field f of obj){
    a = offset of field f
    if(f is not a valid pointer)
      continue;

    child_obj = object pointed by f;
    if(child_obj seen before)
      continue; // child_obj is already canonicalized

    // obj is the parent of child_obj
    reloc[child_obj] = canon(reloc[obj + a], len(child_obj),
                             old_value(child_obj));

    queue child_obj for bfs visit
  }
}
```

Figure 3.5: The improved heap canonicalization algorithm

pointers. This can lead to both kinds of errors: CMC can map two different states to the same canonicalized representation, or can fail to map two equivalent states to the same representation. The former kind of error can cause CMC to miss errors as it fails to explore some behaviors, while the latter causes blowup of the state space. In order to minimize such errors, CMC allocates the heap at random memory locations in each run.

3.5 Automatic Data Slicing

One problem with model checking entire implementations is that the implementation might contain lot more detail than required for the properties to be checked. For a simple example, a network protocol implementation might gather statistics regarding its protocol processing. While the statistics do not alter the functioning of the protocol, they unnecessarily blowup the state space.

Conventional model checkers require that the system description provided by the user should not contain any such extraneous detail. However, manually identifying such inessential state is extremely difficult, especially for large, complex systems. Several approaches in the past [46, 23, 13] use static analysis to automatically slice detail not essential to the properties checked. However, generating an effective slice that contains all the interesting behaviors of the system still requires a lot of human intervention. Moreover, static analysis is conservative in practice. Such conservative slices remove only few (if any) redundant information from the program.

CMC performs a simple dynamic analysis to infer irrelevant variables at runtime. For each generated state, CMC tracks values of all state variables. These variables are identified by their specific word offsets in the state. From the different values a variable takes, CMC automatically infers a variety of inessential variables such as counters, statistics variables and time variables. CMC removes such variables from the state once detected. Optionally, the variables detected in one run of the model checker can be reused in subsequent runs.

Additionally, the user can provide bounds on the different values a variable can take. CMC automatically prunes a path in which some state variable has taken more

than a threshold of different values. For example, if a network protocol retransmits a packet and the packet is subsequently dropped by the network, the protocol enters a state that is similar to the state before the retransmission, except for any retransmission counters maintained by the protocol. It is possible for CMC to repeatedly execute a retransmission followed by a loss of the retransmitted packet till the protocol gives up.[1] However, by setting a bound on the values of the state variables, such retransmissions can be limited.

3.6 Heuristic Search

Since CMC cannot exhaustively explore the state space of real protocols it tries to explore the most interesting portions of them. It does so by attempting to focus on states that are the most different from previously explored states. The intuition for this is that the more different a state is from previous states the more likely it is to have new behaviors and, as a result, bugs.

This section describes some of the heuristics used in CMC. The ideas behind many of these heuristics are due to David Park. The implementation of all the heuristics mentioned below and their effective evaluation is a joint effort by David Park and the author of this thesis.

The first class of heuristics involves dropping states altogether if they are deemed uninteresting. A simple heuristic is to limit the number of times particular transitions occur along a given model checking path. The second class of heuristics involves exploring more interesting states first using best-first search. CMC contains a module to monitor state variables to keep a history of which state bits have changed during checking. The basic idea is that if the number of bit positions that have changed since the initial state suddenly increases or if variables take on less frequented values, the state is considered more interesting and explored earlier. This heuristic tends to bias the search toward cases where outliers occur or where states seem to diverge from the norm. This idea is adapted from DIDUCE [42], a tool that flags such divergent cases

[1] The threshold for these retransmission counters are in practice much smaller than the threshold used by CMC to mark them as irrelevant variables

and reports them to the user during program testing.

Finally, a user can guide CMC by providing a routine to extract an *abstract* state for a given system state. By doing so, the user identifies the variables that are crucial to the behavior of the system. For instance, when model checking a network protocol implementation, the user can identify the *protocol control block* that contains the most essential protocol state variables. A routine to extract the abstract state is typically very easy to implement, and is typically many orders of magnitude easier than providing an abstract description of the entire system. CMC then uses the extraction routine to guide the state space exploration to states that have different abstract states.

Preliminary results indicate that all the errors discovered with the use of the heuristics could also be discovered with simple depth-first search or random search. But the use of heuristics often accelerated the discovery of errors and produced shorter examples of executions leading to a given error. However, much more experimentation with various heuristics is needed on a wider range of protocols to arrive at reliable conclusions.

Chapter 4

The AODV Case Study

4.1 Introduction

The first case study involves the Ad-hoc On-demand Distance Vector (AODV) protocol, a routing protocol for ad-hoc networks. This protocol is designed to work in an environment of mobile nodes, withstanding a variety of network behaviors such as node mobility, link failures and packet losses. AODV is a loop-free routing protocol and thus should guarantee that the network is free of routing loops at all instants.

The choice of using AODV as the first case study was determined by a few key factors. First, AODV is a self starting protocol making it fairly straightforward to model the environment. The only input it needs from the user is a request for a route to a destination. This input generates enough behavior in a network of nodes to cover all the integral parts of the protocol. Apart from this input, the protocol requires a timeout and a link failure notification from the network to trigger error recovery mechanisms. These additional inputs are straightforward to model in the environment.

Second, AODV has a well defined property that CMC can check: the routing tables should be loop-free at all instants. Any behavior that leads to a routing loop is definitely an error, making it easy to identify erroneous behavior during model checking. Also, as AODV has no mechanism to recover from routing loops, any routing loop erroneously introduced in the network persists in the network forever.

This makes it essential to thoroughly check both the protocol specification and any protocol implementation for loop-freedom.

The ease of modeling the environment and specifying the correctness properties enables this case study to concentrate on the general methodologies of CMC and to evaluate the effectiveness of model checking implementations. As will be demonstrated in Chapter 5, the two tasks of environment modeling and specifying correctness are not straightforward for a complex protocol like TCP, and will occupy most of the discussion in that chapter.

This case study involves three publicly available implementations of AODV. These implementations are of moderate complexity and on an average, consist of 6000 lines of code. In these three implementations, CMC found a total of 42 errors of which 35 are unique errors. These errors range from simple memory errors to deep errors that require a complex sequence of events to trigger. Also, by checking the implementations, CMC found an error in the AODV protocol *specification*.

4.1.1 Chapter Overview

This chapter begins with a brief description of the AODV protocol in Section 4.2. Section 4.3 describes the three AODV implementations used in this case study and Section 4.4 discusses the process of building a CMC model for these implementations. Section 4.5 mentions the various AODV-specific state space reduction techniques required and Section 4.6 discusses the results of model checking the AODV implementations. Finally, Section 4.7 compares these results with those obtained by using an alternate technique, static analysis, to find errors in these AODV implementations.

4.2 Description of the AODV Protocol

This section describes the AODV protocol in brief; the reader is referred to [25] for complete details of the protocol.

At each node, AODV maintains a routing table. The routing table entry for a destination contains three essential fields: a next hop node, a sequence number and a

hop count. All packets destined to the destination are sent to the next hop node. The sequence number acts as a form of time-stamping, and is a measure of the freshness of a route. The hop count represents the current distance to the destination node.

Suppose there are two nodes a and b in the network such that b is the next hop of a to some destination d. Also, suppose the sequence number and hop count of the routes to d at a and b are $(seq_a, hcnt_a)$ and $(seq_b, hcnt_b)$ respectively. Then the AODV protocol maintains the following property at all times:

$$(seq_a < seq_b) \vee (seq_a = seq_b \wedge hcnt_a > hcnt_b)$$

In other words, b either has a newer route to d than a, or b has a shorter route that is equally recent. Under this partial order constraint, the protocol is guaranteed to be free of routing loops [6].

In AODV, nodes discover routes in a request-response cycle. When a node needs a new route to a destination, it broadcasts a route request, *RREQ*, for that destination. The RREQ is flooded in the network until it reaches a node that has a valid route to the destination that is as recent or more recent (i.e. greater sequence number) than the entry the requesting node currently has. This node (which can be the destination itself) sends a route response, *RREP*, to the requesting node which retraces the path taken by the RREQ. In the process, every node in the path sets its next hop to be the node from which it heard the RREP.

The RREQ-RREP cycle is shown in Figures 4.1-4.3. These figures show two nodes a, b and the destination d, in a network where b can reach a and d but a and d cannot reach each other. The routing table entries include the next hop node and the *(sequence number, hop count)* pair. Each node has a route to itself with a next hop to itself, a sequence number 1 and hop count 0. Additionally, node b can reach d. The route has a next hop d, a sequence number 1 and hop count 1. Similarly d has a route to b.

In Figure 4.1, node a initiates an RREQ for destination d. This RREQ contains the current sequence number of a's route to d (0) and also the sequence number of its self route (1). Node b receives the RREQ and responds with an RREP(Figure 4.2) as

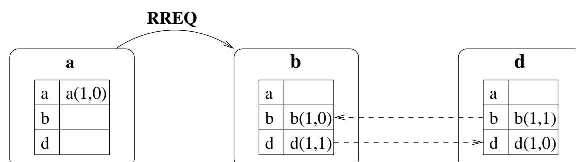

Figure 4.1: Node a initiates an RREQ for destination d.

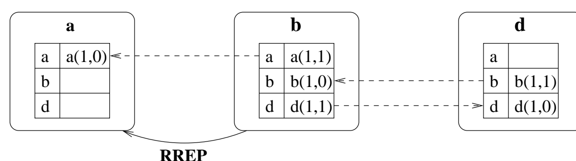

Figure 4.2: Node b responds with an RREP.

it has a valid route to d. Additionally, b installs a *reverse-route* to a with a sequence number of a's self route from the RREQ and a hop count of 1. RREP contains the current sequence number and hop count of node b's route to d, which in this case is $1, 1$. When node a receives the RREP, it installs a route to d with a sequence number 1 and a hop count 2.

Whenever a node detects a link failure to its next hop to a destination, it invalidates its current route to the destination by incrementing the sequence number and setting the hop count to infinity. It then broadcasts a route error (RERR) message to other nodes that might be using its currently invalid route to the destination. In Figure 4.4, when node b detects a link failure to d, it invalidates its current route to d and increments the sequence number to 2. It sends this new sequence number through an RERR message to a. When a receives the RERR message it updates its sequence number to d to 2 and invalidates its route by setting the hop count to

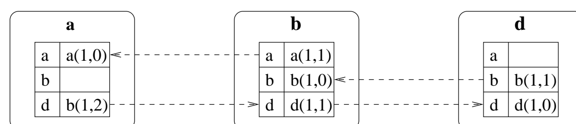

Figure 4.3: State after Node a receives an RREP.

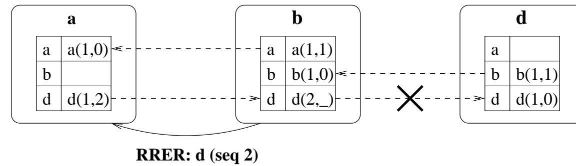

Figure 4.4: Node b sends an RERR message when it detects a link failure to Node d.

infinity.

4.3 The AODV Implementations

This case study uses three implementations of the AODV protocol: *mad-hoc* (Version 1.0) [60], *Kernel AODV* (Version 1.5) [56], and *AODV-UU* (Version 0.5) [34]. All the three implementations are publicly available and were recommended by the AODV protocol authors at the time of this case study. These three implementations differ in their levels of maturity and CMC is able to find errors in all of them. Specifically, the AODV specification error is present in all these three implementations. The following is a brief summary of these implementations.

The mad-hoc implementation [60] is one of the first publicly available AODV implementations and was released two years before the case study. This implementation runs as a user space daemon and contains approximately 5500 lines of code.

The Kernel AODV implementation [56] is built by NIST and is based on the mad-hoc implementation. It contains 7500 lines of code and runs as a loadable kernel module in Linux and ARM based PDAs. This implementation was released less than a year before the case study and has since been under active development.

The AODV-UU implementation runs as a user space daemon on Linux and has been ported to the ns-2 [63] simulator. It contains roughly 7700 lines of code and was released a year before this case study. This implementation is developed by a group devoted to building a simulation testbed [61] for ad-hoc routing protocol implementations and is therefore likely to be heavily tested using their testbed.

Enabling Condition	**AODV Events**
Invalid or no route to destination	Initiation of route request
Pending message in the network	Receipt of AODV message
Valid route in the routing table	Timeout of a route
Always enabled	Detection of link failure
Always enabled	Node reboot

Table 4.1: The set of event handlers used in AODV model checking.

4.4 The AODV model

This section describes the process of building a CMC model for the three AODV implementations. Not coincidentally, the AODV protocol is similar to the running example used in Section 2.3 and the following description mirrors the discussion in that section.

4.4.1 Processes and Transitions

The first step in creating a model involves defining CMC processes to execute the implementation code and then mapping the process transitions to the different executions possible in the implementation. The CMC model consists of a number of processes each executing the AODV implementation. These processes communicate over a network model implemented in the shared memory region.

All the three AODV implementations are single threaded and have an event dispatch loop that calls various event handlers. Accordingly, each process in the CMC model has one thread that executes the event dispatch loop. The initial state of each process is obtained by executing the code from the `main()` function till the implementation enters the event dispatch loop. This code appropriately initializes the various data structures required to further process protocol events.

A transition of the model corresponds to the processing of one AODV event. CMC nondeterministically (§2.3.5) triggers one of the events according to their enabling conditions described in Table 4.1. On each event, the event dispatch loop calls the appropriate event handler in the implementation, triggering the execution of a CMC

Types of Checks	**Examples**
Generic Assertions	Segmentation violations, memory leaks, dangling pointers.
Routing Loop Invariant	The routing tables of all nodes do not form a routing loop.
Routing Table Assertions	At most one routing table entry per destination. No route to self in the AODV-UU implementation. The hop count of the route to self is 0, if present. The hop count is either infinity or less than the number of nodes in the network.
Assertions on Message Fields	All reserved fields are set to 0. The hop count in the packet can not be infinity.

Table 4.2: Properties checked in AODV.

transition. Additionally, the model simulates a node reboot by clearing the contents of the routing table.

4.4.2 The Environment Model

The environment model consists of a network modeled as a bounded-length, unordered message queue. The environment simulates a message loss by nondeterministically dequeuing a message. The message queue is shared by all of the nodes and thus models a completely connected topology.

The implementations use a wrapper function to send network packets. The environment provides an alternate definition to the wrapper function to copy packets to the network model. Additionally, for the Kernel AODV implementation, the environment provides implementations for twenty-two kernel functions (such as kmalloc and printk) and a user space version of the socket buffer library.

4.4.3 Correctness Properties

Table 4.2 lists the correctness properties checked by the AODV model. Apart from the generic assertions checked by CMC, the model contains a global invariant that checks for routing loops. The model also performs sanity checks on the routing table entries

Protocol	**Checked Code**	**Specif-ication**	**Environment**			**Abstract State Gen.**
			network	**stubs**	***skbuff***	
mad-hoc	3336	301	400	100	-	165
Kernel AODV	4508	301	400	266	1210	179
AODV-UU	5286	332	400	128	-	185

Table 4.3: Lines of implementation code vs. CMC modelling code.

and the network messages, such as range violations of the fields. The correctness specifications are mostly shared by the three implementations.

Table 4.3 shows the lines of code from the three implementations that are executed by the transitions in the model against the lines of code required for constructing the model itself. AODV-UU uses a different representation of the routing table and thus requires additional correctness specifications. The network model of the environment is shared by all implementations.

4.5 Dealing with State Space Explosion

The state space of the AODV protocol is essentially infinite. The protocol allows an arbitrary number of nodes in a network. Also, each node has two types of unbounded counters, a *sequence number* to measure the freshness of a route and a *broadcast id* that is incremented by a node on each broadcast. To do any effective search in such an infinite state space, it is necessary to bound the search. The model in this case study is down-scaled to run with 2 to 4 processes. Also, the model discards any state in which the sequence numbers or the broadcast ids exceeds a predefined limit. Additionally, the size of the message queue in the network is limited to sizes of 1 to 3.

Time values stored in the state are another source of state space explosion. For instance, every route response (RREP) contains a lifetime field that determines the freshness of the route. On receipt of this packet, a node adds the lifetime to the current clock value to determine the time at which the route becomes stale. This absolute value is stored in the routing table and can thus increase the state space

	mad-hoc	Kernel AODV	AODV-UU
Mishandling `malloc` failures	4	6	2
Memory Leaks	5	3	-
Use after free	1	1	-
Invalid Routing Table Entry	-	-	1
Unexpected Message	2	-	-
Generating Invalid Packets	3	2 (2)	2
Program Assertion Failures	1	1 (1)	1
Routing Loops	2	3 (2)	2 (1)
Total	18	16 (5)	8 (1)

Table 4.4: Number of bugs of each type in the three implementations of AODV. The figures in parenthesis show the number of bugs that are instances of the same bug in the mad-hoc implementation.

size. The AODV model gets around this problem by modelling route timeouts as nondeterministic events and setting all time variables to predefined constants. Also, the environment of the model contains a definition of the `gettimeofday()` function that always returns a constant value. The handling of time in the model can miss timing related errors and can potentially lead to false positives when an error reported can be caused by a sequence of timeouts that is impossible in the real protocol.

Also, the AODV model contains hand-written code to traverse the routing table (implemented as a linked list in the mad-hoc and Kernel AODV implementations, and as a hash table in the AODV-UU implementation). This traversal code creates a canonicalized representation of the routing table, which along with some selected global variables forms the abstract state of an AODV node of the model. The amount of lines required for this traversal code is shown in the last column of Table 4.3. CMC uses this abstract state as a heuristic to guide the search (§3.6). This heuristic accelerated the discovery of errors and produced shorter error traces.

4.6 Results

Table 4.4 summarizes the set of bugs found using CMC in the three AODV implementations. The bugs range from simple memory errors to protocol invariant violations.

CMC found a total of 42 bugs of which 35 are unique. The Kernel AODV implementation has 5 bugs (shown in parenthesis in the table) that are instances of the same bug in mad-hoc. Also, the AODV specification bug causes a routing loop in all three implementations.

Currently, CMC stops after finding the first bug in the model. It prints the failed assertion and a trace of events starting from the initial state to the error state. After a bug is fixed, CMC is run again to find bugs iteratively. Most bugs were found within minutes of model checking time; the longest took roughly 40 minutes.

The following sections describe the bugs at a high level to give a feel for the breadth of coverage and focus on four of the more interesting bugs to give a feel for its depth.

4.6.1 Memory errors.

The first three error classes illustrate the mishandling of dynamically allocated memory: not checking for allocation failure (12 errors), not freeing allocated memory (8 errors), and using memory after freeing it (2 errors).

All implementations check that the pointer returned by `malloc` is not null. However, functions that call `malloc` can also indirectly return null pointers when allocations fail. The code only erratically checks such cases. Since CMC directly executes the implementation, such errors manifest as segmentation faults during execution.

Most of the memory-related bugs are straightforward. However, there are several interesting errors where the code correctly checks for allocation failure, but its recovery code is broken. Figure 4.5 gives a representative error. Here, the code attempts to allocate `rerrhdr_msg.dst_cnt` temporary message buffers. It correctly checks for `malloc` failure and breaks out of the loop. However, the code after the loop assumes that `rerrhdr_msg.dst_cnt` list entries were indeed allocated. This assumption leads to two bugs. The first (intraprocedural) error attempts to dequeue `rerrhdr_msg.dst_cnt` buffers off of the `rerrhdr_msg.unr_dst` list in order to free them. Since the list has fewer entries than expected, the code will attempt to use a null pointer and get a segmentation fault. The second (interprocedural) error, in

```
/*aodv_deamon.c:aodv_recv_message:*/
...
for(rerri=0; rerri<rerrhdr_msg.dst_cnt;rerri++)
{
   if (!(tp = malloc(sizeof(*tp))))
      break; /* Skip to next packet */
   tp->next = rerrhdr_msg.unr_dst;
   rerrhdr_msg.unr_dst = tp;
   ...
}
// BUG: assumes rerrhdr_msg.dst_cnt buffers
// were allocated!
rec_rerr(info_msg, &rerrhdr_msg);

// Free the list of structs sent to rec_rerr()
for(rerri=0; rerri<rerrhdr_msg.dst_cnt;rerri++)
{
  // BUG: Can be NULL if malloc failed above!
  tp = rerrhdr_msg.unr_dst;
  rerrhdr_msg.unr_dst=rerrhdr_msg.unr_dst->next;
  free(tp);
}
```

Figure 4.5: Mishandled `malloc` failure: if `malloc` fails, the loop will exit after allocating less than `rerrhdr_msg.dst_cnt` buffers. The two errors are in code that assumes `rerrhdr_msg.dst_cnt` buffers were allocated. Both lead to segmentation faults.

`rec_rerr`, similarly tries to walk over the `rerrhdr_msg.dst_cnt` list entries and seg faults because the list is too short.

Most of the memory leaks are similarly caused by mishandled allocation failures. Commonly, code would attempt to do two memory allocations and, if the first allocation succeeded but the second failed, would return with an error, leaking the first pointer.

4.6.2 Unexpected messages

CMC detected two places where unexpected messages causes mad-hoc to crash with a segmentation violation. Figure 4.6(a) shows one of the errors. The error happens because AODV encodes state in its messages. In this error:

1. The current node n receives a Route Request (RREQ) message from node *req* requesting a route to node *dst*.
2. Node n inserts a *reverse route* to *req* in its routing table.
3. Then, n looks up the route to *dst* in its routing table.
4. If the route is not there, n re-broadcasts the RREQ message. The RREQ message contains the IP address of both the destination node *dst* and the requesting node *req*.
5. The response to this request, a Route Response (RREP) message, includes both the route to *dst* and the IP address of *req*.
6. Node n inserts the new route to *dst* in its routing table. It then attempts to relay this route to *req* by looking up the route to *req*. In the normal case, this lookup will return the reverse route inserted in Step 2.

This last step causes the error. The code assumes the normal case and uses the result of the routing table lookup for *req* without checking for null. However, the lookup could fail for two reasons. First, if the machine has rebooted, the implementation will start with an empty routing table. If an old RREP message arrives after the reboot, the lookup of *req* will return a null pointer. Second, an attacker could send a bogus RREP with a node address that does not exist, crashing the router.

4.6.3 Invalid messages

There are 4 cases of invalid packets being created, 2 cases of using uninitialized variables (these could not be detected by gcc -Wall), and 2 cases where invalid routes are used to send routing updates, violating the AODV specification (Figure 4.6(b)

```
/* madhoc:rrep.c:rec_rrep */
...
/* If I'm not the destination of the RREP
   I forward it */
if(my_rrep->src_ip != my_info->ip_pkt_my_ip) {
  ...
  // Get the entry to the source from RT.
  rt_src = getentry(my_rrep->src_ip);

  // BUG: rt_src may not exist!
  if (add_precursor(rt_src, rt->nxt_hop) == -1)
  ...
  // Send gratuitous RREP to destination
  // BUG: rt_src can be invalid
  // (i.e rt_src->hop_cnt == 255 )
  // must check after getentry.
  my_rrep.hop_cnt = rt_src->hop_cnt;
  if (send_datagram(my_info, &my_rrep,
            sizeof(my_rrep)) == -1)
    ...
```

Figure 4.6: Two bugs: an unexpected message and an invalid route response. (a) An unexpected route-response (RREP) message causes `getentry` to return null, crashing the machine. (b) If a route returned by `getentry` has been invalidated the hopcount will be 255. However, the code does not check for this and sends the message.

```
/* Check if the entry already in RT
   is better than the received */

if(rt->dst_seq >= my_rrep->dst_seq){
  ...
  // BUG: should be
  // my_rrep->hop_cnt + 1 >= rt->hop_cnt)
  if(my_rrep->hop_cnt >= rt->hop_cnt)
   return 0;
}
...
rt->hop_cnt = my_rrep->hop_cnt + 1;
```

Figure 4.7: The mad-hoc implementation compares the hop counts of the route received in RREP and its existing route in the routing table. AODV specification requires that this comparison be done *after* an increment

gives a representative example). CMC also detected 2 instances of integer overflow which resulted in program assertion failures. The implementations use an 8 bit integer to store the hop counts and use 255 to represent a hopcount of infinity. In these error cases, an infinite hopcount is erroneously incremented to 0.

4.6.4 Routing loops

CMC found three routing loops that are caused by implementation errors. The description of two of these errors follows. A routing loop is caused when the implementation fails to increment a sequence number while processing specific RERR messages. Another loop is caused when the implementation performs a sequence number comparison before a subsequent increment, while the AODV specification requires the comparison to be done after the increment.

4.6.5 The Specification Bug

CMC found a routing loop that is due to an error in the AODV protocol specification. This bug involves the handling of RERR messages. When a node receives an RERR

```
/* madhoc:rerr.c:rec_rerr */
...
// Get pointer to route table for destination IP
tmp_rtentry = getentry(tmp_unr_dst->unr_dst_ip);
if(tmp_rtentry != NULL && ...) {
  // BUG: uses sequence number from incoming
  // message in tmp_unr_dst without validation.
  // Should check:
  //   if(tmp_rtentry->dst_seq >=
  //          tmp_unr_dst->unr_dst_seq)
  //      return -1;
  tmp_rtentry->dst_seq = tmp_unr_dst->unr_dst_seq;
```

Figure 4.8: The specification bug: the sequence number from an incoming message is used without validation, causing "time" to go backwards when messages are reordered. Fortunately, while the error was not obvious (surviving 6 rounds of specification revisions) the fix is trivial.

from its next hop, it sets the sequence number of its route to the sequence number in an RERR message. Under normal conditions this is the right thing to do. However, when the underlying link layer can reorder messages, the RERR message might have an outdated sequence number resulting in the node setting its sequence number to an older version. This can ultimately result in a routing loop. This bug was mentioned to the authors of the protocol with a suggested fix. Both the bug and the fix were accepted by the protocol authors[71]. Figure 4.8 gives both the error and the fix.

The specification bug was found by running 4 AODV nodes using a depth-first search of the state space. CMC came up with an error trace of length 93. Using best-first search, it was possible to find traces as short as 27. Performing a breadth-first search of the state space would give the shortest trace. However, breadth-first search on AODV ran out of resources without finding the bug. A carefully hand-crafted simulation of the bug required at least 20 transitions. An error of this complexity would be very difficult to catch using conventional means of testing.

4.7 Comparison with Static Analysis

The results described in the previous section demonstrate a clear success of using CMC to find errors in systems. However, these results do not evaluate the extent to which CMC checks a given system. Specifically, it is not clear if the system contains errors even after a thorough inspection by CMC. Determining if CMC misses errors at all, and if so, the reasons behind these missed errors will provide an opportunity to improve CMC.

As one way to measure the bug-finding effectiveness of CMC, this section describes the results from using static analysis to find errors in the same AODV implementations. Static analysis described in this section is done using the *meta-compilation* tool, MC [31]. The results from static analysis are then compared with the results obtained from CMC.

MC found a total of 34 errors in the three AODV implementations of which CMC missed 13 of them. Surprisingly, none of these error misses are due to CMC's inability to completely search the state space. As will be demonstrated below, various shortcomings of the environment model used in the case study restrict CMC from exploring behaviors that lead to these missed errors. This reinforces the need for a coherent environment model, the topic of Chapter 5.

Apart from measuring the effectiveness of CMC, this study provides an opportunity of comparing the two approaches of static analysis and model checking for finding bugs in systems. In general, applying static analysis to a new system takes several orders less effort than applying model checking. Moreover, due to the inherent complete path coverage obtained, static analysis is able to find more errors than model checking. However, the strength of model checking comes in its ability to check for a richer set of properties that are not amenable to static analysis.

4.7.1 Overview of MC

At a high level, the checking done by MC is based on *checkers* or compiler extensions that are dynamically linked into the compiler and applied down a control-flow graph representation of source code. Conceptually these extensions examine one path at a

time, rather than using a more traditional dataflow framework that would conflate information at program joint points. Extensions can perform either intra-procedural or inter-procedural analysis at the discretion of the checker writer. In practice, the approach has been effective, finding hundreds to thousands of errors in Linux, BSD, and various commercial systems.

There are many papers on the approach. [31] gives a reasonable, though dated overview, while [41] gives a more up-to-date view of MC. The tutorial in [18] has a series of checker examples and references that the interested reader can use for a more thorough introduction.

There are key differences between MC and most other static analysis approaches that should be kept in mind when analyzing the comparison results (§4.7.2). First, MC is unsound: code with errors can pass silently through a checker. MC is optimized to find bugs, rather than demonstrating their absence. In particular, when MC cannot determine a needed fact it does not emit a warning. In contrast, a sound approach would conservatively emit an error report whenever it cannot prove the error cannot occur. Unsoundness allows MC to aggressively check properties beyond the practical reach of sound tools, which would overwhelm the user with false positives.

Second, MC uses relatively shallow data flow analysis rather than a deeper simulation based approach such as in PREfix [15]. While MC performs a mild amount of path sensitive analysis to prune infeasible paths [41], MC does not model the heap, does not track most variable values, and has limited aliasing information. The flip side of this shallowness is that MC does not require building an accurate, working model of the environment.

Third, MC tries as much as possible to avoid the need for annotations, in part by using statistical analysis to infer properties to check [32] (such as which functions must be paired, which functions can return null, etc.). The need for annotations would dramatically increase the effort necessary to use the tool, and to an extent diminish its advantages

		Bugs Found		
		by CMC & MC	**by CMC alone**	**by MC alone**
Generic Properties:	Mishandling `malloc` failures	11	1	8
	Memory Leaks	8	-	5
	Use after free	2	-	-
Protocol Specific:	Invalid Routing Table Entry	-	1	-
	Unexpected Message	-	2	-
	Generating Invalid Packets	-	7	-
	Program Assertion Failures	-	3	-
	Routing Loops	-	7	-
	Total	21	21	13

Table 4.5: Comparing MC and CMC

4.7.2 Results

MC checked for generic errors such as memory leaks and invalid pointer accesses. The entire process of checking the three implementations and analyzing the output for errors took two hours. MC found a total of 34 bugs. Table 4.5 compares the bugs found by MC and CMC. It classifies the bugs found into two broad classes depending on the properties violated: generic and protocol specific.

4.7.2.1 Generic Properties

In the class of generic errors, MC found many more bugs than CMC. Except for one, MC found all the bugs that CMC could find.

MC is able to find more bugs as it checks all paths in all code that it can compile. It does not require abstracting away parts of the system, nor does it require coming up with inputs to drive the system to execute a given code path. In contrast, CMC can only execute code triggered by the specific environment model. Of the 13 errors not found by CMC, 6 are in parts of the code that are either not included in the model or stubbed out during environment modeling. For instance, MC found two cases of mishandled `malloc` failures in multicast routing code. The CMC model omitted this code.

CMC missed more errors due to subtle mistakes in the environment model. For

example, the mad-hoc implementation uses a `send_datagram()` function to transmit a packet to the network. A memory leak in the implementation is triggered only when this function fails. The environment model however, erroneously modeled the `send_datagram()` function to always succeed. Thus, CMC never detected this memory leak. CMC missed a total of 6 errors due to such errors in the environment. MC found 1 more error in dead code that can never be executed by any CMC model.

The one error that MC missed is the one described in Figure 4.5. It requires reasoning about the length of a linked list. Here, the `rec_rerr` function in the mad-hoc implementation assumes that the input argument points to a linked list of a particular length. However, when this linked list is allocated, a `malloc` failure can cause the list to be smaller than expected, leading to a null pointer violation. Present static analyzers have difficulty detecting such invariants of heap objects.

4.7.2.2 Protocol Specific Properties

In the class of protocol-specific errors, CMC found 21 errors while MC found none. While this was partly because MC did not check for protocol-specific properties, many of the errors would be difficult to find statically. Properties such as routing loops involve invariants of objects across multiple processes. Detecting such loops statically would require reasoning about the entire execution of the protocol, a difficult task.

Many properties are local to a process, but still difficult to detect statically without generating many false positives. For instance, AODV-UU requires that the routing table of a node does not contain a route to itself. This would imply that the node has to route packets to a neighbor to reach itself. This implementation inserts a route to the *src-id* field in a received route response message. However, there are no checks to ensure that the *src-id* received is not the same as the current node identifier. While a static analyzer could look for such unchecked routing table insertions, it cannot determine which of these are harmless. Even on a manual inspection, it appears that *src-id* can never be the node identifier as no node sends a route response to itself. However, CMC found a specific instance of the protocol in which such a packet can be generated. This bug involves a *Gratuitous Route Response*, which is a later addition to the protocol and is an optional optimization to improve the performance

of bidirectional transport protocols like TCP. In this optimization, a node *spoofs* a route response packet with a different *src-id*. Of course, this can be received by the original source node, causing the bug mentioned above.

A second advantage model checking has is that it checks for actual errors, rather than having to reason about all the different ways the error could be caused. If it catches a particular error type it will do so no matter the cause of the error. For example, a model checker such as CMC that runs code directly will detect all null pointer dereferences, deadlocks, or any operation that causes a runtime exception since the code will crash or lock up. Importantly, it will detect them without having to understand and anticipate all the ways that these errors could arise. In contrast, static analysis cannot do such end-to-end-checks, but must instead look for specific ways of causing a given error. Errors caused by actions that the checker does not know about or cannot analyze will not be flagged.

A good example is the error CMC found in the AODV specification, shown in Figure 4.8. This error arises because the specification elides a check on the sequence number of a received packet. Here, the node receives a packet with a stale route with an old sequence number. The code (and the specification) erroneously updates the sequence number of the current route without checking if the route in the packet is valid. This results in a routing loop. Once the cause of the routing loop is known, it is possible (and easy) to statically ensure that all sequence number updates to the routing table from any received packets involve a validity check. However, there are only a few places where such specialized checks can be applied, making it hard to recoup the cost of writing the checker. Moreover, exhaustively enumerating all different causes for a routing loop is not possible. On the other hand, a model checker can check for actual errors without the need for reasoning about their causes.

4.7.3 Summary

One goal of this comparison study is to evaluate the extent to which CMC tests a given system. As this study elaborates, the behaviors explored by CMC are restricted to only those that are specifically triggered by the environment model. In order to

increase the effectiveness of CMC, it is essential to design environment models that maximize the behavioral coverage of a given system. Designing such environment models will be discussed extensively in Chapter 5.

Another goal of this section is to compare the merits of the two approaches for finding bugs in system software. In the properties that could be checked by both methods, static analysis is clearly more successful: it took less time to do the analysis and found more errors. The static analysis simply requires that the code be compiled, while model checking a system requires a carefully crafted environment model. Also, static analysis can cover *all* paths in the code in a straightforward manner. On the other hand, a model checker executes only those paths that are explicitly triggered by the environment model.

A common misconception is that model checking does not suffer from false errors, while these errors typically inundate a static analysis result. However, this not to true. False execution paths in the model checker can be triggered by erroneous environments, leading to false errors. These errors can be difficult to trace and debug. Meanwhile, false errors in static analysis typically arise out of infeasible paths, which can be eliminated by simple analysis or even unsubstantial manual inspection.

The advantage of model checking is in its ability to check for a richer set of properties. Properties that require reasoning about the system execution are not amenable to static checking. Many protocol specific properties such as routing loops and protocol deadlocks fall in this category. A model checker excels in exploring intricate behaviors of the system and finding errors in corner cases that have not been accounted for by the designers and the implementors of the system. However, the importance of checking these properties should significantly over-weigh the additional effort required to model check a system.

These results suggest that while model checking can get good results on real systems code, in order to justify their significant additional effort they must target properties not checkable statically.

Chapter 5

Environment Modeling - The TCP Case Study

5.1 Introduction

This chapter discusses the challenges in constructing an environment model for a large and complex system in the context of the second study of this thesis. This case study involves the Linux TCP implementation.

The key motivation behind this case study is to evaluate the effectiveness of CMC in model checking large and well tested systems. The TCP protocol [72] is fairly mature and well audited. The particular TCP implementation used in this case study, from the stable 2.4.19 release of the Linux kernel, is widely used in the Internet today and thus heavily tested. This implementation contains approximately $50,000$ lines of code and implements a full-fledged TCP protocol [72, 11, 77, 62, 5].

In CMC, the system description checked is the implementation itself. However, the user still needs to provide an environment model that closes the system and adequately triggers the system behaviors by providing inputs to the system. In the previous case study (§4), building an environment model for the AODV protocol implementations was simple. The protocol required a few well defined inputs, and all the implementations used in that case study were small and easy to manage.

These are not true for the Linux TCP implementation. First, the TCP protocol

itself is very complex involving many control sequences to setup and tear-down connections. Additionally, a TCP data transfer involves many independent mechanisms such as flow control and congestion control that execute simultaneously. The Linux TCP implementation is a complex piece of software and is roughly ten times larger than the AODV implementations. The implementation communicates with the user through a rich set of system calls and has a variety of options that control various parts of the implementation. The difficulty of environment modeling for such large systems is known to an extent in the model checking literature [70], but is typically underplayed.

5.1.1 Chapter Overview

Section 5.2 enumerates the difficulties in modeling the environment for the Linux TCP implementation while discussing an initial attempt that failed. Then, Section 5.3 mentions the lessons inferred from this failed approach and proposes the use of *well-defined* interfaces to simplify the task of environment modeling. This requires an extreme approach of running the *entire* Linux Kernel along with the TCP implementation in CMC. Section 5.4 describes this approach. Section 5.5 discusses methods to iteratively refine environment models to improve the search effectiveness. Finally, Section 5.6 presents the model checking results of this case study.

5.2 Difficulties in Environment Modeling

This section describes the difficulties in modeling the environment for a large system. While this discussion is specific to the particular TCP implementation used in this case study, one can expect similar problems for any complex system.

5.2.1 Environment Modeling Overview

Building an environment model for a system consists of the following steps. Starting from an implementation, the first task is to select specific implementation modules that comprise the system being model checked. Next, a suitable environment model

should be defined that appropriately closes this system. Specifically, the environment should contain implementations of all external functions that the system calls, and contain models of all physical entities, such as the user and the network, that interact with the system. Finally, the environment should provide inputs to the system to trigger behaviors in the system.

With an implementation-level model checker like CMC, this process is very similar to building a harness for unit testing. However, building a comprehensive environment model for a large and complex system can be difficult. The following sections enumerate these difficulties in the context of an initial attempt that applied conventional methods to build an environment model for the Linux TCP implementation, but failed.

5.2.2 Defining the System Boundary

The system to be model checked is typically embedded in a larger execution context. For instance, the Linux TCP implementation is present in the Linux kernel, and closely interacts with other kernel modules. Before model checking, it is first necessary to define a *system boundary* that separates the system from its environment. Only the implementation modules inside the system boundary are executed during model checking. These modules can make external calls to functions implemented by modules outside the system boundary. Such functions should be appropriately implemented by the environment in order to close the system. Typically, it is only necessary to include simplified stubs for these functions in the environment model. This reduces the detail in the environment.

Conventional wisdom dictates that the system be defined along a *minimal* boundary that includes as few implementation modules as possible in the system. Such a minimal boundary reduces the size of the system and the state space explored during model checking.

Sometimes, the environment model designer can conservatively decide to include more modules in the system in order to *simplify* the system boundary. For instance, if a module in the system is tightly coupled with some external module, the system

boundary consists of the many interface functions between these two modules. Instead of providing stubs for all these functions, the environment model designer can choose to include the external module in the system. While this can simplify the system boundary, the added module can call additional functions for which stubs need to be provided in the environment. Also, adding modules to the system can increase the size of the system state, and thus potentially the size of the state space.

Determining the most appropriate system boundary that minimizes the system state while simplifying the task of providing stub functions is not straightforward. It requires a good understanding of the dependencies between the different modules in the system and a *lot* of trial and error. For the Linux TCP implementation for instance, its comprehensive interaction with the rest of the kernel meant that despite repeated attempts, the minimal boundary still contained as much as 150 interface functions.

5.2.3 Closing the System

Once the choice of the system boundary is made, the system needs to be closed by providing stubs for all the external functions at the system boundary. Writing stubs that perfectly replicate the *exact* semantics of these often poorly documented interfaces is extremely difficult. In practice, these stubs have a myriad of subtle corner-case mistakes. Since CMC is tuned to find inconsistencies in corner cases, it will generate a steady stream of bugs, all false.

These false errors can be very hard to debug and fix. For instance, after days of debugging a memory leak of a socket structure was found to be a result of an incorrect stub implementation in a seemingly unrelated timer module. The TCP implementation uses a function `mod_timer()` to modify the expiration time of a previously queued timer. The return value of this function depends on whether the timer is pending when the function is called. The initial stub implementation did not capture this behavior. This incorrect stub confused the reference counting mechanism of the socket structures: as TCP timers are members of the socket structure, a queued timer amounts to an extra reference to the parent socket. This led to the memory leak.

It is conceivable that with more debugging, the stubs for all the 150 functions could be perfected. In practice, however, the environment model never reached a fixed point even after months of effort. In fact, each additional false error seemed harder to diagnose. Also, as seen in Section 4.7, faulty stubs can silently miss errors, reducing the effectiveness of the model checker.

5.2.4 Providing Input Triggers

Once an environment model that closes the system is built, the final step is to provide inputs to trigger the system behaviors. If the environment does not adequately constrain these inputs, it can generate invalid input sequences not possible in a real system. Such invalid sequences can result in false error reports. As an example, a user process can never get access to a partially connected TCP socket using the standard system call interface. To model this constraint, certain socket functions need to be carefully disabled at appropriate TCP states.

5.3 Lessons: Use Well-Defined Interfaces

All of the problems in the previous attempt arise because of the complex and undocumented interfaces between the Linux TCP implementation and the rest of the kernel modules. Defining a system boundary that includes such interfaces complicates the task of creating an environment model as the model should faithfully reproduce the exact semantics of all the interface functions and the complicated dependencies between them.

This suggests that the system boundary should be restricted to only use *well-defined* interfaces that are well documented and clearly understood. Using well-defined interfaces greatly simplifies the environment model. As the semantics of the interface functions are well known, correct stubs for these functions are easy to write. Moreover, the dependencies and the ordering constraints on the inputs across well-defined interfaces are easy to specify in the environment.

For the Linux kernel, there are only two *well-defined* interfaces: the system call

interface that defines the interface between the kernel and the user processes; and the *hardware abstraction interface* that defines the interface between the kernel and the hardware architecture. Though Linux does not explicitly define a hardware abstraction interface, such an interface is implicitly defined for most kernels to simplify the task of porting the kernel to different architectures. Starting from the TCP implementation modules, defining a system boundary along these two well-defined interfaces includes the *entire* Linux kernel in the system.

Obviously, there are some challenges in this approach. First and foremost, model checking the TCP implementation now requires running the entire Linux kernel as a CMC process in user space. However daunting this might seem, it is possible (§5.4.1). Also, the size of the system state is orders of magnitude larger, as the system state should now include the entire kernel state. However, the incremental techniques (§3.2.3) that CMC implements in effect automatically extract the TCP relevant state from the kernel state during model checking. Finally, there is a danger that the system can generate behaviors in the kernel that are not relevant to the TCP protocol. This can result in unnecessary blowup of the state space. To avoid this, the environment should be carefully designed to avoid triggering non-TCP related behaviors. For instance, by ensuring that page faults are not generated, the virtual memory subsystem of the kernel does not get triggered.

Despite these challenges, the ease with which the environment can be modeled at these well-defined interfaces clearly outweighs the additional effort required to execute the entire kernel in CMC. Moreover, given the experience with the conventional approach, this might be the *only* way to generate a coherent environment model for the Linux TCP implementation. An added benefit of using this approach is that well-defined interfaces are less likely to change as the Linux kernel evolves. This allows the reuse of the same environment model for future versions of the TCP implementation.

This extreme approach of running the entire kernel to just check the TCP implementation is possible only because CMC runs the implementation directly without requiring an alternate description. Such an approach is not practical if the system description is manually constructed or automatically extracted from the implementation.

5.4 The Linux TCP Model

5.4.1 Running the Entire Kernel in User Space

Before model checking a system, the implementation of that system should run as a CMC process in user space. For the approach discussed above, the system consists of the Linux kernel, making it necessary to run the entire kernel in user space. While this might seem impossible at first sight, many approaches [80, 78, 79] are possible.

The Linux TCP model uses an approach similar to the User Mode Linux (UML) project [80] and reuses a lot of code from this project. The basic idea involves creating a *virtual architecture* that provides a hardware abstraction interface in software using the system resources available in user space. The Linux kernel is then *ported* to this virtual architecture. Porting the kernel to a new architecture requires changes to some architecture dependent files in the kernel. However, the TCP implementation itself remains unmodified.

Many simplifications are possible when porting the Linux kernel to the virtual architecture. As CMC triggers only TCP related behaviors, only those parts of the architecture required for such behaviors need to be defined. Specifically, it is not necessary to include most of the devices such as the terminal and the hard disk in the virtual architecture.

5.4.2 Processes, Threads and Transitions

Once the TCP implementation can run in user space, the next step involves allocating one or more CMC processes to run the TCP implementation and then appropriately mapping the process transitions to the system executions possible. The TCP protocol consists of two interacting peers and accordingly, the TCP model consists of two CMC processes each executing the Linux kernel. The two kernels communicate with each other using a network model implemented in the shared memory region.

In the kernel, the TCP code can be executed in three contexts: in user context when a user process makes a system call, in the context of a network interrupt handler when a TCP packet is received, and in the context of a timer interrupt handler when

a TCP timer fires. To mimic this behavior a CMC process running the Linux TCP implementation contains the following three threads:

- An *application* thread that makes socket related system calls to the kernel. Of the two application threads in each of the two processes, one behaves as a standard TCP server, while the other behaves as a standard TCP client.
- A *network* thread that emulates a packet interrupt and executes the code to handle packet reception.
- A *timer* thread that emulates a timer interrupt and fires one of the pending timer routines.

These threads once triggered can either execute to completion or can block. In both cases, the threads yield control to the kernel scheduler. In the real kernel, the scheduler would then execute the scheduling algorithm to determine the thread that is scheduled next. In the CMC model, the scheduler is modified to immediately transfer control to CMC. This enables CMC to *nondeterministically* chose the thread that is executed next thereby enabling CMC to explore multiple scenarios from a single state.

Similarly, the kernel timer routine is modified to allow CMC to chose the timer that fires next when multiple timers are enabled. Optionally, CMC can fire timers out of order irrespective of their expiration times. While this can lead to some behaviors not possible in a real implementation, it has the benefit of making the implementation behavior independent of the actual values of the timers.

5.4.3 Environment Model

Once the system boundary is defined along well-defined interfaces, modeling the environment is straightforward, which was the ultimate purpose of using this approach. The Linux kernel along with the virtual architecture already executes as a closed system. Additionally, the environment consists of a network model implemented in the shared memory (§4.4.2). The network model can lose, duplicate, reorder and corrupt messages.

The environment interacts with the system through standard interfaces whose semantics are well defined. For instance, the application thread generates a sequence of system calls to appropriately trigger the behavior of a TCP client or a TCP server. The network model communicates with the Linux kernel through an appropriate network device driver.

5.4.4 Correctness Properties

In addition to the generic properties such as memory leaks that CMC automatically checks (§2.5.1), the Linux TCP model contains checks for the following properties.

5.4.4.1 TCP-specific Assertions

The model contains TCP-specific assertions in the form of boolean functions written in C and evaluated at each state generated during model checking. These assertions check that the TCP data transfer indeed satisfies the reliable byte stream model. Additional checks ensure that any packet generated by the implementation contains a valid checksum.

5.4.4.2 Kernel Resource Leaks

The TCP implementation should release all kernel resources after using them. For instance, the implementation should release all socket buffers allocated for a particular connection, once the connection is closed. Failure to do so can result in resource lockup, which subsequently reduces the performance and the availability of the machine. Note that resource leaks are not the same as memory leaks; a TCP connection that fails to release a resource can still have valid references to that resource.

CMC checks for resource leaks by requiring that the entire kernel eventually reach the initial state after completing a TCP connection. CMC reports any violation of this property as a potential resource leak.

There are, however, valid situations when the kernel might *not* reach the initial state. First, most TCP implementations typically *cache* resources without releasing them. Doing so reduces the cost of subsequent allocations. To account for this fact,

CMC generates an initial state in which the resources are *already* cached. Such an initial state can be generated by tracing a TCP connection to completion, which allocates and caches the necessary resources. CMC can perform state space search from the final state of this connection.

Second, the kernel might update many statistics and counter variables during a TCP connection. Due to these differences, the kernel might not reach the initial state. Surprisingly, this works in CMC's favor. CMC uses these differences to supplement its mechanisms to slice unnecessary variables from the state (§3.5). After some model checking iterations, CMC identifies all such statistics and counter variables and slices them from the state.

5.4.4.3 TCP Specification Conformance

Checking for the TCP specification conformance is a bit involved. The TCP protocol is specified informally as a state machine [72], where each transition specifies the set of actions a TCP implementation should perform in response to an event at a particular protocol state. While each such transition of the TCP protocol can be expressed as an assertion involving the current and next state protocol variables, doing so for the entire specification is unwieldy.

Instead, CMC simultaneously executes a TCP reference model along with the TCP implementation. The reference model consists of the basic state machine transitions literally translated from the specification [72] to C. This reference model is around 500 lines of code and is straightforward to implement. During model checking, CMC checks for behavioral consistency between the implementation and the reference model. CMC provides the same set of input events (system calls, network and timer interrupts) to both the implementation and the reference model, and expects their states and the outputs (network packets and system call return values) to be consistent. Many of the inconsistencies found, especially during the initial phases, are not protocol violation errors. They are due to manual errors in the reference model, known errors in the TCP specification [69] and acceptable differences between the Linux implementation and the TCP specification. The reference model is iteratively modified when such false errors are found.

5.5 Refining the Environment Triggers

There still remains one crucial problem in the environment model defined above. While defining the system boundary along well-defined interfaces makes it possible to construct an environment model that generates valid inputs, it is still necessary to design these inputs to trigger *sufficient* behaviors in the system.

The TCP protocol is complex and includes many functionalities that are enabled by various configuration variables and options. Moreover, the user processes interact with a TCP implementation through a rich set of system calls, each of which contain a variety of options. An environment model has to appropriately select a few among the set of possible inputs and enable them during model checking. Enabling very few inputs can severely restrict the behaviors explored leading to missed errors. On the other hand, enabling too many inputs at the same time can drastically increase the state space, which in turn reduces the effectiveness of CMC.

This clearly indicates a need to evaluate different environment models with respect to the effectiveness they achieve for a given amount of resources. Section 5.5.1 presents two metrics to evaluate different environment models and Section 5.5.2 discusses how these metrics can be used to iteratively refine the environment model for the Linux TCP implementation.

5.5.1 The Search Effectiveness Metrics

For a bug-finding tool like CMC, a search effectiveness metric should measure the extent to which the particular implementation is tested. Thus, metrics based on test coverage are naturally applicable for this purpose. CMC uses the following two coverage metrics that are easy to compute.

- **Line Coverage :** An obvious measure is the line coverage achieved during model checking. While this measure need not correspond to how well the system is tested, it is helpful in detecting those parts that are *n*ot explored during model checking.

	Description	Line Coverage	Protocol Coverage	Branching Factor	Additional Bugs
1	Standard server and client	47.4 %	64.7 %	2.91	2
2	Model 1 + simultaneous connect	51.0 %	66.7 %	3.67	0
3	Model 2 + partial close	52.7 %	79.5 %	3.89	2
4	Model 3 + message corruption	50.6 %	84.3 %	7.01	0
	Combined Coverage	55.4 %	92.1 %		

Table 5.1: Coverage achieved during model refinement. The branching factor is a measure of the state space size.

- **Protocol Coverage :** Protocol coverage corresponds to the behaviors of the protocol tested by the model checker. This is measured as the line coverage achieved in the TCP reference model (§5.4.4.3). This roughly represents the degree to which the protocol transitions have been explored.

5.5.2 Iterative Environment Refinement

The metrics discussed in the previous section can be used to iteratively refine the environment model to trigger increasingly more behavior in the system. Table 5.1 describes the coverage achieved during this model refinement process. For a particular environment model, the table reports the cumulative coverage achieved using three search disciplines: breadth-first, depth-first, and random. In random search, each generated state is given a random priority. Table 5.1 also reports the branching factor of the state space as a measure of its size. For the first three models the branching factor is calculated from the number of states in the queue at depth 10 during a breadth first search. For the fourth model, CMC ran out of resources at depth 8, and the branching factor is calculated at this depth.

The first model consists of a TCP client communicating with a TCP server. Once the connection is established, the client and server exchange data in both directions before closing the connection. This standard model discovered two protocol compliance bugs in the TCP implementation. Section 5.6 describes these and additional errors found by CMC.

Starting from this model, the iterative refinement process involves manually inspecting the line coverage and protocol coverage achieved to determine the interesting behaviors not triggered by the current model. In the second model, the server nondeterministically decides to actively initiate a connection. This enables additional transitions in the protocol that handle simultaneous connection of two TCP peers. In the third model, both the client and the server nondeterministically decide to close a connection during data transfer. This improved the protocol coverage and resulted in the discovery of two more errors.

Still, the environment model does not allow enough *bad* behavior. Specifically, the TCP implementation is designed to robust against malformed packets sent accidentally or on purpose by another TCP peer. A model consisting of two correctly functioning TCP implementation will not generate such packets. As an attempt to generate such malformed packets, the environment model is refined to nondeterministically toggle certain key control flags in the TCP packet. Even such restricted packet corruption can result in an enormous increase in the state space, as shown in Table 5.1. However, these corrupted packets triggered a lot of error recovery code in the implementation.

Tweaking the environment the right way to achieve a more effective search still remains an interesting but unsolved problem. Also, it is not clear how much of this refinement process can be automated.

5.6 Results

CMC found four instances where the Linux implementation fails to meet the TCP specification. These errors amply reflect the kind of corner cases CMC is able to test.

The first bug CMC found involves the processing of RST packets. The Linux implementation fails to honor a RST packet in SYN_RCVD state unless the ACK bit is also set. The TCP specification requires that in response to a RST packet an implementation free any resources held by the connection and gracefully inform the application. Figure 5.1 shows the code containing this bug. The function `tcp_check_req`

```
// net/ipv4/tcp_minisocks.c
// Process an incoming packet for SYN_RECV
// sockets represented as an open_request.
struct sock *tcp_check_req(...) {
  ...
  /* You would think that SYN crossing is
     impossible here, since we should have
     a SYN_SENT socket (from connect()) on
     our end, but this is not true if the
     crossed SYNs were sent to both ends by
     a malicious third party. We must defend
     against this,
     ...
     Note: This case is both harmless, and
     rare.  Possibility is about the same
     as us discovering intelligent life on
     another plant tomorrow.
    ...
   */
  if (!(flg & TCP_FLAG_ACK))
          return NULL;

  /// BUG: Should have checked the RST field
  ///      before checking the ACK field

  // Invalid ACK: reset will be sent by listening socket
  if (TCP_SKB_CB(skb)->ack_seq != req->snt_isn+1)
          return sk;
  ...
  //  RFC793: "second check the RST bit" and
  //         "fourth, check the SYN bit"
  if(flg & (TCP_FLAG_RST|TCP_FLAG_SYN))
     goto embryonic_reset;
  ...
}
```

Figure 5.1: The Linux TCP implementation (version 2.4.19) does not honor the RST flag in the SYN_RECV state unless the ACK bit is set. The bug was inadvertently introduced while fixing a “harmless and rare” case. The code prematurely returns when the ACK field is not set on an incoming packet.

processes incoming packets in the SYN_RCVD state. The bug was inadvertently introduced while trying to handle a case of a maliciously generated packet. The fix causes the function to prematurely exit before processing the RST flag. This bug can potentially lead to lockup of kernel resources long after the TCP connection is dead.

The second bug involves incorrect handling of a duplicate SYN_ACK packet. While the specification requires that any duplicate packets be ignored, the Linux implementation fails to do so and faithfully uses the acknowledgment to open its congestion window. The error happens because of an incorrect sequence number check in the ESTABLISHED state.

CMC also found that the implementation fails to implement one transition in the TCP state diagram. If an application prematurely closes in SYN_RCVD state, the specification requires the implementation to gracefully close the connection using the FIN handshake. An acceptable alternative is to perform an abnormal close by sending a RST packet. CMC found that in this case, the Linux implementation dropped the connection without notifying its peer.

The final bug involves a subtle processing of ACK packets. A TCP implementation is required to abort a connection by sending a RST packet when it receives data that cannot be transferred to the application. This can happen, for instance, when the application closes the connection before the data transfer is complete. The bug involves the case when the implementation receives a packet containing both data and an ACK after the application has closed the connection. The Linux implementation blindly processes the ACK field before processing the data. If the ACK opens the congestion window, the implementation exhibits a peculiar behavior. It sends a stream of data packets and immediately follows by a RST packet aborting the connection.

CMC also detected one instance where the TCP specification might be ambiguous. This concerns the transmission of a FIN packet on a zero window. When the receiver advertises a zero window, the Linux implementation queues a FIN packet, even if no *data* is queued. This is definitely the behavior expected by the TCP specification as the sequence number of FIN lies outside the send window. However, it seems that an acceptable solution is to send the FIN packet as zero window probe.

Chapter 6

Conclusion

6.1 Summary

This thesis proposes a model checking approach to find and diagnose deep errors in network protocol implementations. To this effect, the thesis described CMC, a C Model Checker that checks system implementations directly by executing them. By not requiring an alternate description of the code, CMC avoids the upfront cost of writing a description of the protocol in another language, which is required for most conventional model checkers. This allows CMC to scale to large, complex systems. CMC improves its bug-finding effectiveness by applying various reduction techniques and conserves resources by maintaining only the differential information between states.

This thesis validated CMC on two network protocol implementations: AODV and TCP. On three publicly available AODV implementations, CMC found a total of 35 errors including an error in the AODV protocol specification. CMC successfully checked the Linux TCP implementation, demonstrating its applicability to large, complex systems, and found four instances where the implementation fails to meet the TCP protocol specification.

At a high level, this thesis has shown that explicit model checking of implementations is not only possible but also very effective, provided some key optimizations and trade-offs are made.

6.2 Future Work

This thesis leaves open many unsolved issues. The following sections discuss some of them.

6.2.1 Increasing the Scope of CMC

CMC is specifically designed for model checking network protocol implementations. However, the general approach easily extends to other domains of system software such as kernel schedulers, file system implementations, etc. In such domains, concurrency among the multiple threads present in the system plays a major role. To effectively model check such systems, CMC should allow finer granularities of thread scheduling. Using the approach of [14] in CMC is very promising.

6.2.2 Making CMC More "Push Button"

The primary goal of CMC is to make model checking *readily* applicable to large systems. While CMC has greatly reduced the model checking effort by not requiring an intermediate system description, there is still a substantial amount of effort required to build a comprehensive environment model for the system. Automating this effort is essential.

6.2.2.1 Reusable Environment Models

One way to simplify the task of environment model generation is to reuse the same environment model across many systems. This is possible if the system boundaries are defined along *well-defined* interfaces (§5.3). One such interface is the *libc* interface used by most user-space processes. One unfinished goal of this thesis is to build a CMC-compatible libc library using which user-space programs can simply be linked and model checked.

6.2.2.2 Automated Environment Refinement

Restrictions in the environment inputs can unnecessarily limit the behaviors explored and result in missed errors. Detecting such restrictions and relaxing them without causing unmanageable state space explosion is a challenging problem. This problem is similar to test input generation problem that occurs in conventional testing. One promising approach is to symbolically execute model checked paths and use the symbolic information to refine environment inputs.

6.2.3 Applying Static and Dynamic Analysis

Many static and dynamic analysis techniques are applicable to different aspects of CMC. Invariants about program variables inferred either statically or dynamically can be very useful in performing various state reduction techniques. For instance, dynamically generated dependencies between variables can be helpful in performing symmetry reduction [21] of arbitrary C code. Also, statically generated function summaries can be used as stubs for external functions during environment model generation.

6.2.4 Heuristic Search

Finally, CMC can be significantly improved with better search heuristics to effectively explore large state spaces with limited resources. Initial experiences with heuristic searches, however, are very frustrating. Despite enormous efforts, many “interesting” heuristics perform only marginally better than a simple random search of the state space. In the best case, search heuristics have achieved better coverage faster. But they have not uncovered any more errors than a simple depth-first or a purely random search. Determining the reason behind this can potentially lead to better heuristics. A promising approach is to use the search effectiveness metrics (§5.5.1) to heuristically improve the coverage achieved during model checking.

To conclude, the approach of model checking implementations is an effective

method to systematically test and find errors in large, complex systems. The results of this thesis clearly indicate the success of this approach. The challenges and the extensions discussed above provide interesting avenues for future research.

Bibliography

[1] Agilent. Router Tester. http://advanced.comms.agilent.com/routertester/.

[2] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, 2001.

[3] Thomas Ball, Mayur Naik, and Sriram Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL03*, pages 97–105, New Orleans, Louisiana, USA, January 2003.

[4] S. Begum, M. Sharma, Ahmed Helmy, and Sandeep K. S. Gupta. Systematic testing of protocol robustness: Case studies on mobile IP and MARS. In *LCN*, pages 369–380, 2000.

[5] D.J. Bernstein and E. Schenk. TCP SYN Cookies . http://cr.yp.to/syncookies.html.

[6] K. Bhargavan, D. Obradovic, and C. Gunter. Formal verification of standards for distance vector routing protocols, 1999.

[7] Edoardo Biagioni. A structured TCP in standard ML. In *SIGCOMM*, pages 36–45, 1994.

[8] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. In Rance Cleaveland and Hubert Garavel, editors, *Electronic Notes in Theoretical Computer Science*, volume 66. Elsevier, 2002.

[9] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. In *Software — Practice and Experience*, pages 18:807–820, 1988.

[10] Frederic Boussinot and Robert de Simone. The ESTEREL language. Technical report, INRIA Sophia-Antipolis, July 1991.

[11] R. Braden. Requirements for Internet hosts – Communication layers. RFC 1122, USC/Information Sciences Institute, October 1989.

[12] Lawrence S. Brakmo and Larry L. Peterson. Experiences with network simulation. In *Measurement and Modeling of Computer Systems*, pages 80–90, 1996.

[13] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2000.

[14] D. Bruening. Systematic testing of multithreaded java programs, 1999.

[15] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[16] S Chandra, B. Richards, and J.R. Larus. Teapot: a domain-specific language for writing cache coherence protocols. *IEEE Transactions on Software Engineering*, 25(3):317–33, May-June 1999.

[17] Sathish Chandra, Patrice Godefroid, and Christopher Palm. Software model checking in practice: An industrial case study. In *Proceedings of International Conference on Software Engineering (ICSE)*, 2002.

[18] Benjamin Chelf, Seth Hallem, , and Dawson Engler. How to write system-specific, static checkers in metal. In *Invited Paper. PASTE 2002*, 2002.

[19] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Form. Methods Syst. Des.*, 9(1-2):77–104, 1996.

[20] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[21] C.N. Ip and D.L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[22] Douglas Comer and John C. Lin. Probing TCP implementations. In *USENIX Summer*, pages 245–255, 1994.

[23] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *ICSE 2000*, 2000.

[24] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[25] C.Perkins, E. Royer, and S. Das. Ad-Hoc On-Demand Distance Vector (AODV) Routing. IETF Draft, http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-10.txt, January 2002.

[26] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Conference on Programming Language Design and Implementation*, 2002.

[27] Scott Dawson, Farnam Jahanian, and Todd Mitton. Experiments on six commercial TCP implementations using a software fault injection tool. *Software Practice and Experience*, 27(12):1385–1410, 1997.

[28] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking, 1998.

[29] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[30] F. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, August 1996.

[31] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, September 2000.

[32] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[33] D.R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, October 2000.

[34] Erik Nordstrom *et al.* AODV-UU Implementation . http://user.it.uu.se/˜henrikl/aodv/.

[35] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[36] M. R. Gary and D. S. Johnson. *Computers and Intractability*. Freeman, 1979.

[37] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 1997.

[38] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.

[39] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, May 2003.

[40] J. Hajek. Automatically verified data transfer protocols. In *Proceedings of the 4th ICCC*, pages 749–756, 1978.

[41] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[42] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, May 2002.

[43] Ruibing Hao, David Lee, Rakesh K. Sinha, and Dario Vlah. Testing IP routing protocols - from probabilistic algorithms to a software tool. In *FORTE*, pages 249–264, 2000.

[44] G. Holzmann. Algorithms for automated protocol validation, 1990.

[45] G. Holzmann and M. Smith. Software model checking: Extracting verification models from source code. In *Invited Paper. Proc. PSTV/FORTE99 Publ. Kluwer*, 1999.

[46] G. J. Holzmann. From code to models. pages 3–10, Newcastle upon Tyne, U.K., 2001.

[47] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[48] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[49] G.J. Holzmann. Formal methods for early fault detection. In *Proc. FTRTFT, Confs. on Formal Techniques for Real-Time and Fault Tolerant Systems*, volume 1135, pages 40–54, Uppsala, Sweden, September 1996.

[50] N.C. Hutchinson and L.L. Peterson. The x-kernel: an architecture for implementing network protocols. *IEEE Trans. on Soft. Eng.*, 17(1), January 1991.

[51] Radu Iosif. Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In *Proceedings of 16th IEEE Conference on Automated Software Engineering*, 2001.

[52] Ayal Itzkovitz and Assaf Schuster. Multiview and millipage - fine-grain sharing in page-based DSMs. In *Operating Systems Design and Implementation*, pages 215–228, 1999.

[53] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, CA, 1995.

[54] McMillan K. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[55] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[56] Luke Klein-Berndt and et.al. Kernel AODV Implementation . http://w3.antd.nist.gov/wctg/aodv_kernel/.

[57] Sanjeev Kumar. Using model checking to debug device firmware, December 2002.

[58] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[59] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. *Lecture Notes in Computer Science*, 2057:80–102, 2001.

[60] F. Lilieblad and et.al. Mad-hoc AODV Implementation. http://mad-hoc.flyinglinux.net/.

[61] H. Lundgren, D. Lundberg, J. Nielsen, E. Nordstrom, and C. Tschudin. A large-scale testbed for reproducible ad hoc protocol evaluations. In *IEEE Wireless Communications and Networking Conference*, March 2002.

[62] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgement options. RFC 2018, IETF, October 1996.

[63] S. McCanne and S. Floyd. UCB/LBNL/VINT network simulator - ns (version 2), April 1999. http://www.isi.edu/nsnam/ns/.

[64] K.L. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–51. Tokyo, Japan Inf. Process. Soc., 1991.

[65] Allen Brady Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, Todd A. Proebsting, and John H. Hartman. Scout: A communications-oriented operating system (abstract). In *Operating Systems Design and Implementation*, page 200, 1994.

[66] George C. Necula, Scott McPeak, S.P.Rahul, and Westley Wimer. Cil: Intermediate language and tools for analysis and transformation of c programs. *Proceedings of Conference on Compiler Constructions*, pages 213–228, March 2002.

[67] G. Nelson. *Techniques for program verification.* Available as Xerox PARC Research Report CSL-81-10, June, 1981, Stanford University, 1981.

[68] Vern Paxson. Automated packet trace analysis of TCP implementations. In *SIGCOMM*, pages 167–179, 1997.

[69] Vern Paxson and et.al. Known TCP implementation problems. RFC 2525, March 1999.

[70] J. Penix, W. Visser, E. Engstron, A. Larson, and N. Weininger. Verification of time partitioning in the DEOS real-time scheduling kernel. In *22nd International Conference on Software Engineering (ICSE)*, 2000.

[71] Charles E. Perkins, Elizabeth M. Royer, and Samir R. Das. Private Email Communication.

[72] J. Postel. Transmission Control Protocol. RFC 793, USC/Information Sciences Institute, September 1981.

[73] Rational Software. Purify: Fast detection of memory leaks and access errors. http://www.rational.com/products/whitepapers/319.jsp.

[74] A. Srivastava and A. Eustace. ATOM — a system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.

[75] U. Stern and D. L. Dill. A new scheme for memory-efficient probabilistic verification. In *IFIP TC6/WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, 1996.

[76] U. Stern and D.L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[77] W. Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. RFC 2001, IETF, January 1997.

[78] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of Usenix Annual Technical Conference*, June 2001.

[79] The plex86 x86 Virtual Machine Project. http://plex86.sourceforce.net/.

[80] The User-mode Linux Kernel. http://user-mode-linux.sourceforge.net/.

[81] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the International Conference on Applications and Theory of Petri Nets (ICATPN)*, volume 483, pages 491–515. Springer-Verlag Inc., 1989.

[82] C.H. West. General technique for communications protocol validation. *IBM Journal of Research and Development*, 22(4), 1978.