Model Update for
System Modifications

Yulin Ding

A thesis submitted for the degree of
Doctor of Philosophy at
University of Western Sydney

June 2007
Except where otherwise indicated, this thesis is my own original work. I certify that this thesis contains no material that has been submitted previously, in whole or in part, for the award of any other academic degree.

Y. Ding
29 June 2007
To my parents
I am deeply grateful to my principal supervisor, Associate Professor Yan Zhang. The research would not have happened without his great support and patience. He is not only a good supervisor who has foresight and vision for major directions of research, but also a modest research partner who always respects and values the ideas of a humble student. Thus, he gave me plenty of scope to use my own imagination which also trained me to be an independent researcher.

Thanks to my associate supervisor Dr. Dongmo Zhang for his valuable discussion and encouragement for my research. His commitment and honest attitudes to research influenced me to be a researcher with his invaluable qualities.

My gratitude also goes to my most trusted friend, senior software engineer Neville Cockburn. He is always there whenever I need help, both in life and study. Without his important guidance for the implementation of the research in this dissertation, the project would not be completed in its current form.

Thanks to University of Western Sydney for providing the priority scholarship for me to make my study possible.

I would also like to thank the academic staff and research colleagues in the Intelligent Systems Laboratory research group for their valuable inputs that contributed to my study.

I would like to thank all friends, particularly Mr. Lawrence Lamb, who have helped me both in life and study during the period of my Ph.D study.

Finally, I give thanks to the spirit of my scholarly family of many generations. Without the inspiration of my family tradition, I would not have considered the long journey of Ph.D study at the age when I should have focused on other agendas in life.
Abstract

Model checking is an existing approach for automatic reasoning. The model checker is an important tool and has been applied to software engineering for system verification. As an extension of model checking, model update is a new concept and has been defined and developed in this dissertation. A model updater is employed as an automatic system modification tool for software verification. The combination of model checking and model update completes the task of automatic system verification.

In this dissertation, a comprehensive theoretical foundation and prototype implementation for CTL model update are developed. First, five primitive updates which capture the basic atomic operations in a CTL Kripke model are identified and formalized. Next, the minimal change criteria for CTL model update based on these primitive operations are defined. Then, updating an original model to a new model satisfying CTL semantics based on previous primitive updates and minimal change rules is characterized. The related complexity of CTL model update is also analyzed. Following this research, formal algorithms in the form of pseudo code to be used as the guidance for future implementation are designed. Then, a prototype CTL model updater is implemented. This prototype CTL model updater is a stand alone utility and contains both model checking and update functions. The prototype is then applied to three well known models and has successfully been tested. During the implementation, as a byproduct, the approach for extracting a complete Kripke model from NuSMV is described. Also, during the implementation, an update model explosion problem is discovered and further minimal change rules based on the original minimal change rules are proposed which significantly reduce the number of minimally updated models. The corresponding theory is formalized and the complexity for solving the update model explosion problem is analyzed.
SUGGESTIONS for FUTURE DIRECTIONS

– Using different methods in combination:
  – Finding a suitable style for using different methods together;
  – Finding a suitable meaning for using different methods together.

In the future, we expect that the role of formal methods in the entire system development process will increase, especially as the tools and methods successful in one domain carry over to others.

– Integrated use methods and tools should work in conjunction with each other and with common programming languages and techniques. The use of tools for formal methods should be integrated with that of tools for traditional software development.

——— Edmund Clarke and Jeannette Wing in [17]
# Contents

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ix</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td></td>
</tr>
<tr>
<td>1.2 Background</td>
<td>4</td>
</tr>
<tr>
<td>1.2.1 Model Checking</td>
<td>4</td>
</tr>
<tr>
<td>1.2.2 Model Checker Design Philosophy</td>
<td>7</td>
</tr>
<tr>
<td>1.2.3 Development of Model Checking Systems</td>
<td>14</td>
</tr>
<tr>
<td>1.2.4 Applications of Model Checking</td>
<td>19</td>
</tr>
<tr>
<td>1.2.5 Error Diagnosis and Repair</td>
<td>21</td>
</tr>
<tr>
<td>1.2.6 Knowledge System and Knowledge Updates</td>
<td>25</td>
</tr>
<tr>
<td>1.3 Motivations</td>
<td>29</td>
</tr>
<tr>
<td>1.4 Contributions</td>
<td>30</td>
</tr>
<tr>
<td>1.4.1 Formal Framework of CTL Model Update</td>
<td>31</td>
</tr>
<tr>
<td>1.4.2 Implementation of the CTL Model Updater</td>
<td>32</td>
</tr>
<tr>
<td>1.4.3 Case Studies</td>
<td>33</td>
</tr>
<tr>
<td>1.5 Organization of the Dissertation</td>
<td>33</td>
</tr>
<tr>
<td><strong>2 Minimal Change for CTL Model Update</strong></td>
<td>35</td>
</tr>
<tr>
<td>2.1 CTL Syntax and Semantics</td>
<td>35</td>
</tr>
<tr>
<td>2.2 CTL Model Update</td>
<td>40</td>
</tr>
<tr>
<td>2.3 Primitive Operations</td>
<td>42</td>
</tr>
<tr>
<td>2.3.1 Examples for Combinations of Primitive Updates</td>
<td>43</td>
</tr>
<tr>
<td>2.4 Defining Minimal Change</td>
<td>45</td>
</tr>
</tbody>
</table>

xiii
## Contents

3 CTL Semantic Characterizations and Computational Properties 49  
3.1 Semantic Characterizations 49  
3.2 Computational Properties 58  

4 Algorithms for CTL Model Update 61  
4.1 Algorithms 61  
4.2 Updating Examples with the Algorithms 76  

5 The CTL Model Updater Prototype 79  
5.1 The Relationship between Model Checking and Model Update 79  
5.2 The Prototype of the CTL Model Updater 80  
5.3 Predefined Structures and Library Functions 81  
5.4 The Parser 84  
5.5 Model Checking Functions 86  
5.6 Model Update Functions 88  

6 Three Case Studies 91  
6.1 The First Model – the Microwave Oven 92  
6.1.1 Applying the Model Updater to the Microwave Oven Example 92  
6.1.2 The simulation Results for Updating the Microwave Oven Model 94  
6.2 Another Two Models – AFS0 and AFS1 96  
6.2.1 The AFS0 Model 98  
6.2.2 The AFS1 Model 99  
6.2.3 The Specification Properties 100  
6.2.4 Extracting the Kripke Models of AFS0 and AFS1 from NuSMV 101  
6.2.5 Applying the Model Updater to AFS0 and AFS1 104  

7 Study of a Model Explosion Problem 111  
7.1 An Admissible Model Explosion Problem 111  
7.1.1 Classifying Admissible Models by Reachable State Characteristics 112  
7.1.2 Analyzing Admissible Models 117  
7.2 Solving the Admissible Model Explosion Problem 121
## Contents

7.2.1 Minimal Change with Maximal Reachable States .... 121
7.2.2 The Reachable State Algorithm .... 122
7.3 Implementing the Reachable State Algorithm .... 124
7.3.1 Coding the Reachable State Algorithm .... 124
7.3.2 Simulation Results for Updating AFS0 and AFS1 .... 127
7.4 Characterizations and Complexities .... 133
7.4.1 Characterizations .... 133
7.4.2 Complexities .... 140

8 Conclusions and Future Research 143
8.1 CTL Model Update .... 143
8.1.1 Minimal Change .... 143
8.1.2 Characterizations and Complexities .... 144
8.1.3 Algorithms .... 144
8.1.4 The Prototype and Case studies .... 144
8.1.5 Study of the Model Explosion Problem .... 145
8.2 LTL Model Update .... 145
8.3 Limitations and Future Research .... 146
8.3.1 Theoretical Research .... 146
8.3.2 A Proposed SMV Model Updater .... 148

A LTL Model Update 151
A.1 LTL Model Checking: An Overview .... 151
A.2 LTL Syntax and Semantics .... 152
A.3 The LTL Kripke Model Update: Formal Definitions .... 153
A.4 The Principles of Minimal Change .... 157
A.5 Characterization .... 160

B Extracting Kripke Models from NuSMV 163

C ReadMe for the CTL Model Updater Prototype 175
C.1 Files .... 175
C.2 Building and Running the Prototype .... 177
D  Publications related to this Dissertation  179

Bibliography  181
List of Figures

1.1 Traditional Model Checking ................................. 5
1.2 The process of Model Checking ............................ 6
1.3 The Truth Table of function $f(x, y)$ ...................... 10
1.4 An Example of Binary Decision Tree where dotted lines are labelled
  0 and solid lines 1 .................................. 10
1.5 Model Checking with Knowledge Update ................... 31

2.1 A Transition State Graph .................................. 36
2.2 Unwinding the Transition State Graph as an Infinite Tree .... 36
2.3 $AG\phi$ and $EG\phi$ in an Unwound Tree .................. 40
2.4 The illustration of Model Update .......................... 41
2.5 The illustration of primitive updates PU2 and PU3 .......... 43
2.6 The illustration of combinations of primitive updates ........ 44
2.7 The illustration of minimal change rules .................. 47

5.1 The Model Checking and Model Update System ............ 80
5.2 The flow diagram of the Model Update System ............ 82
5.3 The state definition structure .............................. 82
5.4 The state structure in C code .............................. 82
5.5 The atom structure in C .................................. 84
5.6 The pair structure in C .................................. 84
5.7 The parsed structure of String “Start” .................... 85
5.8 The parsed structure of string “EG¬Heat” ................. 85
5.9 The parsed structure for string “AG(¬(Start \land EG¬Heat))” ..... 87
5.10 The parsed structure for string “¬Start \lor EG¬Heat” ...... 87

6.1 The Original CTL Kripke Structure of a Microwave Oven .... 94
6.2 The Updated Microwave Oven Model with Primitive Update PU2 94
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3</td>
<td>The Updated Microwave Oven Model with Primitive Update PU5</td>
<td>94</td>
</tr>
<tr>
<td>6.4</td>
<td>The Updated Microwave Oven Model with Primitive Update PU3</td>
<td>94</td>
</tr>
<tr>
<td>6.5</td>
<td>Finite State Diagrams for AFS0</td>
<td>98</td>
</tr>
<tr>
<td>6.6</td>
<td>State Transition Diagrams for AFS1</td>
<td>100</td>
</tr>
<tr>
<td>6.7</td>
<td>The extracted CTL Kripke structure of AFS0 from NuSMV</td>
<td>102</td>
</tr>
<tr>
<td>6.8</td>
<td>The extracted CTL Kripke structure of AFS1 from NuSMV</td>
<td>103</td>
</tr>
<tr>
<td>6.9</td>
<td>An Admissible Model of AFS0</td>
<td>109</td>
</tr>
<tr>
<td>6.10</td>
<td>Another Admissible Model of AFS0</td>
<td>109</td>
</tr>
<tr>
<td>6.11</td>
<td>The 3rd Admissible Model of AFS0</td>
<td>110</td>
</tr>
<tr>
<td>7.1</td>
<td>The sub model AFS1-1 of AFS1</td>
<td>113</td>
</tr>
<tr>
<td>7.2</td>
<td>The sub model AFS1-2 of AFS1</td>
<td>114</td>
</tr>
<tr>
<td>7.3</td>
<td>One of the Committed Models of AFS1</td>
<td>133</td>
</tr>
<tr>
<td>7.4</td>
<td>$s_1$ occurs in another valid path ${s_0, s_1', \cdots, s_k', s_1, s_{k+1}', \cdots}$</td>
<td>140</td>
</tr>
<tr>
<td>A.1</td>
<td>The relationship of a model checker and a model updater.</td>
<td>154</td>
</tr>
<tr>
<td>A.2</td>
<td>The illustration of the addition operation.</td>
<td>156</td>
</tr>
<tr>
<td>A.3</td>
<td>The illustration of the deletion operation.</td>
<td>156</td>
</tr>
<tr>
<td>A.4</td>
<td>The illustration of Definition A.6.</td>
<td>159</td>
</tr>
<tr>
<td>A.5</td>
<td>The illustration of minimal changed update for example 4 and Definition A.7</td>
<td>159</td>
</tr>
<tr>
<td>A.6</td>
<td>The original code for example 4.</td>
<td>159</td>
</tr>
<tr>
<td>A.7</td>
<td>The updated code for example 4.</td>
<td>159</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Introduction

Formal reasoning entails comparing two formal objects, e.g., establishing the correctness of a program with respect to a specification or showing that one concurrent process simulates another. There are two general approaches to show the correspondence between these two objects: theorem proving and model checking [86]. Theorem proving and model checking are two major fields in formal methods. Formal methods have already demonstrated success in specifying commercial and safety-critical software, and in verifying protocol standards and hardware designs [17]. As one of the most promising formal methods, automated verification has played an important role in computer science development [83]. Over the last decade, automated formal verification tools, such as model checkers, have shown their ability to provide a thorough analysis of reasonably complex designs [37].

Theorem proving is a technique where both the system and its desired properties are represented as formulas in some logic. The logic is given by a formal system which defines a set of axioms and set of inference rules. Theorem proving is the process of finding a proof of a property from the axioms of the system. Proofs can be constructed by hand or with machine assistance. In contrast to model checking, theorem proving can directly handle infinite state spaces. It relies on techniques such as structural induction to prove over infinite domains.

As one of the two major formal approaches for system verification, theorem proving has been used for many problem domains. In 1996, Clarke et al. used automatic theorem-proving techniques based on symbolic algebraic manipulation to prove the correctness of an SRT division algorithm similar to the one in Pentium [18].
Introduction

Moore and Kaufmann of Computational Logic, Inc., and Lynch of Advanced Micro Devices, Inc., collaborated to prove the correctness of Lynch microcode for floating point division on the AMD5K86 [52]. However, the major disadvantage of theorem proving is that in general, finding a proof is intractable and no limit can be placed on the amount of time or memory that may be needed in order to construct a proof [16].

Besides theorem proving, model checking is rather mature in both theoretical and practical research. Currently, model checkers with SMV or Promela [47] series as their specification languages are widely available for research, experiment, and partial industry usage. Nowadays SMV [16], NuSMV [12], Cadence SMV [60] and SPIN (Simple Promela Interpreter) [47] are well accepted as the state of the art model checkers. More recently, the MCK [38] model checker has enhanced currently in use model checkers by the inclusion of an added knowledge operator to check the knowledge of semantics of a Kripke model.

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. The checking is performed as an exhaustive state space search which is guaranteed to terminate because the model is finite. The major challenge in model checking is in devising algorithms and data structures that allow us to handle large search spaces. The technique of model checking has been successfully used in hardware and protocol verification and the analysis of software system specifications [18, 16].

The major advantage of model checking technique is that it is completely automatic and fast. For example, the model checking algorithm developed by Clark and Emerson [16] for the Computation Tree Logic (CTL) is polynomial in both the size of the model determined by the program and the length of its specification in temporal logic. The model checking complexity varies for different temporal logics. The major difficulty for model checking is to handle the state explosion problem. The approach of ordered binary decision diagrams (OBDDs) was proposed traditionally to represent state transition systems efficiently, thereby decreasing the size of the verified systems. Another traditional method to handle state explosion is called partial order reduction [16, 64], which is used to exploit the independence of concurrently executed events so that it becomes possible to reduce the num-
ber of states that are needed for model checking. More recently, new technologies have been developed such as embedding Boolean satisfiability (SAT) procedures into model checking algorithms as a complement of the traditional OBDD-based model checking [1, 5].

Model checking is a tool to report errors of a designed system. The next step is to repair the errors. In 1998, Stumptner and Wotawa presented an overview of Artificial Intelligence approaches to the development of intelligent debugging systems [79]. These systems range from tutoring systems to traditional debugging approaches. Additionally, the authors examine the more recent use of model-based diagnostic principles as a basis for software debugging research. The authors illustrated the potential of model-based reasoning by discussing several models, which are dependency-based, value-based, and repair-based representations. These three representations correspond to a move from abstract to detailed representation.

Recently, error diagnosis and repair are starting to employ a formal methods approach. Buccafurri et al. [8] applied Artificial Intelligence techniques to model checking and error repairing. They used abductive model revision techniques, repairing errors in concurrent programs. It is a new approach towards automated diagnosis and integrated repair on model checking. It aims at using techniques and concepts from model checking by combining them with AI principles. Harris and Ryan [43] also presented an attempt of system modification, which designed update operations to tackle feature integration performing theory change and belief revision.

The update of the knowledge base has been extensively studied. Winslett [87] first applied update to the knowledge base and used series update and minimal change methods for databases. Herzig and Rifi [45] conducted extensive surveys about updates of the knowledge base and their development before 1999. They listed ten update operations, based on propositional logic and further characteristics. Recently, the update of the knowledge base is enhanced by the modality update by Baral and Zhang [2]. They discussed knowledge update and its minimal change, based on modal logic $S5$.

We intend to apply knowledge update to practice and develop a general methodology to repair errors in a computer aided mode as the main contribution of our
research. We integrate the ideas of knowledge update and model checking together. Based on model checking techniques and knowledge update theories, we have developed minimal change rules for CTL model update. Based on CTL minimal change rules, we have characterized updates for each semantic of CTL, then, designed algorithms based on the characterizations. We have implemented a prototype model updater based on these theoretical foundations. The model updater prototype has been applied to three well known models for case studies. A model explosion problem was also studied during the implementation to further constrain the previous minimal change rules. The contribution in this dissertation is a foundation for future development of system modification. A more comprehensive update system employing a formal compiler can be built based on the research in this dissertation, where the integration of the model checker and the model updater is the major technical challenge.

1.2 Background

1.2.1 Model Checking

Model checking, which has the advantages of being automatic and often computationally less expensive as described in [50], was invented by E. Clarke and E. A. Emerson [21] and by J. Quielle and J. Sifakis [73]. A full history of these ideas can be found in [20], which also provides a survey of the topic.

Model checking is a technique for verifying finite state concurrent systems and the verification can be performed automatically [16]. Model checking is an automatic, model based, and property-verification approach. It is intended to be used for concurrent, reactive system and originated as a post-development methodology.

A typical case study of model checking is due to Wing and Vaziri-Farahani [86]. Fundamental to model checking is its reliance on finite state machines. Model checking exploits its finiteness property by performing an exhaustive case analysis of the machine’s set of states. For hardware, they are usually finite state machines. For software, they are, in general, infinite state machines. We thus rely on induction to prove in a finite number of steps of a property over an infinite domain by theorem
proving which works for highly structured components but fails at the system level. In software verification, model checking is exactly the kind of technology needed to handle the large case analysis at the system level, to find bugs (do formal reasoning) in designs of a natural abstraction (a model) of an actual working system.

![Figure 1.1: Traditional Model Checking](image)

A typical model checking procedure is shown in Figure 1.1 where the result of the model checking is to report errors. A more detailed model checking process can be described in Figure 1.2. The approach relies on finding abstraction mappings, \( A \), to apply to a hardware or software system (possibly an infinite state machine for software systems), \( S \), and then subjecting the abstract model (a finite machine), \( M \), to a model checker. We use a model checker as a black box to check \( M \) against the specification, \( \phi \). The model checker outputs either true, if \( M \) satisfies \( \phi \), or a counterexample, if it does not. One of the major specification languages for model checking is SMV. Another major specification language is Promela for the SPIN model checker.

The procedure normally uses an exhaustive search of the state space of the system to determine if some specification is true or not [16]. i.e., model checking is the process of computing an answer to the question of whether \( M, s \models \phi \) holds, where \( \phi \) is a formula of some logic, \( M \) an appropriate model, \( s \) a state of that model capturing values of variables at a particular instant of time in a system, and \( \models \) the underlying satisfaction relation. \( \phi \) is a formula of CTL, the model \( M \) is a representation of a “system”. In model checking, the models \( M \) are transition
systems and the properties $\phi$ are formulas in temporal logic. To verify that a system satisfies a property, three things must be done as follows:

(1). Model the system using the description language of a model checker, arriving at a model $M$.

(2). Coding the property using the specification language of the model checker, resulting in a temporal logic formula $\phi$.

(3). Run the model checker with inputs $M$ and $\phi$.

This process can be summarized as modeling, specification and verification. The model checker outputs the answer “yes” if $M, s \models \phi$ and “no” otherwise; in the latter case, most model checkers also produce a trace of system behavior which causes this failure. This automatic generation of such “counter traces” is an important tool in the design and debugging of systems. The counter traces are called counter examples in general model checkers.
1.2.2 Model Checker Design Philosophy

Under the model checking concept, a system which is checked by the model checker is represented in a Kripke model. Specification properties are represented as CTL formulas. These are the input of the model checker in the form of the SMV specification language. Then, the model checking algorithms conduct the checking task. During the checking procedure, counterexamples trace and report errors. Matches for each stage can be found in Figure 1.2. The intrinsic mechanisms of the model checker are as follows.

The input, a CTL Kripke model for a system, includes all subsets of the states of a system and the relationships between these states, and the properties $\phi$ of the system. The output is a group of subsets of states of the input states, which satisfy the input properties $\phi$. The input is usually in SMV or similar format. At the input, the Kripke model of a system actually represents a transition graph, where the subsets of the states can be represented with boolean values as shown in [50]. The boolean values and their transition relationships can be represented with boolean functions. The boolean functions are equivalent to a truth table. The truth table can be interpreted into a Binary Decision Diagram (BDD). BDDs can be optimized to a smaller size after leaves and nodes of BDDs are merged. Ordered Binary Decision Diagram (OBDDs) can be derived after the variables in BDDs are listed in order. The OBDDs represent boolean functions of the input system which can be calculated under boolean logic. In the real world, the boolean calculation relationships are often extracted from the “next states” of SMV. This way can avoid the gigantic size of truth tables and (O)BDDs, if the system is large and constructed with a large number of states. After the input transitions are represented in OBDDs, the model checking perform its task according to OBDDs which represent the input transition system. The major algorithm for model checking is “the labeling algorithm” implemented in function SAT(). The more detailed knowledge is as follows.

CTL and Kripke Models

CTL – Computation Tree Logic, is a temporal branching time logic, where its model of time is a tree-like structure. There are different paths in the future, and any one of
these might be the actual path that is realized. The semantics of CTL are defined in terms of Kripke structures and represent specifications properties at the input. The behavior of the underlying system is precisely represented with the Kripke structure at the input. The Kripke Model is a model \( \mu = (S, \rightarrow, L) \) for basic modal logic where there are a set of states \( S \) endowed with a transition relation \( \rightarrow \) (a binary relation on \( S \)), such that every \( s \in S \) has some \( s' \in S \) with \( s \rightarrow s' \), and a labelling function \( L : S \rightarrow P(\text{Atoms}) \). The CTL formulas are defined inductively via a Backus Naur form:

\[
\phi ::= \bot \mid \top \mid p \mid \neg \phi \mid (\phi \land \psi) \mid (\phi \lor \psi) \mid (\phi \rightarrow \psi) \mid \text{AX}\phi \mid \text{EX}\phi \\
\text{A}[\phi \psi] \mid \text{E}[\phi \psi] \mid \text{AG}\phi \mid \text{EG}\phi \mid \text{AF}\phi \mid \text{EF}\phi
\]

where path qualifiers are to form state formulas with atomic formulas \( \phi \) or path formulas: A (for All computation paths) and E (Exist some computation path), and temporal operators are to form path formulas with path formulas: X (next time), F (in the Future), G (Globally) and U (Until).

Other temporal logics which are related to CTL are Linear-time temporal logic (LTL), and CTL* [16, 50]. CTL and LTL are subsets of CTL*. Formulas of LTL have meanings on individual computation paths, i.e. there are no explicit paths quantifiers E and A. CTL* allows nested modalities and boolean connectives before applying the path quantifiers E and A.

Historically, LTL was invented first (by the philosopher A. Prior in the 1960s; and worked on by computer scientists A. Pnueli, Z. Manna and others). The logic CTL was invented by E. Clarke and E. A. Emerson (during the early 1980s); and CTL* was invented by E. A. Emerson and J. Halpern (in 1986) to unify the two [50].

**OBDDs – Ordered Binary Decision Diagrams**

SMV is the first model checker to use (O)BDDs. Originally, McMillan applied (O)BDDs to the SMV model checker to solve the state explosion problem [17]. In the model checker, (O)BDDs interpret binary functions, which represent sets of states instead of individual states to reduce the running time.

As described in [50], binary decision diagrams (BDDs) are a way of representing
boolean functions. A certain class of such diagrams provides the implementation framework for symbolic model-checking algorithms. BDDs were first considered in a simpler form called binary decision trees whose non-terminal nodes are labelled with boolean variables such as $x$, $y$, $z$, $\cdots$ and whose terminal nodes are labelled with either 0 or 1. Each non-terminal node has two edges each with a 0 or 1 value. Each finite binary decision tree determines a unique boolean function of the variables in non-terminal nodes. For example, function $f(x, y) \overset{\text{def}}{=} x + y$ is interpreted into $1 = \neg(x \lor y) = \neg x \land \neg y$. The corresponding truth table is shown in Figure 1.3, where $f = 1$ when both $x$ and $y$ are 0. The corresponding binary decision tree is shown in Figure 1.4. In this figure, there are two layers of variables $x$ and $y$; all left edges (dashed lines) represent variables being 0 and all right edges (solid lines) represent variables being 1. The terminals are labelled with either 0 or 1. The root is the node with variable $x$. It is also fine if two variables $x$ and $y$ swap layers. We start at the root and take the dashed line whenever the value of the variable at the current node is 0; otherwise, we travel along the solid line. The function value is the value of the terminal node we reach. For this function, there are 4 values with all possible values of $x$ and $y$ as listed in the truth table: $f(0, 0) = 1, f(0, 1) = 0, f(1, 0) = 0$ and $f(1, 1) = 0$. Now, we can see the complete matches among the binary decision tree, the truth table and the function. BDDs can be optimized to a smaller size by removal of duplicate terminals, non-terminals and redundant tests (merging nodes). For example, in Figure 1.4, the three terminals with value 0 can be merged into one terminal with value 0. The node on the right side with variable $y$ and its incoming and outgoing paths can be replaced with a path from the root node with variable $x$ to the terminal with value 0. There are algorithms available for optimization [50].

An ordered BDD (OBDD) is a BDD with an ordering on the variables occurring along any path. The reason to have OBDDs is to compare whether two BDDs before or after optimization are the same or not. The reduced OBDD representing a given function $f$ is unique. That is to say, let $B_1$ and $B_2$ be two reduced OBDDs with compatible variable ordering. If $B_1$ and $B_2$ represent the same boolean function, then they have identical structure.
**Figure 1.3:** The Truth Table of function \( f(x, y) \)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 1.4:** An Example of Binary Decision Tree where dotted lines are labelled 0 and solid lines 1

**The Model Checking Algorithm**

The labelling algorithm performs the model checking duties. The labelling algorithm is mainly implemented in the function \( \text{SAT}(\phi) \) [50], which determines the set of states satisfying \( \phi \). Recursive calls are performed in the pseudo-code of the \( \text{SAT} \) function, which is listed below:

```plaintext
function \( \text{SAT}(\phi) \) /* determines the set of states satisfying \( \phi \) */
begin
  case
    \( \phi \) is \( \top \): return \( S \)
    \( \phi \) is \( \bot \): return \( \emptyset \)
    \( \phi \) is atomic: return \( \{ s \in S | \phi \in L(s) \} \)
    \( \phi \) is \( \neg \phi_1 \): return \( S - \text{SAT}(\phi_1) \)
    \( \phi \) is \( \phi_1 \land \phi_2 \): return \( \text{SAT}(\phi_1) \land \text{SAT}(\phi_2) \)
    \( \phi \) is \( \phi_1 \lor \phi_2 \): return \( \text{SAT}(\phi_1) \cup \text{SAT}(\phi_2) \)
    \( \phi \) is \( \phi_1 \rightarrow \phi_2 \): return \( \text{SAT}(\neg \phi_1 \lor \phi_2) \)
    \( \phi \) is \( AX\phi_1 \): return \( \text{SAT}(\neg EX\neg \phi_1) \)
    \( \phi \) is \( EX\phi_1 \): return \( \text{SAT}_{EX}(\phi_1) \)
    \( \phi \) is \( A(\phi_1 U \phi_2) \): return \( \text{SAT}(\neg (E[\neg \phi_2 U (\neg \phi_1 \land \neg \phi_2)] \lor EG\neg \phi_2)) \)
    \( \phi \) is \( E(\phi_1 U \phi_2) \): return \( \text{SAT}_{EU}(\phi_1, \phi_2) \)
end
```
The function \( \text{SA T} \) takes a CTL formula as input and returns the set of states satisfying the formula. It calls the functions \( \text{SA T}_\text{EX}(\phi) \), \( \text{SA T}_\text{AF}(\phi) \) and \( \text{SA T}_\text{EU}(\phi) \) respectively, if EX, AF or EU is the root of the input’s parse tree. Fewer CTL operators than those in Backus Naur form are implemented in the SAT function because of the following theorem [50]: the set of operators \( \perp \), \( \neg \) and \( \land \) together with EX, AF and EU are adequate for CTL: any CTL formula can be transformed into a semantically equivalent CTL formula which uses only those logical connectives. The function \( \text{SAT}_\text{EX} \) is shown in the following as an example of subfunctions of SAT:

function \( \text{SAT}_\text{EX}(\phi) \) /* determines the set of states satisfying \( \text{EX}\phi \) */

local var \( X, Y \)

begin
\begin{align*}
X & := \text{SAT}(\phi); \\
Y & := \{ s_0 \in S | s_0 \rightarrow s_1 \text{ for some } s_1 \in X \}; \\
\text{return } & Y
\end{align*}

end

The function \( \text{SAT}_\text{EX} \) computes the states satisfying \( \phi \) by calling SAT. Then, it looks backwards along \( \rightarrow \) to find the states satisfying \( \text{EX}\phi \).

When the labelling algorithm performs the checking, the counterexamples report the trace of all errors. The satisfied subsets of states of the system are output from the model checker.
The Principle of Counterexamples

One of the most important features of CTL model checking algorithms is the ability to find counterexamples and witnesses. Counterexamples are introduced in [13, 16, 50]. The counterexample system is a special case of a transition system and traces certain types of errors then reports errors during model checking. A disproving rule is applied to counterexamples. When the model checker determines that a formula with a universal path quantifier is false, it will find a computation path which demonstrates that the negation of the formula is true. Likewise, when the model checker determines that a formula with an existential path quantifier is true, it will find a computation path that demonstrates why the formula is true. For example, if the model checker discovers that the formula $AG\phi$ is false, it will produce a path to a state in which $\neg\phi$ holds. Similarly, if it discovers that the formula $EF\phi$ is true, it will produce a path to a state in which $\phi$ holds. The counterexample for a universally quantified formula is the witness for the dual existentially quantified formula. The method to find witnesses is described in [16].

SMV – Symbolic Model Verifier

SMV is introduced in [16, 50, 61, 85]. SMV provides a language for describing models and it directly checks the validity of CTL formulas on those models. SMV takes as input a text consisting of a program describing a model and some specifications (CTL formulas). It produces as output either the word “true” if the specifications hold for all initial states, or a trace showing why the specification is false for the model determined by the program. SMV is used as the input specification language for the model checkers SMV, NuSMV and Cadence SMV. SMV system specifications consist of one or more modules. One of the modules must be called main. Modules can declare variables (“VAR” in SMV code) and assign value (“ASSIGN” in SMV code) to them. Assignments usually give the initial value of a variable and its next value as an expression in terms of the current values of variables. Specifications (“SPEC” in SMV code) provide as input the specification properties to be checked. The example of SMV code as input text of a model checker is as follows:
MODULE main
VAR
  request : boolean;
  status : {ready, busy};
ASSIGN
  init(status) : = ready;
  next(status) : = case /*Non-determinism */
    request : busy;
    1 : ready, busy;
  esac;
SPEC
  AG(request − > AF status = busy) /*represent properties φ*/

Model Checkers

Currently, the state of the art model checkers are SMV [16], NuSMV [12] and Cadence SMV [60, 62], which employ SMV specification language for both CTL and LTL model checking, and SPIN [47], which uses Promela specification language for on the fly LTL model checking. SMV is the earliest CTL model checker written by K. McMillan at Carnegie Mellon University and completed as his Ph.D task in 1992. After that, NuSMV and Cadence SMV were developed. NuSMV is a reimplementation of the SMV model checker and supports more functions than those of the original SMV. It is aimed at being customizable and extensible. It was developed by A. Cimatti, E. Clarke, F. Giunchiglia and M. Roveri in Trento, Italy. Cadence SMV is an entirely new model checker focused on compositional systems. It provides a way to verify an infinite system using a compositional model and was developed by K. McMillan as well at Cadence Berkeley Labs [62]. The counterexample free elaborate is introduced in Cadence SMV [60]. SPIN was developed at Bell Labs starting in 1980 and the software has been available since 1991. SPIN is used to verify correctness requirements for systems of concurrently executing processes [47]. In addition to SMV and SPIN, some other tools supporting model checking are MCK [38], FDR, COSPAN, the Concurrency Workbench, Murϕ, Mec and EXP as
in paper [80].

1.2.3 Development of Model Checking Systems

Abstraction

Abstraction is to extract a system into a mathematic model. We abstract away variables in the model which are not relevant to the formula being checked [50]. Within the scope of model checking, the CTL Kripke model is an abstract model of a real system. The specification property is an abstraction of a required property. The skill of abstraction is very important for an effective and correct model checking process.

Wing and Vaziri-Farahani [86] focus on improving abstractions to improve model checking feasibilities and also gave future directions to improve abstractions. They also pointed out that their method in this paper lacks a formal justification of their abstractions. In [6], Boyer and Sighireanu tried abstracting models as well as developing their model checking application. They obtained a relation by carefully modeling and analyzing the protocol. They designed a formal model for the PGM protocol and the synthesis of the constraint between its parameters. In [11], Cheung tried abstracting the property of what a system does rather than formalize all information in a system. Most formal methods techniques require mathematical training or theorem proving skills which become the barrier for practical use. However, Cheung introduced tabular notation to break this barrier. Cheung also verified the elevator specification with XSPIN. Lawesson et al. in [57] proposed a fault isolation scheme based on model-checking in order to reason about temporal properties of loosely coupled systems of concurrent processes. To address the problem of state space explosion, they advocate an automatic abstraction technique based on a notion of observational equivalence. They statically analyze a system and construct a total function from possible message logs to isolated faults. They reduced fault isolation to table lookup. Tables can be used at design time to find non-diagnosable failures of the system as well as redundant error messages. This paper tells us that improving abstraction skills can solve system problems which are state explosion problems under some circumstances. Felfernig et al. in [35] showed how the usage
of hierarchical abstractions can reduce the computation times for the explanations and in addition gives the possibility to iteratively and interactively refine diagnoses from abstract to more detailed levels. This paper states that improving abstraction skills can improve error diagnosis efficiency.

Refinement is the reverse of abstraction; it is the process of taking one level of specification (or implementation) and through a series of “correctness-preserving transformations” synthesizing a lower-level specification (or implementation) [17]. Refinement is usually employed to produce more concrete versions of a specification, or to add new requirements to it.

In [39], Gorogiannis and Ryan applied the notion of minimal refinements, to avoid over-refinement, and proved that this definition is well-behaved theoretically as well as computationally. In [59], Lomuscio and Ryan argued that model refinement could not be defined satisfactorily on Kripke models, and proposed a definition on Kripke trees obtained from Kripke models instead. The shift from Kripke models to Kripke trees achieves two main results. First, it was shown to be possible to refine trees by a formula expressing knowledge without affecting the knowledge of the other agents. This was not apparently possible on standard Kripke models. Secondly, while it seems impossible to obtain commutativity for even safe formulas on Kripke models, it was shown this is possible for Kripke trees. In [76], Ryan and Schobbens presented a notion of refinement between agent-oriented systems defined using alternating-time temporal logic (ATL). The refinement relation provided a framework for defining roles in a society of interacting agents, and formalizing a relation of conformance between agents and roles. The refinement relation also allows us to construct abstractions in order to make verification more tractable. Chauhan et al. in [9] gave two algorithms which monitor the SAT checking phase in order to analyze the impact of individual variables. It is a paper with a SAT based automatic refinement framework for model checking systems with several thousand state variables in the scope of influence of the specification.
Algorithms for Model Checking

The research in this area mainly involves the model checking algorithm and solving the state explosion problem. These algorithms are the intrinsic principles of a model checker and are embedded inside the model checker compiler to conduct the model checking task. Model checking has been held back by the state explosion problem, which is the problem that the number of states in a system grows exponentially according to the number of system components. The state explosion problem decreases the efficiency of model checking significantly. The idea of reducing the state space by selecting only a subset of the ways one can interleave independently executed transitions has been studied by many researchers [16]. Solving the state explosion problem assists the optimization of model checking. Algorithms for solving state explosion problems were initiated by McMillan in [64] and have been further developed by others. Recently, model checking algorithms based on Boolean satisfiability (SAT) procedures have been developed to complement the traditional BDD-based model checking [1]. Clarke et al. [15] gave a complete summary of the development of bounded model checking, which is model checking with satisfiability solving, before 2001. In [1], Amla et al. classified model checking algorithms into three categories: (1) bounded model checking which is useful for finding failures, (2) hybrid algorithms that combine SAT and BDD based methods for unbounded model checking, and (3) purely SAT-based unbounded model checking algorithms. The paper provided a uniform and comprehensive basis for evaluating these algorithms. The paper describes these algorithms in relation to a large and diverse set of hardware benchmarks. More examples of research in this field are as follows.

In [51], Kang and Park described a boolean satisfiability checking (SAT)-based unbounded symbolic model-checking algorithm. The conjunctive normal form was used to represent sets of states and transition relation. Their experimental results show that the proposed algorithm can verify more benchmark circuits than the previous BDD-based and SAT-based UMC algorithms. The proposed algorithm can prove more false safety properties than McMillan’s SAT-based algorithm and the BDD-based algorithms due to the efficient techniques proposed in this paper such as caching, two-level logic minimization, and so on. In [23], Demri et al. investigated
the complexity of a wide variety of model checking problems for non-flat systems under the light of parameterized complexity, taking the number of synchronized components as a parameter. Baier [3] introduced a symbolic model checking procedure for Probabilistic Computation Tree Logic (PCTL) over labelled Markov chains as models [3]. Their paper introduced labelled Markov chains, PCTL and Multi-terminal binary decision diagrams (MTBDD). MTBDDs represent labelled Markov chains. Then it builds up a model checking algorithm for PCTL which is based on established BDD techniques to represent formulas. Model checking for probabilistic logic typically involves solving linear equation systems in order to ascertain the probability of a given formula holding in a state. The algorithm in this paper is based on the idea of representing the metrics used in the linear equation system by MTBDDs in Verus, a language for specifying verified real time systems. The authors made the following contribution: procedure is efficient because it avoids explicit state space construction. The main difficulty with current probabilistic model checking is the need to integrate a linear algebra package with a conventional model checker. The symbolic model checking procedure avoids the explicit construction of the state space. In [42], The DiVer verification platform uses abstraction and BDDs to complement bounded model checking in the quest for completeness. In [4], Barnat et al. proposed a practical parallel on-the-fly algorithm for enumerative LTL model-checking. This algorithm can alleviate the state explosion problem by distribution of the state space. The algorithm is based on a combination of the breadth-first search state space generation with a back-level controlled depth-first search for accepting cycles. Korf in [53] presented that delaying the detection of duplicate nodes in a breadth-first search can effectively make use of very large capacity disk-storage systems. The solution of the author is rather than checking each newly-generated node against a table of stored nodes as soon as it is generated, he appended each node to a disk file containing previously generated nodes. Then, he sorted the file of nodes by their state representation, thereby bringing nodes representing the same state together in the file. Finally, he scanned the sorted list of nodes in one pass, merging the duplicate nodes. This paper gives a search algorithm for the model checker to sort the needed states. Lafuente solved the state explosion problem in
his paper [56] where the author exploited symmetries and applied heuristic search to mitigate the state explosion problem of automated verification. The first method reduced the state space to be explored to an equivalent smaller one, while the second technique guided the search in the direction of errors. The author also stressed that finding optimal (short) counterexamples is an important question, specially for system designers and debuggers. If symmetry reduction is applied, the shortest path to the error states in the reduced and the original state space have the same length. The search algorithm with symmetry reduction guarantees optimal counterexamples only if combined with admissible heuristics. The FSM distance is an admissible estimate that helps to find the optimal path to a given error state. Nevertheless, when applying symmetry reduction one has to aim at finding the orbit of that state.

Counterexamples

Counterexamples have similar complex mechanisms as model checking. Shankar and Sorea in [78] introduced recent development and common knowledge of counterexamples. A number of research efforts are about the structures and logics of counterexamples. Some of these also involve the error diagnosis technique because the counterexample itself is an error diagnoser. We list a few research papers about the mechanism and structures of counterexamples below. Error diagnosis papers are listed in section 1.2.5.

Clarke et al. in [14] introduced a new general framework for counterexamples. The paper has three main contributions: (1). They determined the general form of ACTL counterexamples. To this end, they investigated the notion of counterexamples and showed that a large class of temporal logic beyond ACTL admits counterexamples with a simple tree-like transition relation. The existence of tree-like counterexamples is related to a universal fragment of extended branching time logic based on $\omega$-regular temporal operators. (2). They presented new symbolic algorithms to generate tree-like counterexamples for ACTL specifications. (3). Based on tree-like counterexamples they extended the abstraction refinement methodology developed recently by Clarke et al. in another paper to full ACTL. This demonstrated the conceptual simplicity and elegance of tree-like counterexamples. Clarke
et al. in [19] introduced a new technique which produced a counterexample execution trace that showed the cause of the problem if the specification was not satisfied. They described an efficient algorithm to produce counterexamples and witnesses for symbolic model checking algorithms. Clarke in [13] also described an abstraction refinement skill related to counterexamples. Buccafurri and his colleagues in [7] gave a thorough complexity study of linear counterexamples generated by ACTL. They drew important conclusions from their analysis: there exists no simple characterization of the ACTL formulas that guarantee linear counterexamples. They identified the maximal set LIN of ACTL formula templates whose instances (obtained by replacing atoms with arbitrary pure state formulas) always guarantee linear counterexamples. They also showed that a single path of Kripke model witnessing the failure by itself can be computed in polynomial time.

1.2.4 Applications of Model Checking

The application of model checking to systems proves the correctness of model checking and the effectiveness of the model checker. Research in this area includes using the model checker to solve real time system problems. It also includes the modification of features or specification languages of the model checker to make the model checker suitable for specific systems.

Wing and Vaziri-Farahani [86] used an approach to apply model checking to software which hinges on identifying appropriate abstractions that exploit the nature of both the system, $S$, and the property, $\phi$, to be verified. They checked $\phi$ on an abstracted, but finite, model of $S$. Following this approach the authors verified three cache coherence protocols used in distributed file systems. The tool for model checking is the SMV model checker. This was the earliest application of the model checker to software protocol models. Before this, the model checker was only applied to hardware models. Krishnan and Hartley in paper [54] used model checking technology to test concurrent programs. Chen et al. [10] presented a formal verification methodology and case studies for property verification of designs represented at different abstraction levels. The used model checker in this paper is SPIN and its Promela specification language for LTL model checking.
Some researchers tried to use alternative model checking methods to improve the usage of the model checker. The related research is as follows.

In [70], Plath and Ryan extended the SMV language with a new construct for describing features. They built a tool called SFI (“SMV Feature Integrator”) which compiles descriptions in this extended language into pure SMV, ready for verification by the SMV model checker. The research is the combination of feature integration and model checking. The experiences of the authors with SMV as the underlying language were mixed. While SMV’s compact language made it easy to define the feature construct and feature integration and the feature construct proved easy to use, they found it not expressive enough for some purposes. Especially, the restrictions on the usage of arrays and the lack of primitives for synchronization and communication were cumbersome. On the other hand, the simplicity of SMV resulted in a small and simple feature construct, much simpler than their proposed feature construct for Promela [72]. Choosing SMV also meant that they were committed to the verification of properties stated in a logic, rather than testing processes for bisimulation or refinement, or checking generic properties such as deadlock-freedom.

In [71], Plath and Ryan used a combination of two techniques: static (syntactic) analysis and model checking using the model checker FDR which is introduced in [74]. In [6], Boyer and Sighireanu proposed a methodology using finite and real-time model-checking to deal with the synthesis problem on a parameterized system. This work gives some ideas about how the existing finite verification tools can be used to deal with parameterized verification. In [31], Edelkamp et al. contributed to the application of trail-based heuristics in the field of Java program verification. The authors introduced two such heuristics and combined them with the existing heuristics of JPF to obtain the shortest possible error trail. As the experimental results suggest, the use of trail-based heuristics can lead to a significant reduction of the error trail for all three tested programs. It is a method to model check code.

Model checking is also applied to knowledge update. Hoek and Wooldridge in [46] addressed the model checking problem for temporal logic of knowledge. They introduced a temporal logic of knowledge and developed an approach to $CKL_n$ model checking that combines ideas from the interpreted systems semantics for knowledge
with the logic of local propositions developed by Engelhardt et al.. With their approach, local propositions provide a means to reduce $CKL_n$ model checking to linear temporal logic model checking. Van der Meyden and Shilov in [65] studied model checking for the modal logic of knowledge and linear time in distributed systems with perfect recall.

### 1.2.5 Error Diagnosis and Repair

Before model checking became standard and well accepted in the early 1990’s, most error diagnosis and repair were localized and detailed methods oriented. These methods targeted detailed and special specification, program debugging, and are domain dependant technical methods. The error diagnosis mainly described static states. All previous attempts to validate it were based entirely on informal techniques [17].

After model checking became more standard and well accepted, the error diagnosis has been becoming more and more formalized as well. Model checking describes dynamic states. While the model checker itself diagnoses errors, other particular error diagnosis methods also use model checking theory. Thus, error diagnosis has progressed significantly in recent years and become more and more standard. Error diagnosing has been improved in theory to achieve a more general approach.

In contrast to error diagnosis, error repair methods have developed more slowly than error diagnosis. Most methods in this field are domain dependent for a long period of time and remain at a similar stage as error diagnosis was before the early 1990s. In recent years, more and more researchers have tried formalizing methods by making use of model checking to improve error repairing. It is promising that error repairing is gradually employing a formal methods approach.

The concept of model-based reasoning consists of providing a model that describes a certain domain and then using it to diagnose different systems from that domain [79]. Intelligent debugging systems are categorized in this domain. These systems range from tutoring systems that possess detailed knowledge about the individual programs as well as about typical programmer errors occurring specifically in these programs to traditional debugging approaches such as algorithmic debugging.

The concept of model based diagnosis originates in research intended to improve
on rule based diagnosis systems. Initially, the approach of model based diagnosis has been used in the search for hardware faults [82]. First attempts to extend the paradigm towards the search for errors in software programs were made by Console et al. [22]. The idea was to take a logic program and remove, add, or replace whole clauses with the goal of arriving at a program that would produce correct answers to certain test queries. Stumptner and Wotawa further extended this method by proposing a diagnosis framework on functional and other programs [80]. They presented two different approaches for diagnosing faults that result in incorrect executions. The first (simpler and quicker) approach uses a fixed system description based on the evaluation trace to pick out errors (Evaluation traces). The second (more accurate) approach incorporates the search for a repair that will correct the fault, expressed in terms of fault modes corresponding to possible repairs (Expression Replacements) where REPLACE-SEM algorithm is used to diagnose the errors in the program and correct the program by using the minimum replacement method. This paper mainly used EXP logic language to do local program replacement. It is an abductive style model checking. The same authors in [82] described the VHDL-DIAG tool which provides design support by using model based reasoning for determining the source of error in hardware designs that are written in the VHDL specification language. Friedrich, and the same authors [36] also described an approach to employ model-based diagnosis for fault detection and localization in very large VHDL programs.

Furthermore, in 1998, Stumptner and Wotawa had a fairly complete survey in [79] where the methods in [81] and [22], which is about model-based diagnosis, are also included. The recent history of debugging started with intelligent tutoring systems that can operate on a detailed understanding of the purpose of the programs to be diagnosed (with PROUST and Talus systems). The next group was presented by functional approaches (ZD and DUDU) which also attempt to provide a detailed additional specification at multiple levels of a program that was intended to show how smaller functions interact to produce the total function of the system as a whole. Then, in the probabilistic diagnosis of the DAACS system, this individual specification was replaced by expert analysis of example cases for specific categories
of runtime errors. Additional focusing was provided by logical analysis of execution paths that remove parts of the program from consideration as possible sources of the error. Ultimately, for the model-based representations (e.g., VHDL representation), they share some basic characteristics: the system description was expected to specify the semantics of the program when executed, albeit at an abstract level, and the provided observations specify the expected output values. Also, none of these approaches require a formal specification external to the program. The dependency-based [81], value-based and repair-based representations correspond to a move from abstract to detailed representation. Other research in this field is as follows.

In paper [55], Kuehlmann et al. described a diagnosis technique for locating design errors in circuit implementations which do not match their functional specification. The method efficiently propagates mismatched patterns from erroneous outputs backward into the network and calculates circuit regions which most likely contain the error(s). In contrast to previous approaches, the described technique does not depend on a fixed set of error models. Therefore, it is more general. The proposed method is also applicable for incomplete sets of mismatched patterns and hence can be used not only as a debugging aid for formal verification techniques but also for simulation-based approaches. In paper [49], Huang et al. presented a fault simulation-based approach for diagnosing the general design errors in a sequential circuit. Their approach allows the implementation to have a different number of flip-flops or a different state encoding from the specification. This capability is particularly useful when the specification is given at a higher level of abstraction. Their approach does not rely on any error model, and thus, is suitable for general types of errors. It can also be easily expended to identify multiple errors. This research could be an important step towards sequential error diagnosis. In paper [68], Nayak and Walker described an approach to design error diagnosis and correction in combinatorial digital circuits. This approach targets small errors introduced during the design process or due to specification changes. They incrementally used simulation to identify suspect nets, and then attempted correction based on their error model. They used multiple iterations to handle multiple errors. In paper [58], by using the constraint-based modeling approach, Le and Menzel have developed a diagnostic
component, which is able to identify errors made by learners of a logic programming language when implementing a given task specification. It uses patterns to hypothesize the intention of a learner and programming techniques to model conditions on the semantic well-formedness of the program code.

The next stage after the above, error diagnosis started to enter a formal methods based approach. Some researches have tried to improve error repair with formal methods to integrate model checking and automatic modification.

Buccafurri et al. [8] have proposed an approach by combining some AI techniques into model checking such that the enhanced model checking algorithm is not only able to identify errors for a concurrent system, but also able to provide possible modifications for the system. They also exploited counterexamples to identify corrections under the circumstance when the counterexample is invariant. This paper is a new approach towards automated diagnosis, and integration of repair into model checking and aims at using techniques and concepts from model checking and combining them with AI principles.

Other research efforts for error diagnosis related to formal methods are summarized as follows. Leino et al. in [75] said that a technique for finding errors in computer programs is to translate a given program and its correctness criteria into a logical formula in mathematics and then let an automatic theorem prover check the validity of the formula. They described a method for reconstructing, from the theorem prover’s mathematical output, error traces that lead to the program errors that the theorem prover discovers. In this paper, the formal methods approach is theory proving. Groce and Visser in [41] used an automated method for finding multiple versions of an error, and analyze these executions to produce a more succinct description of the key elements of the error. Harris and Ryan [43] recently proposed another attempt at system modification. They designed update operations to tackle feature integration performing theory change and belief revision. Huang and et al. in [48] studied diagnosis of discrete event systems (DESs) modelled in rule-based modelling formalism to model failure-prone systems. They developed symbolic techniques for testing diagnosability and computing a diagnoser. Diagnosability test is shown to be an instance of first order temporal logic model-checking. Felfernig et al. [34] de-
1.2 Background

developed a framework suitable for diagnosing configuration knowledge bases based on the formal definition of consistency-based configuration. They started from a clear definition of diagnosis in the configuration domain then developed an algorithm based on conflict. Felfernig and Shchekotykhin [32] presented an approach to automated debugging of faulty process designs of knowledge-based recommenders which increases the productivity of user interface development and maintenance. They used automata to represent recommender interfaces to formalize the system. Felfernig [33] discussed a formal model for defining the intended behavior of recommender user interfaces and showed the application of model based diagnosis concepts which allow the automated debugging of those definitions. Dennis et al. [24] intended to clarify the mechanisms by which failed verification attempts can be used to isolate errors in code, in particular by exploiting the ways in which the branching structure of a proof can match the structure of the functional program being verified. They also found that the automation of proof-directed debugging with proof planning allows code patches to be synthesized at the same time that a bug is located and diagnosed.

1.2.6 Knowledge System and Knowledge Updates

The classical explanation about update is given by Grahne in [40]. Semantically, we change each of the possible worlds “as little as possible” in order to make the new state of affairs hold. Our new syntactic description of the worlds of interest should now correctly reflect the outcome of this set of changes. The function that maps the old description to the new is called an update. Harris and Ryan in [44] said, “Update is an operation of theory change which is closely related to belief revision. The principal difference lies in the fact that belief revision models changing beliefs about a static world whereas update models a changing world”. Belief revision is the change on a set of static formulae (belief set) which belongs to an agent, whereas belief update is the change on a model, which represents the real changing world (or system). Knowledge update is a defined notion in [2], which is analogous to belief update, where the original theory and the new theory are in a language that can express knowledge, and changes are allowed in both the real world and the agent’s
knowledge about the world.

Updates of the knowledge base and knowledge have been extensively studied. Most research in this field is theoretical. The early knowledge based update was mainly based on propositional logic update. Recent knowledge update has been developed and has started to be based on modality logic (Agent), as presented in papers [2] and [77].

Winslett [87] was a pioneer of the update of the knowledge base and used series update and minimal change methods for databases. Hezig and Rifi [45] conducted extensive surveys about updates of the knowledge base and their development before 1999. In [45], the authors listed ten concrete propositional update operations from the literature. Before 1990, the semantics of model based update were more basic. They are: WSS: Minimal change with exceptions, which was proposed by Winslett in 1988; WSS↓: Making WSS syntax-insensitive, which was first proposed by Hegner in 1987, was then simplified by Herzig and Rifi in 1998 and another similar operation called MPMA (Modified PMA) was proposed by Doherty et al. in 1998; PMA (Possible Models Approach): Minimal change, which was proposed by Winslett in 1988; FORBUS: Numeric minimal change, which was proposed by Forbus in 1989 and is also called “M-C with Local CC” by Winslett. After 1990, the semantics of model based update involved more advanced topics. The update operations are: MCD (Minimal Change with Maximal Disjunctive inclusion): Going beyond the PMA, which was proposed by Zhang and Foo in 1996; MCD*: Iterating MCD, which was proposed by Herzig and Rifi as a correction of MCD in 1998; MCE: Making WSS↓ conservative, which was introduced by Zhang and Foo in 1996; WSS^{dep} and WSS↓^{dep}: Enhancing WSS by dependence information, which was introduced by Herzig in 1996; MPMA**: enhancing WSS↓ by causality information, which was introduced by Doherty et al. in 1998.

Most update operations involve calculating distance. If \( p \) is an atom, the distance between \( w \) and \( v \) is the set of atoms whose truth value differs [45]:

\[
DIST(w, v) = (w \setminus v) \cup (v \setminus w) = \{ p : w \in [[p]] \text{ and } v \notin [[p]] \} \cup \{ p : w \notin [[p]] \text{ and } v \in [[p]] \}.
\]

For example, suppose \( ATM = \{ p, q, r \}, w = \{ p, q, \neg r \} \text{ and } v = \{ p, \neg q, r \} \). Then
\[ \text{DIST}(w, v) = \{ q, r \}. \]

Usually, no matter what knowledge update operation is used, the minimal change rules should be applied. “We should look for the closest worlds among those not violating the laws[45].” We illustrate Winslett’s PMA [88] to show minimal change rules. PMA (Possible Models Approach) is for reasoning about action and change. It is based on minimization of the distance \( \text{DIST} \) between interpretations.

Let \( A \) be the formula representing the incoming information (the input). Then the update of \( w \) by \( [[A]] \) is defined as:

\[ w \cdot_{\text{pma}} [[A]] = \{ u \in [[A]] : \forall u' \in [[A]], \text{DIST}(w, u') \not\subset \text{DIST}(w, u) \}. \]

In other terms, the set \( w \cdot_{\text{pma}} [[A]] \) contains all those elements of \( [[A]] \) that are minimal with respect to the closeness ordering \( \leq_w \), where \( \leq_w \) is defined by

\[ u \leq_w v \quad \text{iff} \quad \text{DIST}(w, u) \subseteq \text{DIST}(w, v). \]

Every \( \leq_w \) is a partial pre-order over interpretations.

For example, for \( w = \{ \neg p, \neg q \} \) and \( A = p \lor q \), we get \( [[A]] = \{ \{ p, q \}, \{ \neg p, q \}, \{ p, \neg q \} \} \), \( \text{DIST}(w, \{ p, q \}) = \{ p, q \} \), \( \text{DIST}(w, \{ \neg p, q \}) = \{ q \} \) and \( \text{DIST}(w, \{ p, \neg q \}) = \{ p \} \). Thus, the models of \( A \) which are minimal for distance set inclusion are \( \{ \{ \neg p, q \}, \{ p, \neg q \} \} \). Hence we get \( w \cdot_{\text{pma}} [[p \lor q]] = \{ \{ \neg p, q \}, \{ p, \neg q \} \} \).

The above updates are categorized into knowledge based update. Recently, Baral and Zhang developed knowledge update in [2]. Their minimal change of knowledge update includes two aspects: actual-distance and knowledge-distance. The actual distance takes higher priority. If two changes of their actual distances towards the original model are the same, then the knowledge distances should be compared. The two changes should be compatible in the beginning. Otherwise, the comparison can not be conducted. More details are as follows. If two knowledge models \( M_1 = (W_1, w_1) \) and \( M_2 = (W_2, w_2) \) have different actual worlds then we define their closeness with respect to a reference knowledge model \( M = (W, w) \) by simply comparing the symmetric difference between \( w_1, w_2 \) and \( w \). When \( w_1 = w_2 \), we need additional comparisons, which are to compare the knowledge encoded in each of the
knowledge models. When $M_1$ only loses knowledge with respect to $M$: if $M_2$ both loses and gains knowledge with respect to $M$, then we consider $M_1$ to be closer to $M$ than $M_2$. Also, if $M_2$ (like $M_1$) only loses knowledge with respect to $M$, but loses more than $M_1$ does, then we consider $M_1$ to be closer to $M$ than $M_2$. In contrast, when $M_1$ only gains knowledge with respect to $M$: if $M_2$ both loses and gains knowledge with respect to $M$, then, we consider $M_1$ to be closer to $M$ than $M_2$. Also, if $M_2$ (like $M_1$) only gains knowledge with respect to $M$, but gains more than $M_1$ does, then we consider $M_1$ to be closer to $M$ than $M_2$.

Many researchers have extensively studied knowledge dynamics from the theoretical perspective. As we have learnt, in 1999, Herzig and Rifi [45] surveyed ten update operations proposed before 1999 and further researched their characteristics. They are all based on propositional logic. In 1997, Schoobbens and Ryan [77] pointed out a simple but hitherto ignored link between the theory of updates, and the theory of counterfactuals. Classical modal logic update is a classical existential modality, counterfactual is a classical universal modality, and the accessibility relations corresponding to these modalities are inverses. In 1998, van der Meyden [66] studied the computational aspect of knowledge modeling in distributed environments. He showed that his knowledge update presented a generalization of certain aspects of standard knowledge base update and used it for the purpose of efficiently implementing model checking. In 2000, Zhang and Foo in [89] investigated inclusive interpretation for disjunctions in updates from both syntactical and semantical viewpoints. The authors proposed two approaches, the minimal change with exceptions (MCE) and the minimal change with maximal disjunctive inclusions (MCD) to provide inclusive interpretations for disjunction in updates. The MCE and MCD are then characterized in terms of alternative minimal change criteria relating them to traditional Katsuno and Mendelzon’s update postulates. In 2002, Harris and Ryan in [44] showed that there is a strong connection between update and feature integration by, preliminarily, formulating SMV and the feature construct in propositional logic. In 2005, Baral and Zhang [2] proposed a formal method of knowledge update on the basis of the semantics of modal logic S5. They also investigated the computational complexity of model checking for knowledge update. The authors
discussed minimal change of knowledge update, characterizing specific knowledge updates which included gaining knowledge, ignorance, sensing and forgetting updates. The authors examined the persistence property of knowledge and ignorance associated with knowledge update. This paper updated knowledge based on modal logic.

1.3 Motivations

Although model checking and theorem proving approaches have been used for large complex system verification problems, one of the major limitations of these approaches is that they can only verify the correctness of a system specification. In other words, if a system specification is identified containing some errors by performing model checking or theorem proving, the task of correcting the system is completely left to system designers. That is, model checking and theorem proving generally only check the correctness of a system but cannot modify the system. The same limitation also remains in most diagnosis systems. Although the idea of repair has been indeed proposed in model based diagnosis methods, repairing a system is only possible for some specific cases [80].

Since model checking can handle complex system verification problems and may be implemented via fast algorithms, it is quite natural to consider whether we can improve model checking algorithms to perform system modification as well. In fact, the idea of integrating model checking and automatic modification has been investigated by some researchers recently. Buccafurri et al. [8] applied Artificial Intelligence techniques to model checking and error repairing. They used abductive model revision techniques, repairing errors in concurrent programs. It is a new approach towards automated diagnosis and integrated repair on model checking. It aims at using techniques and concepts from model checking by combining them with AI principles. However, as declared by the authors themselves, their approach may not be general enough for other system modification problems. Harris and Ryan [43] recently proposed another attempt at system modification. In their paper, model checking is formalized with a belief updating operator $\Box$ to satisfy classical proposition knowledge update KM postulates $U1-U8$. However, it is not clear how
their model update operators can be implemented for practical system modification. Recently, the update of the knowledge base is enhanced by the modality update by Baral and Zhang [2]. They discussed knowledge update and its minimal change, based on modal logic S5. This is the first step of the theoretical integration of model checking and knowledge update.

It is observed in [2] that system modification is closely related to knowledge update or belief update. From the knowledge dynamics perspective, we can view a finite transition system to be a model satisfying certain required properties, a knowledge set. Then the problem of system modification is reduced to the problem of updating this model in terms of the knowledge formula so that the new model can satisfy this formula. The methodology of unifying model checking and modification will closely retain the efficiency of model checking as well as being able to develop a systematic approach for system modification.

Based on the investigation of the current research as above, we intend to integrate knowledge update and model checking to develop a practical model updater, which represents a universal tool to repair errors automatically. We developed the research in both theory and implementation of model update, where the model checking technique and knowledge dynamics are combined to form a unified framework for system verification. The descriptions of major contributions follow in the next section.

1.4 Contributions

The overall aim of the research is to design a model updater to improve model checking functions by adding error repairing functions. The model updater consists of the functions shown in Figure 1.5. The difference between this figure and Figure 1.1 is to add updating functions after model checking functions. The outcome from the updater is a corrected Kripke model. The model updater’s function is to automatically correct errors which are reported as counterexamples by a model checking compiler. An eventual model updater is intended to be a universal compiler which can be used in certain common situations for model error detection and correction.

The detailed contributions for this dissertation include the theoretical foundation for model correction and the implementation of a prototype model updater used for
three case studies. Our prototype model updater is a stand alone utility which is yet to be integrated into a formal model checking compiler.

\begin{figure} 
\centering
\includegraphics[width=0.5\textwidth]{model-checking.png} 
\caption{Model Checking with Knowledge Update} 
\end{figure}

1.4.1 Formal Framework of CTL Model Update

For our research, we have selected CTL logic to describe an input model (hardware or software system) and specification properties which are to be checked for error and repaired if necessary. As described in [86], model checking is usually applied to finite systems, whereas software systems are in general infinite state machines. CTL formulae are interpreted with respect to an infinite computational tree derived from finite state transition machines. Each path in the tree is a sequence of states. At the moment, CTL is a suitable logic approach to describe both hardware and software systems.

In the theoretical research for CTL model update, first, the logic of CTL is analyzed. Next, atomic update operations for each temporal logic are extracted. Minimal change rules are proposed based on these atomic update operations. Minimal change achieves optimal update by reducing the number of updated models. According to the minimal change rules and the semantics of CTL logic, characterizations are deduced. Characterizations give the update rules for each semantic of CTL logic, such as EX or AF, to standardize and simplify the update process. Furthermore, the complexities of characterizations are analyzed. Based on the characterizations and the syntax of CTL, algorithms for directing CTL model update
implementation are designed. Sometimes, too many updated models result from the application of the minimal change rules. We have named this effect, \textit{model explosion}. To solve this problem, a reachable state algorithm is introduced to constrain the previous minimal change rules, thus significantly reducing the number of updated models. The final stage of our research is to implement a prototype CTL model updater based on the previous theoretical foundation and apply it to case studies of three system models.

1.4.2 Implementation of the CTL Model Updater

We choose C as the implementation language for the model updater. C is very good for system level programming. Our model updater prototypes model correction philosophy to be integrated into a model checking compiler. It involves a certain degree of system programming. Most model checkers are coded in C/C++. It is intended that integration of our model updater implementation into an existing model checker involve a minimum of recoding effort.

The prototype of the CTL model updater has been implemented in C code in Linux. Our prototype contains over 6000 lines of code, and consists of a separate implementation for each of three well known models. The intrinsic mechanisms of the model updater are as shown in Figure 5.2. They consist of the input - a model which needs to be updated, and the specification properties, a specification string parser, model checking and update functions, library functions and the output - the updated model. The model is structured by predefined model definition functions. The specification properties are parsed before model checking and update. The model checking functions identify the false states on false paths. Then, the model update functions correct the false states. The model update functions implement our theoretical results from research described in the previous section. These functions include performing the atomic update operations, the characterizations containing minimal change rules and the reachable state algorithm to solve a consequential model explosion problem. The output is in the form of screen text and shows updated models by redirecting the transition links and changing values of variables in states. In addition, place markers for more general algorithms are included to
assist the future integration to a formal compiler.

### 1.4.3 Case Studies

We have selected three well known models as case studies for our prototype of the CTL model updater. One of the three models is the microwave oven Kripke model which has been repeatedly illustrated as a fundamental hardware model example by Clarke et al. in [16]. The other two models are Andrew File System characterizations, known as the AFS0 and AFS1 models. These were the first software protocol models used as case studies for the SMV model checker. AFS0 has a lower complexity and was used in [84] as a part of Vaziri-Farahani’s bachelor thesis; AFS1 is much more complex and was illustrated by Wing and Vaziri-Farahani in [86]. The Kripke models for AFS0 and AFS1 are extracted from the NuSMV compiler after processing. The extraction method uses interactive mode commands for NuSMV to determine the state machine paths from each valid initial state.

Our model updater prototype parses the specification properties, performs model checking, and applies characterizations containing minimal change rules and atomic update operations to each model. In addition to the functions applied to the microwave oven example, the reachable state algorithm to solve the state explosion problem is invoked for the AFS0 and AFS1 models. In a nutshell, our model updater prototype has been successfully applied to these three models, indicating that it is suitable for both hardware and software models, and can deal with at least the complexity of the AFS1 model. Our implementation demonstrates that our theoretical results are sound and workable for the creation of a practical model updater.

### 1.5 Organization of the Dissertation

This dissertation is organized as follows: Chapter 2 first reviews the syntax and semantics of CTL, then introduces the fundamental theoretical results of CTL model update, atomic update operations, and minimal change rules. Chapter 3 introduces characterizations and complexity analysis for CTL semantics for model update. Chapter 4 introduces algorithms for CTL model update based on the char-
acterizations from the previous chapter. Chapter 5 introduces our prototype of the model updater. Chapter 6 presents three well known examples to demonstrate our CTL model updater prototype. A technique for extracting Kripke models from the NuSMV compiler is also presented. Chapter 7 presents a study of a model explosion problem which became apparent during the implementation of the model updater. The dissertation concludes with Chapter 8 which discusses proposed future research. In addition, Appendix A presents the research for LTL model update. Appendix B introduces the detailed method to extract Kripke models from NuSMV. Appendix C contains the ReadMe file for our prototype CTL model updater to assist the understanding of the prototype. Appendix D lists the related published papers to this dissertation.
Chapter 2

Minimal Change for CTL Model
Update

2.1 CTL Syntax and Semantics

Before introducing our theoretical research, we briefly review the syntax and semantics of CTL. Readers are referred to [16] and [50] for details. Temporal logic is a formalism for describing sequences of transitions between states in a reactive system [16]. Kripke structures, which are also the models for CTL, can describe this system. The definition of a Kripke model is as follows.

**Definition 2.1** [16] Let $AP$ be a set of atomic propositions. A Kripke model $M$ over $AP$ is a three tuple $M = (S, R, L)$ where

1. $S$ is a finite set of states,
2. $R \subseteq S \times S$ is a transition relation,
3. $L : S \rightarrow 2^{AP}$ is a function that assigns each state with a set of atomic propositions.

The finite Kripke model can be explained as a graph in Figure 2.1. In this graph, a collection of the nodes (the states $S$) contain all propositional atoms that are true in that state, and a set of relations $R$ are for the system moving from state to state. State $s_0$ contains propositional atoms $p$ and $q$ being true; state $s_1$ contains $q$ and $r$ being true and state $s_2$ containing $r$ being true. These propositional atoms are $L(s)$ in definition 2.1.
Figure 2.1: A Transition State Graph

Figure 2.2: Unwinding the Transition State Graph as an Infinite Tree
Computation Tree Logic (CTL), is a temporal logic, having connectives that allow us to refer to the future. It is also a branching-time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be the ‘actual’ path that is realized [50].

**Definition 2.2** [50] Computation tree logic (CTL) has the following syntax given in Backus Naur form:

\[ \phi ::= \top | \bot | p | (\neg \phi) | (\phi_1 \land \phi_2) | (\phi_1 \lor \phi_2) | \phi_1 \rightarrow \phi_2 | \phi_1 \mathbf{AX} \phi | \phi_1 \mathbf{EX} \phi | \phi_1 \mathbf{AG} \phi | \phi_1 \mathbf{EG} \phi | \phi_1 \mathbf{AF} \phi | \phi_1 \mathbf{EF} \phi | \phi_1 \mathbf{AU} \phi_2 | \phi_1 \mathbf{EU} \phi_2 \]

where \( p \) is any propositional atom.

The symbols \( \top \) and \( \bot \) are CTL formulas, as are all atomic descriptions; \( \neg \phi \) is a CTL formula if \( \phi \) is one, etc. The connectives \( \mathbf{AX}, \mathbf{EX}, \mathbf{AG}, \mathbf{EG}, \mathbf{AU}, \mathbf{EU} \) and \( \mathbf{EF} \) are called temporal connectives. Each of the CTL temporal connectives is a pair of symbols. The first of the pair is one of \( A \) and \( E \). \( A \) means ‘along All paths’ (inevitably) and \( E \) means ‘along at least (there Exists) one path’ (possibly). The second one of the pair is \( X, F, G, \) or \( U \), meaning ‘neXt state’, ‘some Future state’, ‘all future states (Globally)’ and Until, respectively. The pair of operators in \( \phi_1 \mathbf{AU} \phi_2 \) is binary. The symbols \( X, F, G \) and \( U \) cannot occur without being preceded by an \( A \) or an \( E \); similarly, every \( A \) or \( E \) must have one of \( X, F, G \) and \( U \) to accompany it [50].

**Convention 1** [50] The unary connectives (consisting of \( \neg \) and the temporal connectives \( \mathbf{AG}, \mathbf{EG}, \mathbf{AF}, \mathbf{EF}, \mathbf{AX} \) and \( \mathbf{EX} \)) bind most tightly. Next in the order come \( \land \) and \( \lor \); and after that come \( \rightarrow \), \( \mathbf{AU} \) and \( \mathbf{EU} \).

A CTL formula is evaluated on a Kripke model \( M \). A path in \( M \) from a state \( s \) is an infinite sequence of states \( \pi \equiv [s_0, s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots] \) such that \( s_0 = s \) and \( (s_i, s_{i+1}) \in R \) holds for all \( i \geq 0 \). We write \( (s_i, s_{i+1}) \subseteq \pi \) and \( s_i \in \pi \). If we express a path as \( \pi = [s_0, s_1, \ldots, s_i, \ldots, s_j, \ldots] \) and \( i < j \), we say that \( s_i \) is a state **earlier** than \( s_j \) in \( \pi \) as \( s_i < s_j \). For simplicity, we use \( \text{pre}(s) \) to denote state \( s' \) if
there is a relation \((s', s)\) in \(R\); we use \(\text{succ}(s)\) to denote state \(s''\) if there is a relation \((s, s'')\) in \(R\). We use \(\text{succ}(s, \neg \phi)\) if \(\text{succ}(s) \not\models \phi\). Now, we review the semantics of CTL as follows.

**Definition 2.3** [50] Let \(M = (S, R, L)\) be a Kripke model for CTL. Given any \(s\) in \(S\), we define whether a CTL formula \(\phi\) holds in state \(s\). We denote this by \((M, s) \models \phi\). The satisfaction relation \(\models\) is defined by structural induction on all CTL formulas:

1. \((M, s) \models \top\) and \(M, s \not\models \bot\) for all \(s \in S\).
2. \((M, s) \models p\) iff \(p \in L(s)\).
3. \((M, s) \models \neg \phi\) iff \((M, s) \not\models \phi\).
4. \((M, s) \models \phi_1 \land \phi_2\) iff \((M, s) \models \phi_1\) and \((M, s) \models \phi_2\).
5. \((M, s) \models \phi_1 \lor \phi_2\) iff \((M, s) \not\models \phi_1\), or \((M, s) \models \phi_2\).
6. \((M, s) \models \phi_1 \to \phi_2\) iff \((M, s) \not\models \phi_1\), or \((M, s) \models \phi_2\).
7. \((M, s) \models \text{AX} \phi\) iff for all \(s_1\) such that \((s, s_1) \in R\), \((M, s_1) \models \phi\).
8. \((M, s) \models \text{EX} \phi\) iff for some \(s_1\) such that \((s, s_1) \in R\), \((M, s_1) \models \phi\).
9. \((M, s) \models \text{AG} \phi\) holds iff for all paths \([s_0, s_1, s_2, \cdots]\), where \(s_0 = s\), and all \(s_i\) along the path, \((M, s_i) \models \phi\).
10. \((M, s) \models \text{EG} \phi\) holds iff there is a path \([s_0, s_1, s_2, \cdots]\), where \(s_0 = s\), and for all \(s_i\) along the path, \((M, s_i) \models \phi\).
11. \((M, s) \models \text{AF} \phi\) holds iff for all paths \([s_0, s_1, s_2, \cdots]\), where \(s_0 = s\), there is some \(s_i\) in the path such that \((M, s_i) \models \phi\).
12. \((M, s) \models \text{EF} \phi\) holds iff there is a path \([s_0, s_1, s_2, \cdots]\), where \(s_0 = s\), and for some \(s_i\) along the path, \((M, s_i) \models \phi\).
13. \((M, s) \models \text{A} [\phi_1 \mathcal{U} \phi_2]\) holds iff for all paths \([s_0, s_1, s_2, \cdots]\), where \(s_0 = s\), the path satisfies \(\phi_1 \mathcal{U} \phi_2\), i.e. there is some \(s_i\) along the path, such that \((M, s_i) \models \phi_2\), and, for each \(j < i\), \((M, s_j) \models \phi_1\).
14. \((M, s) \models E[φ_1 U φ_2]\) holds iff there is a path \([s_0, s_1, s_2, \ldots]\), where \(s_0 = s\), the path satisfies \(φ_1 U φ_2\), i.e. there is some \(s_i\) along the path, such that \((M, s_i) \models φ_2\), and, for each \(j < i\), \((M, s_j) \models φ_1\).

In the first 6 clauses, the truth value of the formula in the state depends on the truth value of \(φ_1\) or \(φ_2\) in the same state. For example, the truth value of \(¬φ\) in a state depends on the truth value of \(φ\) in the same state. This contrasts with the clauses 7 and 8 for AX and EX. The truth value of AX\(φ\) in a state \(s\) is determined not by the truth value of \(φ\) in \(s\), but by \(φ\) in states that are related to \(s\) by the relation \(R\); if \((s, s) \in R\), then this value also depends on the truth value of \(φ\) in \(s\).

The next four clauses (9 - 12) also exhibit this phenomenon. For example, the truth value of AG\(φ\) involves looking at the truth value of \(φ\) not only in the immediately related states, but in indirectly related states as well. In the case of AG\(φ\), we have to examine the truth value of \(φ\) in every state related by any number of forward links of \(→\) to the current state.

Clauses 9-14 above refer to computation paths in models. It is therefore useful to visualize all possible computation paths from a given state \(s\) by unwinding the transition system to obtain an infinite computation tree, whence “computation tree logic”. This greatly facilitates deciding whether a state satisfies a CTL formula. The unwound tree of the graph in Figure 2.1 is as Figure 2.2.

In Figure 2.2, if \(φ = r\), then AX\(r\) is true; if \(φ = q\), then EX\(q\) is true. In the same figure, if \(φ = r\), then AF\(r\) is true because some states on all paths sometimes in the future satisfy \(r\). If \(φ = q\), EF\(q\) is true because some states on some paths satisfy \(q\). In the same figure, if \(φ_1 = q\), \(φ_2 = r\), then A[\(φ_1 U φ_2\)] is true; if \(φ_1 = p\), \(φ_2 = r\), then, E[\(φ_1 U φ_2\)] is true. The clauses for AG and EG can be explained in Figure 2.3. In this tree, all states satisfy \(φ = r\). Thus, AG\(r\) is true for this Kripke model. There is one path where all states satisfy \(φ = q\). Thus, EG\(q\) is true for this Kripke model.

In this dissertation, without explicit declaration, we will assume that all CTL formulas occurring in our context will be satisfiable. For instance, when we consider to update a Kripke model with a CTL formula \(φ\), we already assume that \(φ\) is satisfiable.

**Definition 2.4** [50] Two CTL formulas \(φ\) and \(ψ\) are said to be semantically equiv-
Given a CTL kripke model and a (satisfiable) CTL formula, we consider how this model can be updated in order to satisfy the given formula. We first give the following general definition on CTL model update.

**Definition 2.5 (CTL Model Update)** Given a CTL Kripke model \( M = (S, R, L) \) and a CTL formula \( \phi \). \( \mathcal{M} = (M, s_0) \not\models \phi \), where \( s_0 \in S \). An update of \( \mathcal{M} = (M, s_0) \)
§2.2  CTL Model Update

The illustration of Model Update

to satisfy $\phi$ is a CTL Kripke model $M' = (S', R', L')$ such that $M' = (M', s'_0) \models \phi$ where $s'_0 \in S'$. We use Update$(M, \phi)$ to denote the result $M'$ and Update$(M, \phi) = M$ if $M \models \phi$.

We use an example to illustrate the concept of model update. In Figure 2.4, if $\phi = p$, the model on the left side does not satisfy AG$p$, because only $s_0$ contains $p = \text{true}$. After updating states $s_1$ and $s_2$ in the left side model, we have the model on the right side which satisfies AG$p$, where all states contain $p = \text{true}$. If we want the model on the right side to satisfy $\neg$AG$p$, then we should update it to the model on the left side, where not all states contain $p = \text{true}$, thus this model satisfies $\neg$AG$p$. During this update, states $s'_1$ and $s'_2$ are updated with $p = \text{false}$. These updates are not constrained by minimal change which will be introduced in the next section.

Model update and its algorithms are published in [29]. Definition 2.5 only presents an essential requirement for a CTL model update and does not tell how such an update should be conducted. Basically, as in traditional knowledge base update [87], we would expect that a CTL model update obeys an underlying minimal change principle. Furthermore, this minimal change should be defined based on some operational process so that a concrete algorithm for CTL model update can be implemented. The theory of minimal change is published in [28]. We first consider five primitive operations on the CTL model that provide a basis for all complex CTL model updates.
2.3 Primitive Operations

The operations to update the CTL model can be decomposed into five types identified as PU1, PU2, PU3, PU4 and PU5. These primitive updates are defined in their simplest forms as follows.

**PU1: Adding a relation only**
Given $M = (S, R, L)$, its updated model $M' = (S', R', L')$ is the result of $M$ having only added one new relation. That is $S' = S$, $L' = L$, and $R' = R \cup \{(s_{ar}, s_{ar2})\}$ where $(s_{ar}, s_{ar2}) \notin R$ for one pair of $s_{ar}, s_{ar2} \in S$.

**PU2: Removing a relation only**
Given $M = (S, R, L)$, its updated model $M' = (S', R', L')$ is the result of $M$ having only removed one existing relation. That is, $S' = S$; $L' = L$, and $R' = R - \{(s_{rr}, s_{rr2})\}$ where $(s_{rr}, s_{rr2}) \in R$ for one pair of $s_{rr}, s_{rr2} \in S$.

**PU3: Substituting a state and its associated relation(s) only**
Given $M = (S, R, L)$, its updated model $M' = (S', R', L')$ is the result of $M$ having only substituted one existing state and its associated relation(s). That is, $S' = S[s/s_{ss}]$ (i.e. $S'$ are the set of states where one state $s$ in $S$ is substituted by $s_{ss}$), $R' = R - \{(s_i, s), (s, s_j) \mid (s_i, s), (s, s_j) \in R\} \cup \{(s_i, s_{ss}), (s_{ss}, s_j) \mid (s_i, s), (s, s_j) \in R\}$, and $\forall s \in S \cap S'$, such that $L'(s) = L(s)$, and $L'(s_{ss})$ is a set of true variables assigned in $s_{ss}$.

**PU4: Adding a state and its associated relation(s) only**
Given $M = (S, R, L)$, its updated model $M' = (S', R', L')$ is the result of $M$ having only added one new state and its associated relation(s). That is, $S' = S \cup \{s_{as}\}$, $R' = R \cup \{(s_i, s_{as}), (s_{as}, s_j) \mid$ for some $s_i, s_j \in S'\}$, and $\forall s \in S \cap S'$, such that $L'(s) = L(s)$, and $L'(s_{as})$ is a set of true variables assigned in $s_{as}$.

**PU5: Removing a state and its associated relation(s) only**
Given $M = (S, R, L)$, its updated model $M' = (S', R', L')$ is the result of $M$ having only removed one existing state and its associated relation(s). That is, $S' = S - \{s_{rs} \mid s_{rs} \in S\}$, $R' = R - \{(s_i, s_{rs}), (s_{rs}, s_j) \mid$ for some $s_i, s_j \in S\}$, and $L'(s) = L(s)$ for all $s \in S \cap S'$ (note $S' \subset S$).

We call the above five operations *atomic* since all changes on a CTL model can
be expressed by these five operations. It may be argued that PU3 can be expressed by PU4 and PU5. However, we will treat state substitution differently from a combination of state addition and state deletion. That is, in our context, whenever a state substitution is needed, we will apply PU3 directly rather than PU4 followed by PU5. This will simplify our definition on CTL model minimal change.

We illustrate some of the operations to explain the primitive operations. In Figure 2.5, M1 is updated with PU2 from M by removing a relation \((s_1, s_2)\) from M; M2 is updated with PU3 from M by substituting \(s_3\) with \(s'_3\) and associated relations in M.

### 2.3.1 Examples for Combinations of Primitive Updates

One kind of combination is that each primitive update is repeated a few times within an updated model. Besides this, which is relatively simple, the combinations of different primitive updates could result in \(C_2^5 + C_3^5 + C_4^5 + C_5^5 = 26\) possible cases. We give a couple of examples to illustrate combinations of primitive updates.

**Case A: Combining PU1, PU2 and PU3**

Given \(M = (S, R, L)\), its updated model \(M' = (S', R', L')\) is the result of \(M\) having added a new relation, removing an existing relation and substituting an existing state and its associated relations with a new state and its associated relations, iff

\[
S' = S \cup \{s_{ss}|s_{ss} \notin S, s_{ss} \in S'\} - \{s_i|s_i \in S, s_i \notin S'\};
\]

\[
R' = R \cup \{(s_{ar}, s_{ar2})|(s_{ar}, s_{ar2}) \notin R, s_{ar} \in S, s_{ar} \in S', s_{ar2} \in S, s_{ar2} \in S'\}
\]

\[
\cup \{(s_{i-1}, s_{ss})|s_{ss}, s_{i+1}+1|s_{i-1} \in S, s_{i-1} \in S', s_{i+1} \in S, s_{i+1} \in S'\}
\]
Figure 2.6: The illustration of combinations of primitive updates

\[ - \{(s_{rr}, s_{rr2}) | (s_{rr}, s_{rr2}) \in R, s_{rr} \in S, s_{rr} \in S', s_{rr2} \in S, s_{rr2} \in S'\} \]

\[ - \{(s_{i-1}, s_i), (s_i, s_{i+1}) | s_{i-1} \in S, s_{i-1} \in S', s_{i+1} \in S, s_{i+1} \in S'\}; \]

\[ L': S \rightarrow 2^{AP}, \text{ where } \forall s \in S', \text{ if } s \in S, \text{ then } L'(s) = L(s), \text{ else } \]

\[ L'(s_{as}) \text{ is a set of true variables assigned in } s_{as}. \]

**Example 1** In Figure 2.6, \( M_1 \) is a resulting model by updating \( M \) with PU1, PU2 and PU3 as in case A. During the update, a new relation \((s_0, s_2)\) is added to \( M \) which belongs to PU1; an existing relation \((s_1, s_2)\) is removed from \( M \) which belongs to PU2; an existing state \( s_3 \) is substituted with \( s'_3 \) and its associated relations \((s_2, s_3)\) and \((s_3, s_0)\) are substituted with \((s_2, s'_3)\) and \((s'_3, s_0)\) respectively, which belongs to PU3. \( \square \)

**Case B: Combining PU2, PU4 and PU5**

Given \( M = (S, R, L) \), its updated model \( M' = (S', R', L') \) is the result of \( M \) having added a new state, removing an existing relation and removing a state with its associated relations, if

\[ S' = S \cup \{s_{as} | s_{as} \notin S, s_{as} \in S'\} - \{s_{rs} | s_{rs} \in S, s_{rs} \notin S'\}; \]

\[ R' = R \cup \{(s_{as-1}, s_{as}) | (s_{as}, s_{as+1}) | s_{as-1} \in S, s_{as-1} \in S', s_{as+1} \in S, s_{as+1} \in S'\} \]

\[ - \{(s_{rs-1}, s_{rs}) | s_{rs-1} \in S, s_{rs-1} \in S', s_{rs+1} \in S, s_{rs+1} \in S'\} \]

\[ - \{(s_{rr}, s_{rr2}) | (s_{rr}, s_{rr2}) \in R, s_{rr} \in S, s_{rr} \in S', s_{rr2} \in S, s_{rr2} \in S'\}; \]

\[ L': S \rightarrow 2^{AP}, \text{ where } \forall s \in S', \text{ if } s \in S, \text{ then } L'(s) = L(s), \text{ else } \]

\[ L'(s_{as}) \text{ is a set of true variables assigned in } s_{as}. \]
Example 2 In Figure 2.6, $M_2$ is a resulting model by updating $M$ with PU2, PU4 and PU5 as in case B. During the update, an existing relation $(s_1, s_2)$ is removed from $M$ which belongs to PU2; a new state $s'$ and its associated relations $(s_0, s')$ and $(s', s_1)$ are added to $M$, which belongs to PU4; and an existing state $s_3$ and its associated relations $(s_2, s_3)$ and $(s_3, s_0)$ are removed, which belongs to PU5. □

2.4 Defining Minimal Change

In order to define the minimal change criteria for CTL model update, we need to consider changes on both states and relations for the underlying CTL models. We achieve this by specifying the differences of states and relations on CTL models through primitive operations. First we introduce some useful notions.

Given two CTL models $M = (S, R, L)$ and $M' = (S', R', L')$, $Diff_{PUi}(M, M')$ denotes the differences between two CTL models after certain primitive operations, where $M'$ is a resulting model from $M$ and $PU_i$ could be any primitive update within PU1-PU5 in section 2.3. Since PU1 and PU2 only change relations, we define $Diff_{PUi}(M, M') = (R - R') \cup (R' - R)$ ($i = 1, 2$). For operations PU3, PU4 and PU5, on the other hand, we define $Diff_{PUi}(M, M') = (S - S') \cup (S' - S)$ ($i = 3, 4, 5$). Although any changes on states caused by PU3-PU5 will also imply corresponding changes on relations, we only count changes on states and take state changes as the primitive factor to measure the difference between $M$ and $M'$. For operation PU3, we should also consider the case that when a state is substituted by a new state, we require the difference between these two states to be minimal under the condition of satisfying the updated formula. Finally, we specify

$$Diff(M, M') = (Diff_{PU1}(M, M'), \ldots, Diff_{PU5}(M, M')),$$

which contains the difference of each of the five primitive updates. If the change of a primitive update $Diff_{PUi}(M, M')$ in $Diff(M, M')$ is empty, it means this type of primitive update $PU_i$ is not applied during the update process.

\footnote{We consider that a state contains more information than its associate relations and state changes affect a system more than associate relation changes, thus, associated relation changes do not play a crucial role to influence an updated CTL model.}
Let $M$, $M_1$ and $M_2$ be three CTL models. We denote $Diff(M, M_1) \subseteq Diff(M, M_2)$ iff (1) for each $i$ $(i = 1, \ldots, 5)$, $Diff_{PU_i}(M, M_1) \subseteq Diff_{PU_i}(M, M_2)$;
or (2) $Diff_{PU_i}(M, M_1) \subseteq Diff_{PU_i}(M, M_2)$ for each $i = 1, 2, 4, 5,$
and $|Diff_{PU_3}(M, M_1)| = |Diff_{PU_3}(M, M_2)|$ implies $Diff(s, s_1) \subseteq Diff(s, s_2)$,
if any state $s$ in $M$ is substituted by $s_1$ in $M_1$ or $s_2$ in $M_2$ respectively.

The above notation shows conditions in that one updated model $M_1$ is more
minimally changed than another $M_2$ if both of them are updated from the same
original model $M$. Condition (1) states that changes on relations (in the cases of
PU1 and PU2) or states (in the cases of PU3, PU4 and PU5) from updating $M$ to
$M_1$ are subsets of changes from updating $M$ to $M_2$ under each operation PU1, PU2,
PU3, PU4, Pu5 respectively. For example, if $M$ is updated to $M_1$ using PU1 by
adding a relation $r$, $M$ is updated to $M_2$ using PU1 by adding two relations $\{r, r'\}$
and using PU4 by adding a state $s'$, then the updated model $M_1$ using PU1 is more
minimally changed comparing with $M_2$ because $\{r\} \subset \{r, r'\}$; the updated model
$M_1$ using PU4 is more minimally changed comparing with $M_2$ because $\{\emptyset\} \subset \{s'\}$.
Thus, $M_1$ is a more minimally changed model than $M_2$. Condition (2) states a
special case for comparison using the operation PU3. If updates from $M$ to $M_1$ and
$M_2$ using PU3 are both on the same state $s$, then the changes of variables from $s$
in $M$ to $s_1$ in $M_1$ should be compared with the changes of variables from $s$ in $M$
to $s_2$ in $M_2$. For example, in addition to the previous example, we have changes
from updating $s$ in $M$ to $s_1$ in $M_1$ or to $s_2$ in $M_2$. If $s$ contains variables $\{a, b\}$,
s$1$ contains variables $\{a, b, c\}$ and $s_2$ contains variables $\{a, b, c, d\}$, then the change
from $s$ to $s_1$ is $\{c\}$ and from $s$ to $s_2$ is $\{c, d\}$. It is obvious $\{c\} \subset \{c, d\}$, thus, $M_1$ is
more minimally changed than $M_2$ from $M$. These ideas are formalized as follows.

**Definition 2.6 (Closeness Ordering)** Given three CTL Kripke models $M$, $M_1$
and $M_2$, where $M_1$ and $M_2$ are obtained from $M$ by applying PU1-PU5 operations.
We say that $M_1$ is closer or as close to $M$ as $M_2$, denoted as $M_1 \leq_M M_2$, iff
$Diff(M, M_1) \subseteq Diff(M, M_2)$. We denote $M_1 <_M M_2$ if $M_1 \leq_M M_2$ and $M_2 \not\leq_M M_1$.

**Definition 2.7 (Admissible Update)** Given a CTL Kripke model $M = (S, R, L)$,
$M = (M, s_0)$ where $s_0 \in S$, and a CTL formula $\phi$, $Update(M, \phi)$ is called admis-
sible if the following conditions hold: (1) $\text{Update}(M, \phi) = (M', s'_0) \models \phi$ where $M' = (S', R', L')$ and $s'_0 \in S'$; and (2) there does not exist another resulting model $M'' = (S'', R'', L'')$ and $s''_0 \in S''$ such that $(M'', s''_0) \models \phi$ and $M'' <_M M'$.

**Example 3** In Figure 2.7, model $M$ is updated in two different ways. $M_1$ is a resulting model after $M$ is updated by applying PU1. $M_2$ is another resulting model after $M$ is updated by applying PU1 and PU5. We have $\text{Diff}_{PU1}(M, M_1) = \{(s_0, s_2)\}$, $\text{Diff}_{PU1}(M, M_2) = \{(s_1, s_0), (s_0, s_2)\}$, such that $\text{Diff}_{PU1}(M, M_1) \subset \text{Diff}_{PU1}(M, M_2)$; $\text{Diff}_{PU5}(M, M_1) = \emptyset$, $\text{Diff}_{PU5}(M, M_2) = \{s_3\}$, such that $\text{Diff}_{PU5}(M, M_1) \subset \text{Diff}_{PU5}(M, M_2)$; we have $\text{Diff}(M, M_1) < \text{Diff}(M, M_2)$, such that $M_1 \leq_M M_2$ and $M_1$ is the resulting model of admissible update. \(\Box\)
3.1 Semantic Characterizations

From Definition 2.7, we observe that for a given CTL Kripke model $M$ and a formula $\phi$, there may be many admissible updates to satisfy $\phi$, where some updates are simpler than others. In this chapter, we provide various semantic characterizations on CTL model update that present possible solutions to achieve admissible updates under certain conditions. The characterizations are published in [26]. In general, in order to achieve admissible update results, we may have to combine various primitive operations during an update process. Nevertheless, as will be shown in the following, for many situations, a single type primitive operation will be enough to achieve an admissible updated model. These characterizations also play an essential role to simplify CTL model update implementations and improve the efficiency of the CTL model update performance.

**Theorem 3.1** Let $M = (S, R, L)$ be a Kripke model and $s_0$ an initial state in $S$ and $M = (M, s_0) \not\models \text{EX} \phi$, where $\phi$ is a propositional formula. Then an admissible updated model $M' = \text{Update}(M, \text{EX} \phi)$ can be obtained by one of the following operations:

1. if $\forall s_i \not\models \phi$, where $s_i \in S$, $\text{PU3}$ is applied to any $\text{succ}(s_0)$ once to substitute it with a new state $s^* \models \phi$ and $\text{Diff}(\text{succ}(s_0), s^*)$ to be minimal, or $\text{PU4}$ is applied one time by adding a new state $s^* \models \phi$ and a new relation $(s_0, s^*)$;
2. if there exists some \( s_i \in S \) such that \( s_i \models \phi \) and \( s_i \neq \text{succ}(s_0) \), PU1 is applied one time to add a new relation \((s_0, s_i)\).

Theorem 3.1 provides two cases where admissible CTL model update results can be achieved for formula \( \text{EX} \phi \). The first case says that we can either select one of \( s_0 \)'s successor states and substitute it with a new state satisfying \( \phi \) (i.e. applying PU3 one time), or simply add a new state that satisfies \( \phi \) as a successor of \( s_0 \) (i.e. applying PU4 one time), if no state in \( S \) satisfies \( \phi \). The second case indicates that if there is some state \( s_i \) in \( S \) that already satisfies \( \phi \), then it is enough to simply add a new relation \((s_0, s_i)\) and make it a successor of \( s_0 \) (i.e. applying PU1 once). It is easy to see that both cases will yield new CTL models that satisfy \( \text{EX} \phi \). The theorem shows that such new models are also minimally changed models with respect to the original CTL model, which is proved as follows.

**Proof.** For case 1, after PU3 is applied to a \( \text{succ}(s_0) \), or PU4 is applied to add a new state \( s^* \) as a successor of \( s_0 \), the new model \( M' \) contains a \( \text{succ}(s_0) \) which satisfies \( \phi \). Thus, \( M' \models \text{EX} \phi \). If PU3 is applied once, then

\[
A = \text{Diff}(M, M') = (\emptyset, \emptyset, \{\text{succ}(s_0), s^*\}, \emptyset, \emptyset); \quad \text{if PU4 is applied once, } \text{Diff}(M, M') = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset).
\]

Thus, the updates by the application of PU3 or PU4 once are not compatible. For PU3, if any other update is applied together with PU3, \( \text{Diff}(M, M') \) will be either not compatible with \( A \) or contains \( A \) (e.g., another PU3 together with the previous PU3); same principle for PU4. Thus, either PU3 and PU4 applied once represents minimal change. For case 2, after PU1 is applied to connect \( s_0 \) and \( s_i \models \phi \), the new model \( M' \) has a successor which satisfies \( \phi \). Thus, \( M' \models \text{EX} \phi \). If PU1 is applied, \( \text{Diff}(M, M') = (\{(s_0, s_i)\}, \emptyset, \emptyset, \emptyset, \emptyset) \). This case and case 1 are not compatible. Thus, case 2 represents an equal minimal change to case 1. \( \square \)

**Theorem 3.2** Let \( M = (S, R, L) \) be a Kripke model and \( M = (M, s_0) \not\models \text{AX} \phi \), where \( s_0 \in S \) and \( \phi \) is a propositional formula. Then an admissible updated model \( M' = \text{Update}(M, \text{AX} \phi) \) can be obtained by one of the following results:

1. if \( \forall \text{succ}(s_0) \not\models \phi \), which implies \( s' = \text{succ}(s_0, \lnot \phi) \), then first PU3 is applied to substitute some \( s' \) with \( s^* \models \phi \) and \( \text{Diff}(s', s^*) \) is minimal, and then
if \( \exists \text{succ}(s_0) \models \phi \), then PU2 is applied to delete some relation(s) \((s_0, \text{succ}(s_0, \neg \phi))\), PU3 to substitute some remaining state(s) \(\text{succ}(s_0, \neg \phi)\) or PU5 to delete any other remaining state(s) \(\text{succ}(s_0, \neg \phi)\);

2. if \( \nexists \text{succ}(s_0) \) and \( \forall s_i \models \phi \), where \( s_i \in S \), then PU4 is applied to next to \( s_0 \) once to add a state \( s^* \models \phi \);

3. if \( \nexists \text{succ}(s_0) \) and \( \exists s_i \models \phi \), where \( s_i \in S \), then PU1 is applied to connect \((s_0, s_i)\) once.

For Theorem 3.2, in case 1, if none of successors of the initial state \( s_0 \) satisfies \( \phi \), then, we should keep at least one path reachable to protect the structure of the original model, where \( \text{succ}(s_0, \neg \phi) \) should be updated with PU3. The other successors of \( s_0 \) could be updated with either PU2, PU3 or PU5. PU2 is for deleting the relations \((s_0, \text{succ}(s_0, \neg \phi))\) and PU5 is for deleting states \(\text{succ}(s_0, \neg \phi)\). After these updates, the new updated CTL model satisfies \(\text{AX} \phi\). In case 2, if the initial state \( s_0 \) is a solo state and all other states are not reachable from \( s_0 \), and also, none of the states satisfy \( \phi \), then we should apply PU4 to add a state next to \( s_0 \). Then, \( s_0 \) has a successor and this successor should satisfy \( \phi \). Thus, the new updated CTL model satisfies \( \text{AX} \phi \). In case 3, the circumstance is the same as for case 2 but there are some states satisfying \( \phi \), so, we should link \( s_0 \) and a state \( s_i \models \phi \) by applying PU1 to make a successor of \( s_0 \) satisfying \( \phi \). Thus, the new updated CTL model satisfies \( \text{AX} \phi \). These methods are all minimal changes. We prove them as follows.

**Proof.** In case 1, after these updates, all successors of \( s_0 \) satisfy \( \phi \). Thus, the new updated CTL model satisfies \( \text{AX} \phi \). We assume there are \( n = 6 \) successors which do not satisfy \( \phi \). Then, we can apply PU3 to one successor, PU2 to two successors and PU5 to three successors. Thus, \( Diff(\mathcal{M}, \mathcal{M}') = (\emptyset, \{(s_0, \text{succ}^1(s_0)), (s_0, \text{succ}^2(s_0))\}, \{\text{succ}^3(s_0), s^*\}, \emptyset, \{(\text{succ}^4(s_0), \text{succ}^5(s_0), \text{succ}^6(s_0))\}) \). If we change the number of applications of each primitive update, such as applying PU3 to two successors, PU2 to two successors and PU5 to two successors of \( s_0 \), then, the new \( Diff(\mathcal{M}, \mathcal{M}') \) is not compatible with the previous one. We cannot find any other update which is contained in the updates in case 1. Thus, these updates in case 1 are minimal changes. In case 2, \( s^* \) is the only successor of \( s_0 \) after update. Thus, the new updated CTL
model satisfies \( \text{AX}\phi \). PU4 is applied once which is not compatible with case 1. If we applied PU4 together with any other primitive updates, the new \( \text{Diff}(M,M') \) will be either not compatible with PU4 or contains PU4. Thus, case 2 is a minimal change. Case 3 is analogous to case 2 and is also minimal change. \( \square \)

**Theorem 3.3** Let \( M = (S,R,L) \) be a Kripke model and \( M = (M,s_0) \not\models \text{EF}\phi \), where \( s_0 \in S \) and \( \phi \) is a propositional formula. Then an admissible updated model \( M' = \text{Update}(M,\text{EF}\phi) \) can be obtained by one of the following results:

1. PU3 is applied to substitute any state \( s_i \in S \) reachable from \( s_0 \) once with a new state \( s^* \models \phi \) and \( \text{Diff}(s_i, s^*) \) is minimal;

2. PU4 is applied to add a new state \( s^* \models \phi \) next to any state \( s_i \) reachable from \( s_0 \) once.

**Proof.** For Theorem 3.3, after the updates in case 1 and case 2, a state \( s^* \) in the new updated model \( M' \) satisfies \( \phi \). Thus, \( M' \models \text{EF}\phi \). In case 1, after PU3 is applied once, \( \text{Diff}(M,M') = (\emptyset, \emptyset, \{s_i, s^*\}, \emptyset, \emptyset) \); in case 2, after PU4 is applied once, \( \text{Diff}(M,M') = (\emptyset, \emptyset, \emptyset, \{s^*\}, \emptyset) \). Both \( \text{Diff}(M,M') \) are not compatible. Also, either update together with other primitive updates could yield other \( \text{Diff}(M,M') \) which will be either not compatible with the two in both cases or contain the two cases, thus, PU3 in case 1 and PU4 in case 2 are minimal change. \( \square \)

**Theorem 3.4** Let \( M = (S,R,L) \) be a Kripke model, and \( M = (M,s_0) \not\models \text{AF}\phi \), where \( s_0 \in S \) and \( \phi \) is a propositional formula. Then an admissible updated model \( M' = \text{Update}(M,\text{AF}\phi) \) can be obtained by the following results: for each path starting from \( s_0 \): \( \pi = [s_0, \ldots, s_i, \ldots] \):

1. if \( \not\exists s \models \phi \), where \( s \in S \), then, PU3 is applied to a state \( s_i \) once on a path \( \pi \) to substitute a state \( s^* \models \phi \) and \( \text{Diff}(s_i, s^*) \) is minimal, then

2. for each path \( \pi \), where \( \forall s_i \in \pi \), such that \( s_i \not\models \phi \):

   either PU3 is applied to any state \( s_i \) once to substitute a state \( s^* \models \phi \) and \( \text{Diff}(s_i, s^*) \) is minimal, or
PU2 is applied once to delete relation \((s_0, \text{succ}(s_0))\), or
PU5 is applied once to delete state \(\text{succ}(s_0)\).

In Theorem 3.4, case 1 is for protecting the structure of the original model during update. If none of the states in a model satisfies \(\phi\), then, PU3 should be applied to at least one path, thus, the original model has at least one path preserved after update. Then, other paths can be updated either by PU3, PU2 or PU5. PU2 or PU5 in case 2 delete relations or states, thus some of the reachable states in the original model may be lost. However, because we have case 1 to protect the basic structure of the original model, PU2 and PU5 can be applied due to their simplicity. After these updates, each path contains at least one updated state which satisfies \(\phi\). Thus, the updated model \(\mathcal{M}' \models AF\phi\). The updates in this theorem are all minimal changes. The method of proof is similar as case 1 in Theorem 3.2, so, we do not repeat the proof here.

**Theorem 3.5** Let \(M = (S, R, L)\) be a Kripke model, \(\mathcal{M} = (M, s_0) \not\models EG\phi\), where \(s_0 \in S\) and \(\phi\) is a propositional formula. Then an admissible updated model \(\mathcal{M}' = Update(\mathcal{M}, EG\phi)\) can be obtained by the following: Select a path \(\pi = [s_0, s_1, \ldots, s_i, \ldots, s_j, \ldots]\) from \(M\) which contains minimal number of states not satisfying \(\phi\), and then

1. if for all \(s' \in \pi\) such that \(s' \not\models \phi\), there exist \(s_i, s_j \in \pi\) satisfying \(s_i < s' < s_j\) and \(\forall s \leq s_i\) or \(\forall s \geq s_j\), \(s \models \phi\), then PU1 is applied to add a relation \((s_i, s_j)\), or PU4 is applied to add a state \(s^* \models \phi\) and new relations \((s_i, s^*)\) and \((s^*, s_j)\);

2. \(\exists s_i \in \pi\), such that \(\forall s \leq s_i\), \(s \models \phi\); \(\exists s_k \in \pi''\), where \(\pi'' = [s_0, \ldots, s_k, \ldots]\), such that \(\forall s \geq s_k\), \(s \models \phi\), then PU1 is applied to connect \(s_i\) and \(s_k\);

3. if \(\exists s_i \in \pi\) \((i > 1)\) such that for all \(s' < s_i\), \(s' \models \phi\), \(s_i \not\models \phi\), then,
   a. PU1 is applied to connect \(s_{i-1}\) and one of such \(s'\) to form a new transition \((s_{i-1}, s')\);
   b. if \(s_i\) is the only successor of \(s_{i-1}\), then PU2 is applied to remove relation \((s_{i-1}, s_i)\), or PU5 is applied to remove state \(s_i\) and its associated relations;
4. if $\exists s' \in \pi$, such that $s' \not\models \phi$, then PU3 is applied to substitute all $s'$ with new state $s^* \models \phi$ and $\text{Diff}(s, s^*)$ to be minimal.

Proof. In case 1, without loss of generality, we assume for the selected path $\pi$, there exist states $s'$ that do not satisfy $\phi$, and all other states in $\pi$ satisfy $\phi$. We also assume that such $s'$ are in the middle of path $\pi$. Therefore, there are two other states $s_i, s_j$ in $\pi$ such that $s_i < s' < s_j$. That is, $\pi = [s_0, \ldots, s_{i-1}, s_i, \ldots, s', \ldots, s_j, s_{j+1}, \ldots]$. We first consider applying PU1. It is clear that by applying PU1 to add a new relation $\pi$ state in $M$ we consider applying PU4. In this case, we will have a new model $\pi'$ that satisfies $\text{EG} \pi$. After PU1 is applied, $\text{Diff}(M, M') = ((s_i, s_j), \emptyset, \emptyset, \emptyset, \emptyset)$, which implies $M'$ must be a minimally changed model with respect to $\leq_M$ that satisfies $\text{EG} \phi$. Now we consider applying PU4. In this case, we will have a new model $M' = (S \cup \{s^*\}, R \cup \{(s_i, s^*), (s^*, s_j)\}, L')$ where $L'$ is an extension of $L$ on new state $s^*$ that satisfies $\phi$. We can see that $\pi' = [s_0, \ldots, s_i, s^*, s_j, \ldots]$ is a path in $M'$ which shares all states with path $\pi$ except the state $s^*$ in $\pi'$ and those states between $s_{i+1}$ and $s_{j-1}$ including $s'$ in $\pi$. So we also have $(M', s_0) \models \text{EG} \phi$. On the other hand, we have $\text{Diff}(M, M') = (\emptyset, \emptyset, \emptyset, \{s^*\}, \emptyset)$. Obviously, $M'$ is a minimally changed model with respect to $\leq_M$ that satisfies $\text{EG} \phi$.

In case 2, if all states on the first part of a path satisfy $\phi$, and all states on the last part of another path satisfy $\phi$, then, PU1 is applied to connect a new transition $(s_i, s_k)$ which connects the first part of one path and the last part of the other path. Now, all states on the new path $[s_0, \ldots, s_i, s_k, \ldots]$ satisfy $\phi$. Thus, $M' \models \text{EG} \phi$. After PU1 is applied, $\text{Diff}(M, M') = ((s_i, s_k), \emptyset, \emptyset, \emptyset) = \text{Diff}(M, M') = ((s_{i-1}, s'), \emptyset, \emptyset, \emptyset)$ is minimum and $M'$ is a minimally changed model with respect to $\leq_M$ that satisfies $\text{EG} \phi$.

In case 3, (a) if PU1 is applied to form a new transition $(s_{i-1}, s')$, then, the new path $[s_0, \ldots, s', s_{i-1}, s', \ldots, s_{i-1}, s', \ldots]$ contains a Strongly Connected Component (SCC) [16, 50] where all states satisfy $\phi$ and $\text{Diff}(M, M') = ((s_{i-1}, s'), \emptyset, \emptyset, \emptyset, \emptyset)$ is minimum. Thus, $M'$ is a minimally changed model with respect to $\leq_M$ that satisfies $\text{EG} \phi$. 
§3.1 Semantic Characterizations

(b). If PU2 is applied, then, a new path $\pi'$ containing $[s_0, \ldots, s', \ldots, s_{i-1}]$ is derived where all states satisfy $\phi$ and $\text{Diff}(\mathcal{M}, \mathcal{M}') = (\emptyset, \{(s_{i-1}, s_i)\}, \emptyset, \emptyset, \emptyset)$ is minimal. Obviously, $\mathcal{M}'$ is a minimally changed model with respect to $\leq_M$ that satisfies $\text{EG}\phi$. If PU5 is applied, then, a same new path $\pi'$ is formed and $\text{Diff}(\mathcal{M}, \mathcal{M}') = (\emptyset, \emptyset, \emptyset, \emptyset, \{s_i\})$ is minimum. Obviously, $\mathcal{M}'$ is a minimally changed model with respect to $\leq_M$ that satisfies $\text{EG}\phi$.

In case 4, suppose there are $n$ states on a path that do not satisfy $\phi$, after PU3 is applied to all these states, $\text{Diff}(\mathcal{M}, \mathcal{M}') = (\emptyset, \emptyset, \{s'_1, s'_2, \ldots, s'_{i-1}, s^*_1, s^*_2, \ldots, s^*_n\}, \emptyset, \emptyset)$. $\text{Diff}(\mathcal{M}, \mathcal{M}')$ in this case is not compatible with case 1, case 2 or case 3. Thus, $\mathcal{M}'$ is a minimally changed model with respect to $\leq_M$ that satisfies $\text{EG}\phi$. $\square$

Theorem 3.5 characterizes four typical situations for the update with formula $\text{EG}\phi$. Basically, this theorem says that in order to make formula $\text{EG}\phi$ be true, we first select a path, then we can either make a new path based on this path so that all states in the new path satisfy $\phi$ (i.e., Case 1, Case 2 and Case 3(a)), or trim the path from the state where all previous states satisfy $\phi$ (i.e., Case 3(b)), if the previous state has only this state as its successor; or simply substitute all states not satisfying $\phi$ in the path with new states satisfying $\phi$ (i.e., Case 4). Our proof shows that resulting models from these operations are admissible.

**Theorem 3.6** Let $M = (S, R, L)$ be a Kripke model and $\mathcal{M} = (M, s_0) \not\models \text{AG}\phi$, where $s_0 \in S$ and $\phi$ is a propositional formula. Then an admissible updated model $\mathcal{M}' = \text{Update}(\mathcal{M}, \text{AG}\phi)$ can be obtained by the following: for each path starting from $s_0$: $\pi = [s_0, \ldots, s_i, \ldots]$:

1. if for all $s < s_i$ in $\pi$, $s \models \phi$ but $s_i \not\models \phi$, PU2 is applied to $s_i$ to remove relation $(s_{i-1}, s_i)$, or PU5 is applied to $s_i$ to remove $s_i$ and its associated relations, or
2. PU3 is applied to all states $s$ in $\pi$ not satisfying $\phi$ to substitute $s$ with $s^* \models \phi$ and $\text{Diff}(s, s^*)$ to be minimal.

In Theorem 3.6, Case 1 considers a special form of path $\pi$ where the first $i$ states starting from $s_0$ already satisfy formula $\phi$. Under this situation, we can simply cut off the path (i.e. applying PU2 or PU5 one time) to disconnect all other states not
satisfying \( \phi \). Case 2 is straightforward. The proofs are similar to case 3 and 4 in theorem 3.5, and are not repeated here.

**Theorem 3.7** Let \( M = (S, R, L) \) be a Kripke model and \( M = (M, s_0) \not\models E[\phi \land \psi] \), where \( s_0 \in S \), \( \phi \) and \( \psi \) are propositional formulae. Then an admissible updated model \( M' = \text{Update}(M, E[\phi \land \psi]) \) can be obtained by the following results: Select a path \( \pi = [s_0, s_1, \cdots, s_i, \cdots, s_j, \cdots] \) from \( M \) which contains minimal number of states not satisfying \( \phi \), and then

1. if \( s_0 \not\models \phi \) on \( \pi \), PU3 is applied to substitute a new state \( s'_0 \models \phi \) on \( s_0 \), then,

2. if \( \forall s \leq s_i, s_i = \phi \) and \( \forall s \geq s_j, s_j \models \phi \lor \psi \) on \( \pi \), \( \exists s' \in \pi \), such that \( s' \not\models \phi \lor \psi \) implies \( s_i < s' < s_j \), then,

   - PU1 is applied to add a transition \( (s_i, s_j) \), or
   - PU4 is applied to add a new state \( s^* \models \phi \) between \( s_i \) and \( s_j \), or
   - if \( \text{Diff}(s_{i+1}, s^*) \leq \text{Diff}(s_{i+1}, s'') \), where \( s^* \models \psi \) and \( s'' \models \phi \), then,
     - PU3 is applied to substitute a new state \( s^* \models \psi \) on the first \( s' \not\models \phi \lor \psi \), else
     - PU3 is applied to all \( s' \) with new states \( s'' \models \phi \), or

3. if \( \exists s_j \models \psi \) on \( \pi \), \( \forall s \leq s_i, s_i = \phi \), \( s_{i+1} \not\models \phi \), then,

   - PU3 is applied to substitute a new state \( s^* \models \psi \) on \( s_{i+1} \), or
   - PU4 is applied to add a new state \( s^* \models \psi \) after \( s_i \), or

   if \( \exists s \models \psi \), where \( s \in S \) and \( s \not\in \pi \), then, PU1 is applied to connect \( (s_i, s) \).

*Proof.* In Theorem 3.7, case 1 is for protecting the initial state with update PU3 if the initial state does not satisfy \( \phi \). In case 2, PU1 and PU4 updates have a similar principle to case 1 in theorem 3.5, and are not repeated here. We explain update PU3 in details. If PU3 substitutes \( s^* \models \psi \), then, \( A = \text{Diff}(\mathcal{M}, \mathcal{M}') = (\emptyset, \emptyset, \{s'_{i+1}, s^*\}, \emptyset, \emptyset) \);

If PU3 substitutes \( s'' \models \phi \) and we suppose there are 3 false states which need to be substituted, then, \( B = \text{Diff}(\mathcal{M}, \mathcal{M}') = (\emptyset, \emptyset, \{s'_{i+1}, s'_{i+2}, s'_{i+3}, s''_1, s''_2, s''_3\}, \emptyset, \emptyset) \).

If \( \text{Diff}(s_{i+1}, s^*) \leq \text{Diff}(s_{i+1}, s'') \) which is \( \text{Diff}(s'_{i+1}, s^*) \leq \text{Diff}(s'_{i+1}, s'') \) in \( A \) and \( B \), then \( A \) is a more minimally changed update. Otherwise, \( A \) and \( B \) are not
compatible and both are minimal changes. Thus, $\mathcal{M}'$ is an admissible model with respect to $\leq_M$ that satisfies $E[φUψ]$. For case 3, PU3 results in $C = Diff(\mathcal{M}, \mathcal{M}') = (\emptyset, \emptyset, \{s_{i+1}, s^*\}, \emptyset, \emptyset)$, PU4 results in $D = Diff(\mathcal{M}, \mathcal{M}') = (\emptyset, \emptyset, \emptyset, \{s^*\}, \emptyset)$, and PU1 results in $E = Diff(\mathcal{M}, \mathcal{M}') = (\{(s_i, s)\}, \emptyset, \emptyset, \emptyset, \emptyset)$. Each update PU3, PU4 and PU1 are not compatible with each other. If any other update is applied together with each of these, the new $Diff(\mathcal{M}, \mathcal{M}')$ derived from the combined update will be either not compatible with $C$, $D$ or $E$, or contains $C$, $D$ or $E$. For example, we first use PU3 to update $s_{i+1}$ with a new state $s'' \models φ$ or PU4 to add a new state $s'' \models φ$ after $s_i$, then, apply PU3 or PU4 with a new state $s^* \models ψ$ after $s'' \models φ$, or PU1 to connect states $s''$ and $s$, where $s \models ψ$ and $s \in S$. These updates contain each update from case 3 and yield less minimally changed models. Thus, each update in case 3 is a minimal change and yields $\mathcal{M}'$ which is an admissible model with respect to $\leq_M$. □

**Theorem 3.8** Let $\mathcal{M} = (S, R, L)$ be a Kripke model and $\mathcal{M} = (M, s_0) \not\models A[φUψ]$, where $s_0 \in S$ and $φ$ and $ψ$ are propositional formulas. Then an admissible updated model $\mathcal{M}' = Update(\mathcal{M}, A[φUψ])$ can be obtained by the following: for each false path starting from $s_0$, $π = [s_0, \cdots, s_i, \cdots, s_j, \cdots]$:  

1. if $s_0 \not\models φ$, then, PU3 is applied to $s_0$ to substitute a new state $s'_0 \models φ$, then,

2. if $∀s \leq s_i, s \models φ$, $s' \not\models φ \lor ψ$, $∃s_j \models ψ \lor φ$, where $s_i < s' < s_j$, then

   if $Diff(s_{i+1}, s^*) \leq Diff(s_{i+1}, s'')$, where $s^* \models ψ$ and $s'' \models φ$,

   then, PU3 is applied to substitute a new state $s*$ on $s_{i+1}$,

   else,

   $PU3$ is applied to substitute new states $s'' \models φ$ on all $s'$;

3. if $∀s_i \models φ$, $∀s_j \models ψ$ and $∃s_j \models φ \lor ¬ψ$, then

   $PU3$ is applied to substitute a new state $s^* \models ψ$ on $s_{i+1}$.

4. If $\mathcal{M} \models E[φUψ]$ and $\mathcal{M} \not\models A[φUψ]$, then

   $PU2$ is applied to remove transition $(s_0, succ(s_0))$ or

   $PU5$ is applied to remove state $succ(s_0)$. 
We observe that if $s_0 \models \phi$ in model $M$, then Theorem 3.8 can be viewed as a complete characterization for the update with formula $A[\phi U \psi]$. In the case where $s_0 \not\models \phi$, we should first substitute $s_0$ with a new initial state $s_0'$ to satisfy $\phi$ (i.e. applying PU3 one time on $s_0$) which is not only for protecting the structure of the original model but also for making the other cases possible. Then, we perform updates under different conditions. Case 2 and 3 are a subset of those in theorem 3.7. Case 2 in Theorem 3.8 is the same as case 2 in Theorem 3.7 excluding updates with PU1 and PU4; Case 3 in Theorem 3.8 is the same as case 3 in Theorem 3.7 excluding updates with PU4 and PU1. Thus, we have not repeated the proof of minimal update here. Case 4 is to cut off a false path at the successor of the initial state or the transition from the initial state to its successor, if an original model or an intermediate updated model satisfies $E \phi U \psi$ before it satisfies $A \phi U \psi$. For this case, the proof is similar to case 3 in Theorem 3.5, and is not repeated here.

Note that our characterizations deal with simple cases and rule out the case that $\phi$ is a nested modality.

### 3.2 Computational Properties

In this section, we study the computational complexity of our approach for CTL model update. We first present the following general result.

**Theorem 3.9** Given two CTL Kripke models $M = (S, R, L)$ and $M' = (S', R', L')$, where $s_0 \in S$ and $s_0' \in S'$, and a CTL formula $\phi$. Deciding whether $(M', s_0')$ is an admissible result of $Update((M, s_0), \phi)$ to satisfy $\phi$ is co-NP-complete. The hardness holds even if $\phi$ is of the form $EX \psi$ where $\psi$ is a propositional formula.

**Proof.** Membership proof. First from [16], we know that checking whether $(M', s_0') \models \phi$ can be done in time $O(|\phi| \cdot (|S| + |R|))$. In order to check whether $(M', s_0')$ is an admissible update result, we need to check whether $M'$ is minimal with respect to ordering $\leq_M$. To do so, we consider the complement of the problem. That is to check whether $M'$ is not a minimal model. Therefore, we do two things: (1) proposing a CTL model $M'' = (S'', R'', L'')$ satisfying $\phi$ for some $s'' \in S''$; and (2) testing whether $M'' <_M M'$. Step (1) can be done in polynomial time. To
check $M'' \prec_M M'$, we first compute $Diff(S, S')$, $Diff(S, S'')$, $Diff(R, R')$ and $Diff(R, R'')$. All these can be computed in polynomial time. Then according to these sets, we identify, through PU1 to PU5, $Diff_{PUi}(M, M')$ and $Diff_{PUi}(M, M'')$ ($i = 1, \cdots, 5$). Again, these can also be done in polynomial time. Finally, by checking $Diff_{PUi}(M, M'') \subseteq Diff_{PUi}(M, M')$ ($i = 1, \cdots, 5$) we can decide whether $M'' \prec_M M'$. So both steps (1) and (2) can be achieved in polynomial time with a non-deterministic Turing machine.

Hardness proof. It is well known that the validity problem for a propositional formula is co-NP-complete. Given a propositional formula $\phi$, we construct in polynomial time a transformation from the problem of deciding $\phi$’s validity to a CTL model update. Let $X$ be the set of all variables occurring in $\phi$, and $a, b$ two new variables not occurring in $X$. We denote $\neg X = \bigwedge_{x_i \in X} \neg x_i$. We specify a CTL Kripke model based on the variable set $X \cup \{a, b\}$: $M = (\{s_0, s_1\}, \{(s_0, s_1), (s_1, s_1)\}, L)$, where $L(s_0) = \emptyset$ (i.e. all variables are assigned false), $L(s_1) = X$ (i.e. variables in $X$ are assigned true, while $a, b$ are assigned false). Now we define a new formula $\mu = EX(((\phi \supset a) \land (\neg X \land b)) \lor (\neg \phi \land a))$. Clearly, formula $((\phi \supset a) \land (\neg X \land b)) \lor (\neg \phi \land a)$ is satisfiable and $s_1 \not\models ((\phi \supset a) \land (\neg X \land b)) \lor (\neg \phi \land a)$. So $(M, s_0) \not\models \mu$. Consider the update $Update((M, s_0), \mu)$. We define $M' = (\{s'_0, s'_1\}, \{(s'_0, s'_1), (s'_1, s'_1)\}, L')$, where $L'(s'_0) = L(s_0)$ and $L'(s'_1) = \{a, b\}$. Now we show that $\phi$ is valid iff $(M', s'_0)$ is an admissible update result from $Update((M, s_0), \mu)$.

Case 1. We show that if $\phi$ is valid, then $(M', s'_0)$ is an admissible update result from $Update((M, s_0), \mu)$. Since $\phi$ is valid, we have $\neg X \models \phi$. So we have $s'_1 \models (\phi \supset a) \land (\neg X \supset b))$. This follows $(M', s'_0) \models \mu$. Also note that $M'$ is obtained by applying PU3 to substitute $s_1$ with $s'_1$. $Diff(s_1, s'_1) = X \cup \{a, b\}$ which presents a minimal change from $s_1$ in order to satisfy $(\phi \supset a) \land (\neg X \land b)$.

Case 2. Suppose that $\phi$ is not valid. Then there exists $X_1 \subseteq X$ such that $X_1 \models \neg \phi$. We construct $M'' = (\{s''_0, s''_1\}, \{(s''_0, s''_1), (s''_1, s''_1)\}, L'')$, where $L''(s''_0) = L(s_0)$ and $L''(s''_1) = X_1 \cup \{a\}$. It is easy to see that $s''_1 \models (\neg \phi \land a)$, and hence $(M'', s''_0) \models \mu$. Now we show that $(M', s'_0) \models \mu$ implies $M'' \prec_M M'$. Suppose $(M', s'_0) \models \mu$. Clearly, both $M'$ and $M''$ are obtained from $M$ by applying PU3 one time on $s_1$ respectively. But we have $Diff(s_1, s'_1) = (X - X_1) \cup \{a\} \subset X \cup \{a, b\} = Diff(s, s'_1)$. This leads
to the conclusion that \((M', s'_0)\) is not an admissible update model. 

Theorem 3.9 implies that it is probably not feasible to develop a polynomial time algorithm to implement our CTL model update. Indeed, our algorithm described in the next section, generally runs in exponential time. Nevertheless, we can identify certain typical update cases that may be achieved in polynomial time.

**Theorem 3.10** Let \(M = (S, R, L)\) be a CTL Kripke model and \(\phi\) a CTL formula. If an admissible update result \(\text{Update}((M, s_0), \phi)\) \((s_0 \in S)\) can be obtained by only applying PU1, PU2, PU4 and PU5 without path selection in \(M\) (i.e. \(\phi\) does not involve \(\text{EG}\psi\), \(\text{EF}\psi\) or \(\text{E}[\psi_1\text{U}\psi_2]\)), then this result can be computed in polynomial time.

**Proof.** From the algorithm \(\text{CTL*UPDATE}(\mathcal{M}, \phi)\) in Chapter 4, we can see that if an admissible model can be obtained by only using PU1, PU2, PU4 and PU5 without path selection, then \(\text{CTL*UPDATE}(\mathcal{M}, \phi)\) will at most visit all states in \(\mathcal{M}\) polynomial times, and each time, operations PU1, PU2, PU4 and PU5 can be completed by just removing and/or adding states and/or relations. This means the resulting model can be computed in polynomial time. 

\(\square\)
Chapter 4

Algorithms for CTL Model Update

4.1 Algorithms

The purpose of CTL model update and its minimal change is to design algorithms which can be used in practice for the CTL model updater. The major tasks for the CTL update algorithms are to direct the coding of the principles of characterization as described in the previous chapter based on our minimal change rules in Chapter 2. For example, the characterization for EF to achieve minimal change is to perform either PU3 or PU4 once on any path of a model as described in the function UPDATE$_{EF}$. If PU3 is performed, the function UPDATE$_{EF}$ calls the main function CTL*UPDATE to reach cases such as “$\phi$ is $\phi_1 \land \phi_2$” where $\phi$ is not atomic, or reach function UPDATE$_p$ for the case “$\phi$ is $p$”, which is performed as one case of the atomic operation PU3, or call the function UPDATE$_{\neg}$, to access the case “$\phi$ is $\neg p$” where $\phi$ is another case of atomic formula for PU3. If PU4 is performed in the function UPDATE$_{EF}$, the updated model is described in this function itself by adding a state into a path which satisfies $\phi$. To achieve minimal change update in the characterization for AG in the function UPDATE$_{AG}$, either PU2, PU5 or PU3 is applied to each path for different circumstances. In this function, PU2 and PU5 are for deleting the relation or state succeeding the initial state respectively; PU3 is for substituting all states in a path which do not satisfy the required properties.

We have designed the algorithms in the form of pseudo-code to be implemented with recursive call usage and to be compatible with the SAT function which implements the labeling algorithms for CTL model checking as presented in [50, 16].
recursive call usage allows the checked property $\phi$ range from the nested modality to the atomic propositional formula. The compatibility makes our algorithms more standard and portable for later integration of our CTL model updater into the existing SMV model checker. We first present the main function:

function \text{CTL*UPDATE}(M, \phi) \quad /* \quad M \neq \phi. Update } M \text{ to satisfy } \phi. */

INPUT \quad M = (S, R, L), M = (M, s_0), where $s_0 \in S$ and $M \not\models \phi$;

OUTPUT \quad M' = (S', R', L'), M' = (M', s'_0), where $s'_0 \in S'$ and $M' \models \phi'$;

{ 

\text{case}

\phi \text{ is } \bot : \text{return}\{M\};

\phi \text{ is atomic } p : \text{return} \{\text{UPDATE}_p(M, p)\};

\phi \text{ is } \neg \phi_1 : \text{return} \{\text{UPDATE}_{\neg}(M, \phi_1)\};

\phi \text{ is } \phi_1 \lor \phi_2 : \text{return}\{\text{CTL*UPDATE}(M, \phi_1) \text{ or } \text{CTL*UPDATE}(M, \phi_2)\};

\phi \text{ is } \phi_1 \land \phi_2 : \text{return} \{\text{UPDATE}_{\land}(M, \phi_1, \phi_2)\};

\phi \text{ is } \text{EX}\phi_1 : \text{return} \{\text{UPDATE}_{\text{EX}}(M, \phi_1)\};

\phi \text{ is } \text{AX}\phi_1 : \text{return} \{\text{UPDATE}_{\text{AX}}(M, \phi_1)\};

\phi \text{ is } \text{EF}\phi_1 : \text{return} \{\text{UPDATE}_{\text{EF}}(M, \phi_1)\};

\phi \text{ is } \text{AF}\phi_1 : \text{return} \{\text{UPDATE}_{\text{AF}}(M, \phi_1)\};

\phi \text{ is } \text{EG}\phi_1 : \text{return} \{\text{UPDATE}_{\text{EG}}(M, \phi_1)\};

\phi \text{ is } \text{AG}\phi_1 : \text{return} \{\text{UPDATE}_{\text{AG}}(M, \phi_1)\};

\phi \text{ is } \text{E}(\phi_1 U \phi_2) : \text{return} \{\text{UPDATE}_{EU}(M, \phi_1, \phi_2)\};

\phi \text{ is } \text{A}(\phi_1 U \phi_2) : \text{return} \{\text{UPDATE}_{AU}(M, \phi_1, \phi_2)\};

}

The main function \text{CTL*UPDATE}(M, \phi) is organized according to Definition 2.2. The main function is a launch point for the whole set of algorithms. Subfunctions are called in the main function. These subfunctions then call either themselves or the main function to access other subfunctions. Before the specification properties are satisfied, the intermediate level subfunctions always call the main function or themselves as recursive calls. No matter how many recursive calls are executed, the final
functions to perform the atomic update are either the function $\text{UPDATE}_p(\mathcal{M}, \phi)$ for the case of "$\phi$ is atomic $p$", or the function $\text{UPDATE}_-(\mathcal{M}, \neg \phi)$ for the case of "$\neg \phi$ is $\neg p$", where both cases are atomic PU3, or the subfunctions themselves if PU1, PU2, PU4 or PU5 operations are performed in these functions.

function $\text{UPDATE}_p(\mathcal{M}, p)$ /* $\mathcal{M} \models p$. Update $s_0$ to satisfy $p$. */
{
1. $s'_0 := s_0 \cup \{p\}$;
   PU3 is applied to substitute $s_0 \not\models p$ with $s'_0 \models p$, and $\text{Diff}(s_0, s'_0) = p$;
2. $S' := S - \{s_0\} \cup \{s'_0\}$;
3. $R' := R - \{(s_0, s_i) \mid s_i = \text{succ}(s_0)\} \cup \{(s'_0, s_i) \mid s_i = \text{succ}(s_0)\}$
   $\cup \{(s_j, s_0) \mid s_j = \text{pre}(s_0)\} \cup \{(s_j, s'_0) \mid s_j = \text{pre}(s_0)\}$;
4. $L': S' \rightarrow 2^{AP}$, where $\forall s \in S'$, if $s \in S$, then $L'(s) = L(s)$,
   else $s = s'_0$, and
   $L'(s'_0)$ is a set of true variables assigned in $s'_0$;
5. $\mathcal{M}' := (M', s'_0)$, where $M' = (S', R', L')$ and $s'_0 \in S'$;
6. return $\{\mathcal{M}'\}$;
}

A state is a snapshot or instantaneous description of the system that captures the values of the variables at a particular instant of time [16]. The update is eventually finalized in the state or its transitions. The function $\text{UPDATE}_p(\mathcal{M}, p)$ is a lowest level call in the whole set of algorithms and contains the update of the smallest element, the propositional atom. This function is one case of the usage of the atomic operation PU3. After the application of atomic update PU3, the updated state $s'_0$ and three tuples of $\mathcal{M}'$: $S'$, $R'$ and $L'$ are assigned. The new updated state $s'_0$ contains propositional atoms which are the union of the original propositional atoms, for example, $s_0 = \{a, b, c, \cdots\}$, where $s_0 \not\models p$, and a new propositional atom $p$. Thus, $s'_0 \models p$. The new set of states $S'$ contain the original set of states $S$ in $\mathcal{M}$ with the new state $s'_0$, but excluding the old state $s_0$. The relations (transitions) of $\mathcal{M}'$ are the original set of relations $R$ in $\mathcal{M}$ plus the incoming and outgoing relations of the new state $s'_0$, and excluding the original incoming and outgoing relations of
the original state \( s_0 \). If there are more than 2 attached relations to the old or new states, there could also be more than 2 added or removed relations. These new assignments form a new updated model \( \mathcal{M}' \), which is returned by the function \( \text{UPDATE}_p(\mathcal{M},p) \). The update is shown in Figure 2.4 where the model on the left is updated to the model on the right by adding atom \( p \) to states \( s_1 \) and \( s_2 \), creating the updated states \( s'_1 \) and \( s'_2 \). A similar principle is applied to the case “\( \phi \) is \( \neg p \)” but this time the process is reverse, and is implemented in the function \( \text{UPDATE}_-(\mathcal{M},\phi) \). This is another case of the usage of the atomic operation PU3. In Figure 2.4, the corresponding update is the update from the model on the right to the model on the left to remove atom \( p \) from states \( s'_1 \) and \( s'_2 \).

function \( \text{UPDATE}_-(\mathcal{M},\phi) \) /* \( \mathcal{M} \not\models \phi \). Update \( \mathcal{M} \) to satisfy \( \phi \). */
{

    case

    \( \phi \) is \( \neg p \):

    1. \( s'_0 := s_0 - \{p\} \);

        PU3 is applied to substitute \( s_0 \not\models \neg p \) with \( s'_0 \models \neg p \), and \( \text{Diff}(s_0, s'_0) = p \):

    2. 6. are the same as those in the function \( \text{UPDATE}_p(\mathcal{M},p) \);

    \( \phi \) is \( \neg(\phi_1 \lor \phi_2) = \neg\phi_1 \land \neg\phi_2 \): return \{UPDATE\( _\land(\mathcal{M},\neg\phi_1,\neg\phi_2) \)\};

    \( \phi \) is \( \neg(\phi_1 \land \phi_2) = \neg\phi_1 \lor \neg\phi_2 \): return \{UPDATE\( _-(\mathcal{M},\phi_1) \) or UPDATE\( _-(\mathcal{M},\phi_2) \)\};

    \( \phi \) is \( \neg\text{EX}(\phi_1) = \text{AX}(\neg\phi_1) \): return \{UPDATE\( _{\text{AX}}(\mathcal{M},\neg\phi_1) \) \};

    \( \phi \) is \( \neg\text{AX}(\phi_1) = \text{EX}(\neg\phi_1) \): return \{UPDATE\( _{\text{EX}}(\mathcal{M},\neg\phi_1) \) \};

    \( \phi \) is \( \neg\text{EF}(\phi_1) = \text{AG}(\neg\phi_1) \): return \{UPDATE\( _{\text{AG}}(\mathcal{M},\neg\phi_1) \) \};

    \( \phi \) is \( \neg\text{AF}(\phi_1) = \text{EG}(\neg\phi_1) \): return \{UPDATE\( _{\text{EG}}(\mathcal{M},\neg\phi_1) \) \};

    \( \phi \) is \( \neg\text{EG}(\phi_1) = \text{AF}(\neg\phi_1) \): return \{UPDATE\( _{\text{AF}}(\mathcal{M},\neg\phi_1) \) \};

    \( \phi \) is \( \neg\text{AG}(\phi_1) = \text{EF}(\neg\phi_1) \): return \{UPDATE\( _{\text{EF}}(\mathcal{M},\neg\phi_1) \) \};

    \( \phi \) is \( \neg\text{E}(\phi_1 \cup \phi_2) = \text{E}[\neg\phi_2 \cup (\neg\phi_1 \land \neg\phi_2)] \lor \text{EG}\neg\phi_2 \):

        return \{UPDATE\( _{\text{EU}}(\mathcal{M},\neg\phi_2,\neg\phi_1 \land \neg\phi_2) \) or UPDATE\( _{\text{EG}}(\mathcal{M},\neg\phi_2) \) \};


}

The function \( \text{UPDATE}_-(\mathcal{M},\phi) \) is actually a parsing function. Semantic conver-
sions in this function are based on the de Morgan rules described in section 2.1. The main idea of these conversions is to push the “\(^{\neg}\)” sign appearing in front of CTL operators such as “\(^{\neg}\text{EX}(\phi)\)” and “\(^{\neg}\text{EG}(\phi)\)” to after the operators such as “\(\text{EX}(\neg\phi)\)” and “\(\text{EG}(\neg\phi)\)” , then “\(\neg\phi\)” can be delivered as a parameter into the subfunctions. These subfunctions then call the main function to access other subfunctions, the function \(\text{UPDATE}_{\neg}\), or themselves to conduct recursive calls. Eventually, the atomic update for the case “\(\phi\) is \(\neg p\)” is performed to complete the update. “\(\phi\) is \(\neg\text{E}(\phi_1\text{U}\phi_2)\)” cannot be converted to the intended form using the general \(\text{UPDATE}_{\neg}\) function and must be processed individually by the subfunction \(\text{UPDATE}_{\neg-EU}\):

\[
\text{function UPDATE}_{\neg-EU}(\mathcal{M},\phi_1,\phi_2)
\]

/* \(\mathcal{M}\nmodel\neg\text{E}(\phi_1\text{U}\phi_2)\). Update \(\mathcal{M}\) to satisfy \(\neg\text{E}(\phi_1\text{U}\phi_2)\). */

\{
1. Select a path \(\pi = [s_0, s_1, \ldots, s_j, \ldots, s_i, \ldots]\),
   where \(\mathcal{M}_j = (M, s_j) \models \phi_1\) and \(\mathcal{M}_i = (M, s_i) \models \phi_2\);
2. \(\mathcal{M}' := \text{CTL}^*\text{UPDATE}(\mathcal{M}_i, \neg\phi_2)\), or
   \(\mathcal{M}' := \text{CTL}^*\text{UPDATE}(\mathcal{M}_j, \neg\phi_1)\)\(^1\);
3. If \(\mathcal{M}' \models \neg\text{E}(\phi_1\text{U}\phi_2)\), return\{\(\mathcal{M}'\)\}
   else \(\text{UPDATE}_{\neg-EU}(\mathcal{M}', \phi_1, \phi_2)\);
\}

The function \(\text{UPDATE}_{\neg-EU}\) calls the main function to update some states \(s \leq s_j, s \models \phi_1\) with “\(\neg\phi_1\)” or states starting from and following \(s_i \models \phi_2\) with “\(\neg\phi_2\)” . The number of combinations of update is not determined. For example, we can update one or more \(s_j\) with state(s) \(s^* \models \neg\phi_1\) and \(s^* \not\models \phi_2\) by applying PU3 to achieve the goal. We can also update one or more \(s_i \models \phi_2\) with \(s'' \models \neg\phi_2\) and \(s'' \not\models \phi_1\) depending on the circumstances. In both cases, the minimal change is where only one state is updated.

\[
\text{function UPDATE}_{\land}(\mathcal{M},\phi_1,\phi_2)/*/\mathcal{M} \not\models \phi_2 \land \phi_2. \text{ Update } \mathcal{M} \text{ to satisfy } \phi_1 \land \phi_2. */
\]

\(^1\)It is a nondeterministic choice.
The function \( \text{UPDATE} \land (\mathcal{M}, \phi_1, \phi_2) \) first calls the main function to update the state \( s_0 \) with \( \phi_1 \), to make \( s_0 \models \phi_1 \), then, calls the main function again to also make \( s_0 \models \phi_2 \). After these recursive calls, it calls itself again if the 2nd update affects the first update to resulting in the state not satisfying \( \phi_1 \land \phi_2 \). Otherwise, the function returns the updated model.

Intermediate functions such as \( \text{UPDATE}_{AX} \) and \( \text{UPDATE}_{EG} \) are based on the semantic characterizations in the previous chapter. These functions assign update with particular primitive operations on paths or states according to semantics of CTL. After update is assigned to states, if the delivered \( \phi \) is not an atomic formula or the update involves PU3, these functions call the main function. Then, the main function decomposes the specification property by calling itself or other functions until \( \phi \) is atomic.

function \( \text{UPDATE}_{EX}(\mathcal{M}, \phi) \) /*  \( \mathcal{M} \not\models \text{EX}\phi \). Update \( \mathcal{M} \) to satisfy \( \text{EX}\phi \). */

1. if \( \exists s_i \in S \), such that \((M, s_i) \models \phi \) and \( s_i \neq \text{succ}(s_0) \), then
   PU1 is applied to add a relation \((s_0, s_i)\):
   \( S' = S, \ R' = R \cup \{(s_0, s_i)|s_0, s_i \in S \cap S'\} \) and \( L' = L \);
   \( \mathcal{M}' = (M', s'_0) \models \text{EX}\phi \), where \( M' = (S', R', L') \) and \( s'_0 \in S' \);
   return \( \{M'\} \);
2. if \( \nexists s_i \in S \), such that \((M, s_i) \models \phi \), then
   2.1 PU3 is applied to substitute \( s_1 \) with \( s^* \models \phi \) and \( \text{Diff}(s_1, s^*) \) is minimal:
§4.1 Algorithms

a. Select a state \( s_1 = \text{succ}(s_0) \), such that \( \mathcal{M}_1 = (M, s_1) \not\models \phi \); 
b. return \{\text{CTL}^*\text{UPDATE}(\mathcal{M}_1, \phi)\}; or

2.2 PU4 is applied to add a state \( s^* = \text{succ}(s_0) \):

\[
S' = S \cup \{s^* | s^* \notin S\};
\]

\[
R' = R \cup \{(s_0, s^*) | s_0 \in S \cap S'\};
\]

\[
L' : S' \rightarrow 2^{AP}, \text{ where } \forall s \in S', \text{ if } s \in S, \text{ then } L'(s) = L(s),
\]

else \( s = s^* \), and

\[
L'(s^*) \text{ is a set of true variables assigned in } s^*;
\]

\[
\mathcal{M}' := (M', s'_0) \models \text{EX}\phi, \text{ where } M' = (S', R', L') \text{ and } s'_0 \in S';
\]

return \{\(\mathcal{M}'\)\};

The principle of the function \text{UPDATE}_{EX}(\mathcal{M}, \phi)\) is guided by Theorem 3.1 in Chapter 3. This function consists of two cases of update. PU1 is for one case and PU3 and PU4 are for the other case. If PU3 is performed, the function must call the main function to decompose “\(\phi\)” or access the atomic update functions (Same principles are applied to PU3 in other subfunctions). After one of these updates, the updated model satisfies the specification properties, so, this function does not need to call itself to update again.

function \text{UPDATE}_{AX}(\mathcal{M}, \phi) /* \mathcal{M} \not\models \text{AX}\phi. Update } \mathcal{M} \text{ to satisfy } \text{AX}\phi. */

\{
1. if \( \forall \text{succ}(s_0) \not\models \phi \), then,

PU3 is applied to substitute \( \text{succ}(s_0) \) with \( s^* \models \phi \) and \( \text{Diff}(s_1, s^*) \) is minimal:

a. Select a state \( s_1 = \text{succ}(s_0) \), such that \( \mathcal{M}_1 = (M, s_1) \not\models \phi \);

b. return \{\text{CTL}^*\text{UPDATE}(\mathcal{M}_1, \phi)\};

2. if \( \exists \text{succ}(s_0) \models \phi \), select a state \( s_1 = \text{succ}(s_0, \neg \phi) \), then

PU2 is applied to delete the relation \( (s_0, \text{succ}(s_0, \neg \phi)) \):

\[
S' = S, \ R' = R - \{(s_0, s_1) | s_0, s_1 \in S, \in S'\} \text{ and } L' = L, \text{ or}
\]

PU3 is applied to substitute \( s_1 \) with \( s^* \models \phi \) and \( \text{Diff}(s_1, s^*) \) is minimal:

a. For \( s_1 = \text{succ}(s_0, \neg \phi) \), such that \( \mathcal{M}_1 = (M, s_1) \not\models \phi \);

b. return \{\text{CTL}^*\text{UPDATE}(\mathcal{M}_1, \phi)\}, or
PU5 to delete the state \( s_1 = \text{succ}(s_0, \neg \phi) \):

\[
S' = S - \{ s_1 | s_1 \in S, \neg \phi \},
\]

\[
R' = R - \{ (s_0, s_1) | s_0 \in S, s_1 \in S' \} - \{ (s_1, \text{succ}(s_1)) | \text{succ}(s_1) \in S, s_1 \in S' \},
\]

\[
L' : S' \rightarrow 2^{AP}, \text{ where } \forall \ s \in S', \text{ also } s \in S, \text{ thus, } L'(s) = L(s);
\]

3. if \( \exists \text{succ}(s_0) \) and \( \forall s_i \not\models \phi \), where \( s_i \in S \), then

PU4 is applied to next to \( s_0 \) once to add a state \( s^* \models \phi \):

\[
S' = S \cup \{ s^* | s^* \not\in S, s^* \in S' \},
\]

\[
R' = R \cup \{ (s_0, s^*) | s^* \in S, s_1 \in S' \},
\]

\[
L' : S' \rightarrow 2^{AP}, \text{ where } \forall s \in S', \text{ if } s \in S, \text{ then } L'(s) = L(s);
\]

else \( s = s^* \), and

\( L'(s^*) \) is a set of true variables assigned in \( s^* \);

4. if \( \exists \neg \text{succ}(s_0) \) and \( \exists s_i \models \phi \), where \( s_i \in S \), then

PU1 is applied to connect \( (s_0, s_i) \) once.

\[
S' = S, \quad R' = R \cup \{ (s_0, s_i) | s_0, s_i \in S, s_i \in S' \} \quad \text{and} \quad L' = L;
\]

5. \( M' = (M', s'_0) \), where \( M' = (S', R', L') \) and \( s'_0 \in S' \);

if \( M' \models \text{AX} \phi \), then return \( \{ M' \} \); else \( \text{UPDATE}_{\text{AX}}(M', \phi) \);

The function \( \text{UPDATE}_{\text{AX}}(M, \phi) \) is an implementation of Theorem 3.2 in Chapter 3. In this function, case 1 is for protecting the structure of an updated model. Then, in case 2, every process of an execution only updates one successor of \( s_0 \). The final results should be that all successors of \( s_0 \) satisfy \( \phi \), which is the accumulation of updated successors and original true successors. Thus, after each update, it checks whether the updated model \( M' \) satisfies \( \text{AX} \phi \) or not. The condition checked is whether the accumulation of successors of \( s_0 \) satisfy \( \phi \) or not. If they do not, the function calls itself again recursively. Otherwise it returns the new updated model \( M' \). However, for special cases 3 and 4, each update is just performed once, then, the updated model satisfies “\( \text{AX} \phi \)”.

\[
\text{function UPDATE}_{\text{EF}}(M, \phi) / * M \not\models \text{EF} \phi. Update } M \text{ to satisfy } \text{EF} \phi. */ \]

\[
\}
\]

1. Select a state \( s_i \) on a path \( \pi = [s_0, s_1, \ldots] \), such that \( (M, s_i) \not\models \phi \);
2. Update the state $s_i$ with minimal change rules:

2.1. PU3 is applied to substitute $s_i$ with $s^* \models \phi$, and $Diff(s_i, s^*)$ is minimal:
return $\{\text{CTL*UPDATE}(\mathcal{M}_i, \phi)\}$; /* where $\mathcal{M}_i = (M, s_i)$ */

2.2. PU4 is applied to add a state $s^* \models \phi$ next to $s_i$:

$S' := S \cup \{s^* | s^* \notin S, s^* \in S'\}$;
$R' := R \cup \{(pre(s^*), s^*), (s^*, succ(s^*))| pre(s^*), succ(s^*) \in S, \in S'\}$;
$L' : S' \to 2^{AP}$, where $\forall s \in S'$, if $s \in S$, then $L'(s) = L(s)$,
else, $s = s^*$, and
$L'(s^*)$ is a set of true variables assigned in $s^*$;
$\mathcal{M}' := (M', s'_0) \models EF\phi$, where $M' = (S', R', L')$ and $s'_0 \in S'$;
return $\{\mathcal{M}'\}$;

The function $\text{UPDATE}_{EF}(\mathcal{M}, \phi)$ analogizes Theorem 3.3. This function contains a similar update principle to $\text{UPDATE}_{EX}$. After update of one state in a model by PU3 or PU4, the updated model satisfies the required properties and the function does not need to call itself again.

function $\text{UPDATE}_{AF}(\mathcal{M}, \phi)$ /* $\mathcal{M} \not\models AF\phi$. Update $\mathcal{M}$ to satisfy $AF\phi$. */

{  
1. if $\exists s \in S$ such that $(M, s) \models \phi$, then
PU3 is applied to a state $s_j \not\models \phi$ on a path $\pi$ to substitute a state $s'_j \models \phi$ and
Diff($s_j, s'_j$) is minimal:
return $\{\text{CTL*UPDATE}(\mathcal{M}_j, \phi)\}$, else
2. Select a path $\pi$ where $\forall s_i \in \pi$, such that $s_i \not\models \phi$;
   either PU3 is applied to a state $s_i$ to substitute a state $s^* \models \phi$ and
   Diff($s_i, s^*$) is minimal:
   return $\{\text{CTL*UPDATE}(\mathcal{M}_i, \phi)\}$, or
   PU2 is applied once to delete relation $(s_0, succ(s_0))$:  
   $S' = S$, $R' = R - \{(s_0, succ(s_0))| s_0, succ(s_0) \in S, \in S'\}$ and $L' = L$, or
   PU5 is applied once to delete state $succ(s_0)$:  
   $S' = S - \{succ(s_0)| succ(s_0) \in S, \notin S'\}$,
}
$R' = R - \{(s_0, \text{succ}(s_0))|s_0 \in S, \in S'\}$

$-\{(\text{succ}(s_0), \text{succ}(\text{succ}(s_0))|\text{succ}(\text{succ}(s_0)) \in S, \in S'\},$

$L': S' \rightarrow 2^\text{AP}$, where $\forall s \in S'$, also $s \in S$, thus, $L'(s) = L(s);$

3. $\mathcal{M}':=(\mathcal{M}', s'_0)$, where $\mathcal{M}'=(S', R', L')$ and $s'_0 \in S'$;

if $\mathcal{M}' \models \text{AF}\phi$, then return $\{\mathcal{M}'\}$; else $\text{UPDATE}_{\text{AF}}(\mathcal{M}', \phi);$

The function $\text{UPDATE}_{\text{AF}}(\mathcal{M}, \phi)$ analogizes Theorem 3.4. It contains a similar principle to $\text{UPDATE}_{\text{AX}}$. Case 1 protects the structure of an updated model to ensure that the updated model has at least one reachable path. In case 2, PU3 is for updating a state on a false path, PU2 and PU5 for deleting false paths. Each update only changes one path. After each update, the updated model may still not satisfy the specification properties. Thus, this function calls itself recursively until the updated model satisfies the required properties.

function $\text{UPDATE}_{\text{EG}}(\mathcal{M}, \phi) / *\mathcal{M} \not\models \text{EG}\phi$. Update $\mathcal{M}$ to satisfy $\text{EG}\phi$. */

{ Select a path $\pi = [s_0, s_1, \ldots, s_i, \ldots]$ in $\mathcal{M}$:

1. if $\exists s' \in \pi$ such that $s' \not\models \phi,$

there exist $s_i, s_j \in \pi$ satisfying $s_i < s' < s_j$ and $s_i \models \phi$ and $s_j \models \phi,$ then

PU1 is applied to add a relation $(s_i, s_j)$:

$S' = S, R' = R \cup \{(s_i, s_j)|s_i, s_j \in S, \in S'\}$ and $L' = L,$ or

PU4 is applied to add a state $s^* \models \phi$ and new relations $(s_i, s^*)$ and $(s^*, s_j)$:

$S' = S \cup \{s^* | s^* \not\in S, \in S'\};$

$R' = R \cup \{(s_i, s^*)|(s^*, s_j)|s_i, s_j \in S, \in S'\};$

$L': S' \rightarrow 2^\text{AP}$, where $\forall s \in S'$, if $s \in S$, then $L'(s) = L(s),$

else $s = s^*,$ and

$L'(s^*)$ is a set of true variables assigned in $s^*$;

2. if $\exists s_i \in \pi$, such that $\forall s \leq s_i, s \models \phi; \exists s_k \in \pi''$, where $\pi'' = [s_0, \ldots, s_k, \ldots],$

such that $\forall s \geq s_k, s \models \phi$, then

PU1 is applied to connect $s_i$ and $s_k$:

$S' = S, R' = R \cup \{(s_i, s_k)|s_i, s_k \in S, \in S'\}$ and $L' = L;$
3. if \( \exists s_i \in \pi (i > 1) \) such that for all \( s' < s_i, s' \models \phi, s_i \not\models \phi \), then

3.1 PU1 is applied to connect \( s_{i-1} \) and \( s' \) to form a new transition \( (s_{i-1}, s') \):
\[
S' = S, \quad R' = R \cup \{(s_{i-1}, s') \mid s_{i-1}, s' \in S, s' \in S'\} \text{ and } L' = L;
\]

3.2 If \( s_i \) is the only successor of \( s_{i-1} \), then

PU2 is applied to remove relation \( (s_{i-1}, s_i) \):
\[
S' = S, \quad R' = R - \{(s_{i-1}, s_i) \mid s_{i-1}, s_i \in S, s_i \in S'\} \text{ and } L' = L, \text{ or }
\]

PU5 is applied to remove state \( s_i \) and its associated relations:
\[
S' = S - \{s_i \mid s_i \in S, s_i \not\in S'\}, \quad R' = R - \{\{(pre(s_i), s_i) \mid pre(s_i) \in S, s_i \in S'\} - \{(s_i, succ(s_i)) \mid succ(s_i) \in S, s_i \in S'\}, \quad L' : S' \rightarrow 2^AP, \text{ where } \forall s \in S', \text{ also } s \in S, \text{ thus, } L'(s) = L(s);
\]

4. if \( \exists s_k \in \pi \), such that \( s_k \not\models \phi \), then

PU3 is applied to substitute all \( s_k \) with new state \( s^* \models \phi \) and
\[
Diff(s, s^*) \text{ to be minimal; return } \{\text{CTL*UPDATE}(M_k, \phi)\};
\]

5. \( \mathcal{M}' := (M', s'_0) \), where \( M' = (S', R', L') \) and \( s'_0 \in S' \);

if \( \mathcal{M}' \models \text{EG} \phi \), then return \( \{\mathcal{M}'\} \); else \text{UPDATE}_{\text{EG}}(\mathcal{M}', \phi);

The function \text{UPDATE}_{\text{EG}}(\mathcal{M}, \phi) \) follows the principle in Theorem 3.5. It updates all false states on one path with different update methods in 4 cases depending on the circumstances. The function \text{UPDATE}_{\text{EG}} \) calls itself repeatedly if there are still false states on a path until this false path becomes a truth path. Then, the updated model satisfies the required properties.

function \text{UPDATE}_{\text{AG}}(\mathcal{M}, \phi) \ /* \ \mathcal{M} \not\models \text{AG} \phi \). Update \mathcal{M} to satisfy \text{AG} \phi. */

\{
1. If \( \mathcal{M}_0 = (M, s_0) \not\models \phi \), then PU3 is applied to substitute \( s_0 \) with \( s'_0 \models \phi \), and \( Diff(s_0, s'_0) \) is minimal, such that
\[
\mathcal{M}' := \text{CTL*UPDATE}(\mathcal{M}_0, \phi);
\]
2. else \{ 
2.1. Select a path \( \pi = [s_0, s_1, \ldots] \), where \( \exists s_i \in \pi \) such that
\[
\mathcal{M}_i = (M, s_i) \not\models \phi;
\]

\}
2.2. select a state $s_i$, such that $\not\exists s_j < s_i$ with $(M, s_j) \not|= \phi$, then

2.2.1. PU2 is applied to delete relation $(pre(s_i), s_i)$, then

$S' := S$;

$R' := R - \{ (pre(s_i), s_i) | pre(s_i), s_i \in S, s_i \in S' \}$;

$L' := L$, since only a relation is removed;

or

2.2.2. PU5 is applied to delete $s_i$ and its associated relations, then

$S' := S - \{ s_i | s_i \in S \}$;

$R' := R - \{ (pre(s_i), s_i) | pre(s_i), s_i \in S, s_i \in S' \} - \{ (s_i, succ(s_i)) | succ(s_i) \in S, s_i \in S' \}$,

where if associated relations of $s_i$ are 2;

$L': S' \rightarrow 2AP$, since $S' \subseteq S$, $\forall s \in S'$, such that $L'(s) := L(s)$;

or

2.3. PU3 is applied to substitute any state $s_i \not|= \phi$ with $s^* |= \phi$,

and $Diff(s_i, s^*)$ is minimal:

$\mathcal{M'} := \text{CTL*UPDATE}(\mathcal{M}_i, \phi)$;

}\}

3. $\mathcal{M'} := (\mathcal{M}', s'_0)$, where $\mathcal{M}' = (S', R', L')$ and $s'_0 \in S'$;

if $\mathcal{M}' |= AG\phi$, then return \{ $\mathcal{M}'$ \}; else $\mathcal{M}' = \text{UPDATE}_{AG}(\mathcal{M}', \phi)$;

}\}

The function $\text{UPDATE}_{AG}(\mathcal{M}, \phi)$ is directed by Theorem 3.6. In this function, before update, if the initial state does not satisfy required properties, it should be updated with PU3 to protect the structure of an updated model. Then, this function updates each false path with either PU2, PU3 or PU5. The function $\text{UPDATE}_{AG}$ calls itself repeatedly if there are still false paths which are reachable from an initial state in an intermediate updated model. The function stops recursive calls and returns an updated model once there are no false states on any false paths in a model.

function $\text{UPDATE}_{EU}(\mathcal{M}, \phi_1, \phi_2)$

/* $\mathcal{M} \not|= E[\phi_1 U \phi_2]$. Update $\mathcal{M}$ to satisfy $E[\phi_1 U \phi_2]$. */

{
Select a path $\pi = [s_0, s_1, \ldots, s_i, \ldots, s_j, \ldots]$,  

1. If $\mathcal{M}_0 = (M, s_0) \not\models \phi_1$, then PU3 is applied to substitute $s_0$ with $s'_0 \models \phi_1$, and $Diff(s_0, s'_0)$ is minimal, such that  
   $\mathcal{M}' := \text{CTL*UPDATE}(\mathcal{M}_0, \phi_1)$, else
2. On path $\pi$, $\forall s_i \models \phi_1$, $\forall s_j \models \phi_1 \lor \phi_2$ such that $s' \not\models \phi_1 \lor \phi_2$ implies $s_i < s' < s_j$, then
   
   2.1 PU1 is applied to connect relation $(s_i, s_j)$: 
   
   $S' = S$, $R' = R \cup \{(s_i, s_j) | s_i, s_j \in S, S'\}$ and $L' = L$, or  
   
   2.2 PU4 is applied to add a new state $s^*$ between $s_i$ and $s_j$: 
   
   $S' = S \cup \{s^* | s^* \not\in S\}$, 
   $R' = R \cup \{(s_i, s^*)(s^*, s_j) | s_i, s_j \in S, S'\}$, 
   $L' : S' \rightarrow 2^{AP}$, where $\forall s \in S'$, if $s \in S$, then $L'(s) = L(s)$; 
   else $s = s^*$, and 
   $L'(s^*)$ is a set of true variables assigned in $s^*$, or  
   
   2.3 $\mathcal{M} = (M, s') \not\models \phi_1 \lor \phi_2$ implies $\mathcal{M}_{i+1} = (M, s_{i+1}) \not\models \phi_1 \lor \phi_2$:  
   
   2.3.1 if $Diff(s_{i+1}, s^*) \leq Diff(s_{i+1}, s'')$, where $s^* \models \phi_2$ and $s'' \models \phi_1$, then  
      
      PU3 is applied to $s_{i+1}$ with new states $s^* \models \phi_2$, such that  
      $\mathcal{M}' := \text{CTL*UPDATE}(\mathcal{M}_{i+1}, \phi_2)$, else
      
      2.3.2 PU3 is applied to all $s'$ with new states $s'' \models \phi_1$:  
      $\mathcal{M}' := \text{CTL*UPDATE}(\mathcal{M}_i, \phi_1)$, or  
      
   3. if $\exists s_j \models \phi_2$ on $\pi$, $\forall s \leq s_i$, $s \models \phi_1$, $s_{i+1} \not\models \phi_1$, such that  
      $\mathcal{M}_{i+1} = (M, s_{i+1}) \not\models \phi_1 \lor \phi_2$, then
      
      3.1 PU3 is applied to substitute a new state $s^* \models \phi_2$ on $s_{i+1}$, 
      and $Diff(s_{i+1}, s^*)$ is minimal, such that  
      $\mathcal{M}' := \text{CTL*UPDATE}(\mathcal{M}_{i+1}, \phi_2)$, or
      
      3.2 PU4 is applied to add a new state $s^* \models \phi_2$ after $s_i$, such that  
      $S' = S \cup \{s^* | s^* \not\in S\}$, 
      $R' = R \cup \{(s_i, s^*)(s^*, succ(s^*)) | s_i, succ(s^*) \in S, S'\}$, 
      $L' : S' \rightarrow 2^{AP}$, where $\forall s \in S'$, if $s \in S$, then $L'(s) = L(s)$; 
      else $s = s^*$, and 
      $L'(s^*)$ is a set of true variables assigned in $s^*$, or
3.3 if $\exists s \models \phi_2$, where $s \in S$ and $s \not\in \pi$, then,
PU1 is applied to connect $(s_i, s)$:

$S' = S$, $R' = R \cup \{(s_i, s) | s_i, s \in S, S'\}$ and $L' = L$;
4. $\mathcal{M}' = (M', s'_0)$, where $M' = (S', R', L')$ and $s'_0 \in S'$;
if $\mathcal{M}' \models E[\phi_1 U \phi_2]$, then return $\{\mathcal{M}'\}$; else $\mathcal{M}' = \text{UPDATE}_{EU}(\mathcal{M}', \phi_1, \phi_2)$;

The function $\text{UPDATE}_{EU}(\mathcal{M}, \phi_1, \phi_2)$ performs Theorem 3.7. It consists of a similar algorithm principle as $\text{UPDATE}_{EG}$. It conducts update for only one path. It updates each false state on a false path at a time then calls itself recursively to update other false states on the same path. Eventually, if there is a path in an updated model where all states before a state which satisfies $\phi_2$ satisfy $\phi_1$, this updated model is returned as an admissible model.

function $\text{UPDATE}_{AU}(\mathcal{M}, \phi_1, \phi_2)$
/* $\mathcal{M} \not\models E[\phi_1 U \phi_2]$. Update $\mathcal{M}$ to satisfy $E[\phi_1 U \phi_2]$.*/
{
Select a path $\pi = [s_0, s_1, \ldots, s_i, \ldots, s_j, \ldots]$,
1. If $\mathcal{M}_0 = (M, s_0) \not\models \phi_1$, then PU3 is applied to $s_0$, such that $\mathcal{M}' := \text{CTL}-\text{UPDATE}(\mathcal{M}_0, \phi_1)$, else
2. On path $\pi$, $\forall s_i \models \phi_1$, $\exists s_j \models \phi_1 \lor \phi_2$ such that $s' \not\models \phi_1 \lor \phi_2$ implies $s_i < s' < s_j$, then
   2.1 if $\text{Diff}(s_{i+1}, s^*) \leq \text{Diff}(s_{i+1}, s'')$, where $s^* \models \phi_2$ and $s'' \models \phi_1$, then,
      2.1.1 PU3 is applied to $s_{i+1}$ with new states $s^* \models \phi_2$, such that $\mathcal{M}' := \text{CTL}-\text{UPDATE}(\mathcal{M}_{i+1}, \phi_2)$, else
      2.1.2 PU3 is applied to all $s'$ with new states $s'' \models \phi_1$,
      $\mathcal{M}' := \text{CTL}-\text{UPDATE}(\mathcal{M}_i, \phi_1)$, or
3. if $\not\exists s_j \models \phi_2$ on $\pi$, $\forall s \leq s_i$, $s \models \phi_1$, $s_{i+1} \not\models \phi_1$, such that $\mathcal{M}_{i+1} = (M, s_{i+1}) \not\models \phi_1 \lor \phi_2$, then
   PU3 is applied to substitute a new state $s^* \models \phi_2$ on $s_{i+1}$,
   and $\text{Diff}(s_{i+1}, s^*)$ is minimal, such that $\mathcal{M}' := \text{CTL}-\text{UPDATE}(\mathcal{M}_{i+1}, \phi_2)$;
4. if $\mathcal{M} \models E[\phi_1 U \phi_2]$ and $\mathcal{M} \not\models A[\phi_1 U \phi_2]$, then

PU2 is applied to remove transition $(s_0, succ(s_0))$, such that

$S' := S$;

$R' := R - \{(s_0, succ(s_0))|s_0, succ(s_0) \in S, s \in S'\}$;

$L' := L$, since only a relation is removed, or

PU5 is applied to remove state $succ(s_0)$, such that

$S' := S - \{succ(s_0)|succ(s_0) \in S, s \not\in S'\}$;

$R' := R - \{(s_0, succ(s_0))|s_0 \in S, s \in S'\}$

$- \{(succ(s_0), succ(succ(s_i)))|succ(succ(s_0)) \in S, s \in S'\}$,

where if associated relations of $succ(s_0)$ are 2;

$L' : S' \rightarrow 2AP$, since $S' \subseteq S$, $\forall s \in S'$, such that $L'(s) := L(s)$;

5. $\mathcal{M}' := (M', s'_0)$, where $M' = (S', R', L')$ and $s'_0 \in S'$;

if $\mathcal{M}' \models A[\phi_1 U \phi_2]$, then return $\{\mathcal{M}'\}$; else $\mathcal{M}' = \text{UPDATE}_{AU}(\mathcal{M}', \phi_1, \phi_2)$;

The function $\text{UPDATE}_{AU}(\mathcal{M}, \phi_1, \phi_2)$ implements Theorem 3.8. This function updates all false paths in a model. Only PU3 is used in the first three cases. PU2 and PU5 are used in case 4. Case 1 is for protecting the structure of an updated model and providing the existing conditions to the rest of the cases. Cases 2 and 3 are for updating false states on each false path. Case 4 is for trimming false paths if there are already some true paths in a model where all states before a state which satisfies $\phi_2$ satisfy $\phi_1$ (e.g., the model satisfies $E[\phi_1 U \phi_2]$). The function must recursively call itself until all states before a state which satisfies $\phi_2$ satisfy $\phi_1$ on each path (e.g., the model satisfies $A[\phi_1 U \phi_2]$).

The primitive updates PU1-PU5 have been used in most functions in our algorithms. We now explain the atomic operations PU1-PU5 from the algorithmic perspective based on the principle of recursive calls in more detail. As we have stated earlier, PU3 is actually the combination of PU4 and PU5. However, from the algorithmic view, PU4 and PU5 can be performed by PU3. We perform PU1, PU2, PU4 and PU5 in subfunctions without calling the main function in our algo-
Algorithms for CTL Model Update

rithms. Whereas, if PU3 is performed, the subfunctions always need to call the main function first, then access other subfunctions. If PU3 is atomic by either adding or removing an atom in a state, then, the function UPDATE_p(M,p) is called through the main function if the atom is added into a state; the function UPDATE¬p(M,φ) is called through the main function to access the case if the atom is removed from a state. If PU3 is not atomic, such as for the required formula being “p ∨ q”, Then, the function UPDATE_p for the case “φ is φ_1 ∧ φ_2” is called through the main function. This function calls the main function again to access other subfunctions.

Having described the recursive calls used for performing PU3, we now analyze the intrinsic principles of performing PU4 and PU5 in the algorithms. If we want to perform PU4 to add a state to a model, where the state is supposed to have atom “p ∨ q”, then, we can decompose the adding process as follows: first, we just add an empty state into a model to redefine the updated model in the subfunction where PU4 is performed. Then, we apply PU3 to the new empty state with “φ is p ∨ q” by calling the main function to access other functions to eventually reach the atomic function UPDATE_p. If we want to perform PU5 to remove a state in a model, where the state is supposed to have atom “p ∨ q”, we first perform PU3 on this state with “φ is ¬p ∧ ¬q” by calling the main function. After this update, this state is empty. Then, we remove this empty state to redefine the updated model in the subfunction where PU5 is performed. We have seen, the atomic update operations PU4 and PU5 can be designed with a similar pattern to PU3 to make use of recursive calls. However, we do not show them in the same way as PU3 and merely describe needed PU4 and PU5 activation in subfunctions to simplify the expression.

4.2 Updating Examples with the Algorithms

We use two models, the microwave oven [16] and the AFS0 [84] in Chapter 6 to illustrate our designed algorithms. At this stage, we only consider the corresponding Kripke models and specification properties. We will use our CTL model update algorithms in this chapter to yield updated models.

For the microwave oven model, its Kripke model is shown in Figure 6.1. Its specification property is “¬EF(Start ∧ EG¬Heat)”. From model checking demon-
strated in [16], we know that states \(s_2\) and \(s_5\) do not satisfy the specification property. Now, we process model update with our algorithms. First, we access the main function CTL\(*\)UPDATE to start. The main function calls the function UPDATE\(_\neg\) under the case “\(\phi\) is \(\neg\phi_1\)” for the specification property described above.

In the function UPDATE\(_\neg\), the function UPDATE\(_{AG}\) is called under the case “\(\phi\) is \(\neg\text{EF}(\phi_1) = \text{AG}(\neg\phi_1)\)” after the original specification property is converted into the formula “\(\text{AG}(\neg(\text{Start} \land \text{EG}\neg\text{Heat}))\)”.

Then detailed update according to minimal change rules is executed in the function UPDATE\(_{AG}\). If the update involves PU2 and PU5, either the transition \((s_1, s_2)\) is removed as shown in Figure 6.2 or the state \(s_2\) is removed as shown in Figure 6.3. For the case which involves update with PU3, the function UPDATE\(_{AG}\) calls the main function with “\(\phi\) is \(\neg\text{(Start} \land \text{EG}\neg\text{Heat})\)”.

This is categorized into the case “\(\phi\) is \(\neg\phi_1\)” in the main function. Then, the main function calls the function UPDATE\(_\neg\), which converts “\(\neg(\text{Start} \land \text{EG}\neg\text{Heat})\)” into “\(\neg\text{Start} \lor \neg\text{EG}\neg\text{Heat})\)” under the case “\(\phi\) is \(\neg(\phi_1 \land \phi_2) = \neg\phi_1 \lor \neg\phi_2\)”.

After the conversion, either “\(\phi_1\) is \(\neg\text{Start}\)” or “\(\phi_2\) is \(\neg\text{EG}\neg\text{Heat}\)” could be selected as the input \(\phi\) for the function UPDATE\(_\neg\). Because “\(\neg\text{Start}\)” is simpler, it is selected.

Then, the function UPDATE\(_\neg\) updates states \(s_2\) and \(s_5\) with atomic update under the case “\(\phi\) is \(\neg p\)”.

For our case, “\(\neg p\) is \(\neg\text{Start}\)”.

The updated model with PU3 is shown in Figure 6.4.

For the AFS0 model, its Kripke model is shown in Figure 6.7. Its specification property is formula (6.2) which can be converted into formula (6.3) as “\(\text{AG}((\text{Server}.\text{belief} = \neg\text{valid}) \lor (\text{Client}.\text{belief} = \text{valid}))\)”.

The false states in Figure 6.7 are identified as states 8 and 16. Then, we conduct model update to repair the model using the principles in our algorithms. We first access the main function to call the function UPDATE\(_{AG}\). This function executes update with PU2, PU3 or PU5 for false paths [2, 10, 8, · · ·] and [2, 10, 16, · · ·]. If the update is PU2, either the transition \((10, 8)\) or \((10, 16)\) is removed. If the update is PU5, either state 8 or 16 is removed. If the update is PU3, then the function UPDATE\(_{AG}\) calls the main function with “\(\phi\) is \(\phi_1 \lor \phi_2\)”.

For the case in the AFS0 model, “\(\phi\) is \((\text{Server}.\text{belief} = \neg\text{valid}) \lor (\text{Client}.\text{belief} = \text{valid}))\)” and “\(\phi\) is \((\text{Server}.\text{belief} = \neg\text{valid})\)” or “\(\phi\) is \((\text{Client}.\text{belief} = \text{valid})\)”.

Then, the main function calls itself with either “\(\phi\) is \((\text{Server}.\text{belief} = \neg\text{valid})\)” or “\(\phi\) is \((\text{Client}.\text{belief} = \text{valid})\)”.
For the former, the main function calls the function $\text{UPDATE}_\neg$ under the case “$\phi$ is $\neg \phi_1$” to access the atomic update “$\phi$ is $\neg p$”. For the latter, the main function calls the function $\text{UPDATE}_p$ under the case “$\phi$ is atomic $p$”. After update with the combinations of PU1, PU2 and PU3, some of the updated admissible models are shown in Figures 6.9, 6.10 and 6.11.

From the above two examples, we have seen that our designed algorithms can properly perform model update functions. These algorithms form the basis for the implementation of a prototype CTL model updater. The detailed implementation will be introduced in Chapters 5 and 6 following our algorithmic principles and will prove that our algorithms make sense for practical implementation.
Chapter 5

The CTL Model Updater
Prototype

5.1 The Relationship between Model Checking and Model Update

Model checking is to verify whether a model satisfies certain required properties. Model checking is performed by the model checker. The SMV model checker was first developed by McMillan [63] based on previous developed model checking theoretical results. This SMV model checker uses SMV as its specification language. Models and specification properties are all in the form of SMV as the input. The SMV model checker parses the input into a structured representation for processing. Then, the system conducts model checking by labelling algorithms implemented in the SAT function [16, 50]. The output is usually counterexamples which report error messages as the result of model checking. During the model checking, there was a state explosion problem, which significantly influenced the SMV model checking efficiency. The introduction of OBDD [16, 50] in the SMV model checker greatly reduced the state explosion problem. After the first successful SMV compiler, the enhanced model checking compilers, NuSMV and Cadence SMV, were developed. NuSMV is an enhanced model checker from SMV and is more robust by the integration of a CUDD package [12]. It also supports LTL model checking. Cadence SMV was implemented for compositional systems and industrial use. From SMV, NuSMV to Cadence SMV, the model checkers have been developed from experimental versions to industrialized usage versions.
In Figure 5.1, the original model checker is the part of flow starting from “A Software System” and “Specification Properties in CTL” and ending at “the original model” box. A software system is abstracted as a Kripke model [86]. Then, the Kripke model and specification properties are coded in SMV as input. The input SMV code is checked by the SMV model checker. The output consists of counterexamples if the model checker found any errors in the model by comparing it with the specification properties. The model remains unchanged.

Model update is an attempt to repair errors in a model if the model does not satisfy certain properties and is performed by the model updater. Our model updater updates the model after checking by the model checker if it does not satisfy the specification properties. The eventual output should be an updated model which satisfies the specification properties. The part of flow after “The Original model” in Figure 5.1 shows the model updater. The whole figure shows the complete process of model checking and model update.

5.2 The Prototype of the CTL Model Updater

We have simulated a prototype of the CTL model updater in Linux C as the implementation of our algorithms. This simulation is a simplified CTL model updating compiler due to the time limitation of a Ph.D study. The implementation obeys
the spirit of our pre-designed algorithms but has minor deviations due to practical considerations. The implementation meets our algorithmic requirements with a significant focus on coding techniques. Unlike SMV, the input models are pre-specified (defined) in C code. Our system does not contain OBDD [50] optimization unlike the SMV mode updater. Our model checker does not represent as significant a processing load as the SMV compiler because it contains a much less flexible, simpler parsing and checking implementation sufficient to accommodate the models studied for this thesis. We have coded our own model checking functions to perform the model checking duty during the update process. Because our coding does not involve SMV parsing, we do not use the result directly from SMV, and our CTL model update approach has not yet been integrated into an existing SMV model checker. Although we can use the output of SMV as the result of model checking, we need to involve more details such as the correct and error states in a model or a path and so on. Thus, it is easier for us to code our own model checking functions which we then use as part of our CTL model updater. The CTL model updater includes library functions, predefined model definition functions, a specification string parser, model checking functions and model update functions. The ReadMe file for this prototype is included as Appendix C of this document. The diagram of the code structure is shown in Figure 5.2. A detailed description of the system follows.

5.3 Predefined Structures and Library Functions

We have coded a set of pre-defined structures for the whole system. The most significant structures are the model definition structure, the state structure, the state data structure, the variable definition structure and the atom and calc_pair structures for storing specification string parsing results.

The model definition structure contains the major elements of a CTL model. This structure first contains the name of a model. This structure next contains a variable definition structure. The structure also contains a state pointer array and a state count, where each reachable state is defined in a state structure. The model definition structure also contains a path pointer array and a path count, where each path is defined in a path structure. A path is a set of states in a model.
The CTL Model Updater Prototype

Figure 5.2: The flow diagram of the Model Update System

typedef struct {
    char shortname[MAXCHAR];
    char longname[MAXCHAR];
    int numvar;
    var_defn var[MAXVAR];
    int numstates;
    state_ptr state[MAXSTATE];
    int numpaths;
    path_ptr path[MAXPATH];
} state_defn;

Figure 5.3: The state definition structure

typedef struct {
    int num;
    boolean initial;
    variable var[MAXVAR];
    int numnext;
    int next[MAXTRANS];
    int numprev;
    int prev[MAXTRANS];
    boolean result;
} state;

Figure 5.4: The state structure in C code

representing a valid set of state to state transitions. The path structure contains a state count and an array of state pointers. The structure in C code is as Figure 5.3.

In the model definition structure, “shortcutname” is an abbreviated name for the currently defined model. This is used by the updater logic for selective conditional model processing. “longname” is a more comprehensive name of the model used for model print out; “numvar” is the number of variables; “var[MAXVAR]” is an array of variable definition structures; “numstates” is the number of non repeated reachable states in a model. “state[MAXSTATE]” is an array of pointers to state
structures containing each non repeated reachable state; “numpaths” is the number of paths in a model, where a path in our system is a set of states beginning from an initial state and terminating when a state is visited for a 2nd time or an initial state is arrived at; “path[MAXPATH]” is an array of pointers to the defining path structures. In our CTL model updater, the model definition structure is defined as a static instance, hence the original model is overwritten. The change due to update on a model is eventually stored in the definition instance.

The state structure is the major component defining a model. The state structure contains all information in a state, in particular the values of the variables of the state, and the relations in between this state and its previous or successive states. The state structure is defined as Figure 5.4. In this structure, “num” is an identifier as an integer of a state; “initial” is a boolean variable to define this state as an initial state in the model; “var[MAXVAR]” is an array of values of indexed variables for this state in a particular model; “numnext” is the total number of next states; “next[MAXTRANS]” is an array of the integer identifiers of next states; “numprev” is the total number of previous states; prev[MAXTRANS] is an array of the integer identifiers of previous states. “result” is a boolean variable to store the checking result for the state.

Another two major structures are called “state_data”, which is an interface structure to actually load a state structure, and “var_defn”, which stores the names, number and value names of the defined variables in a particular model. For example, in the microwave oven model [16], “Heat” and “Error” are variables which have “true” or “false” values. The variable definition structure accommodates different variable names with the corresponding value names in different models.

The library functions include all initializations of the model in the definition structure, simple operations for model checking and update, and printing functions for a model and its paths. For the initializations, there are functions for defining a model, its name and states, setting data in states, setting and clearing links in between states and so on. The simple operations for model checking and update include checking individual and all states in a model, checking a path or all paths in a model, adding or removing states, building or removing links in between states
and calculating paths etc. The printing functions include printing states, paths and the model. The printing functions assist the user in understanding the operations performed by the model updater.

The library functions and structures are called and used by other parts of the code. Virtually all other functions call library functions directly or indirectly.

### 5.4 The Parser

The parsing functions decompose a complex CTL formula, expressed as a string, into a number of linked structures. The components of the structures have direct equivalence to each recognizable component of the specification string as our case study illustrates below. For our system the part which needs to be parsed is the string representing the specification property, such as the property in the microwave oven model: “¬EF(Start ∧ EG¬Heat)”. Our parser rationalizes a CTL specification string according to the Backus Naur form [50] expressed as Definition 2.2. There are two major structures used by our parsing library functions which store our parsing results.

An atom structure (Figure 5.5) stores the results of parsing a symbol \( \phi \) expression including \( \neg \) and path navigation expressions. An atom structure assumes that the string contains semantics such as AG, EG and so on with an atomic variable successor.
§5.4 The Parser

\begin{verbatim}
| negate1       | false |
| negate2       | false |
| navigate      | 0     |
| varindex      | 101   |
| valindex      | 1     |
| error         | false |
\end{verbatim}

**Figure 5.7:** The parsed structure of String “Start”

\begin{verbatim}
| negate1       | true  |
| navigate      | 6     |
| varindex      | 103   |
| valindex      | 1     |
| error         | false |
\end{verbatim}

**Figure 5.8:** The parsed structure of string “EG¬Heat”

In Figure 5.5, “negate1” is the negation symbol in front of “navigate” (such as AG or EG); if “negate1” is true, the negation symbol in front of “navigate” is there, otherwise, there is not a negation symbol; “negate2” is the negation symbol after “navigate”. It behaves the same as “negate1”; “navigate” is the semantics about the model such as “AG” or “EG”. We define numbers to represent different semantics. For example, “AF” is 4, “AG” is 5 and “EG” is 6; “varindex” is the index number of the variables in our system and represents the index position of the variable in the model definition object but includes an adder to avoid conflict with other indexes, which serves for our code only; “valindex” is the index of the value corresponding with the specified variable index; “error” indicates whether the atom parsed correctly or not. If “error” is true, it means that the atom may not exist in our model. For example, if a string is “Start”, which is a name of a variable in the model, then the structure of the parsed string should be as Figure 5.7.

If a string is “EG¬Heat”, where “Heat” is a name of a variable in the model, then the structure of the parsed string is as Figure 5.8.

A pair structure stores results of parsing an expression containing two CTL formulae expressions and a separating operator. This structure includes storage for a path navigation expression and leading and following negate declarations. If the structure of a string is more complex than an atom, then it needs to be expressed in a pair structure in Figure 5.6. In this structure, “negate1”, “negate2”, “navigate” and “error” are the same concepts as those in the atom structure; “operator” is a logic symbol such as “∧” or “∨” and is defined as an integer in the structure; “operand1” is the “atom” before “operator”; “operand2” is the “atom” after “operator”; “nestedpair1” (“nestedpair2”) is a casted type of “calc_pair” if the string
before (or after) “operator” is a “calc_pair”, which can accommodate recursively nested “calc_pair” structures; For example, the string “AG(¬(Start ∧ EG¬Heat))” can be parsed into the “calc_pair” structure as in Figure 5.9.

In this Figure, the part of string after “AG” is interpreted as the first out pair: “nested pair1”. The elements before “nested pair1” match AG in the given string; the elements after “nested pair1” and before “operator” are the “¬” after “AG”; “operator· · · 23” is the “∧” in between “Start” and “EG¬Heat”; the elements after “operand1” and “operand2” are atoms which have been explained earlier.

During model checking and update, we select the needed elements for any parts of the string from the corresponding parsed structure.

The parser also contains a set of functions to rationalize negate symbols (normalize) in a specification to simplify processing. These functions use the parsing structures as input and output.

As an example consider the specification string “¬EF(Start ∧ EG¬Heat)”. After parsing, the resulting structures are processed by the normalize functions to achieve the following conversions. To assist understanding, the conversions are described using their string equivalents. The stage 1 conversion moves the outer “¬” to the right. Thus, “¬EF(Start ∧ EG¬Heat)” becomes “AG(¬(Start ∧ EG¬Heat))”. The state 2 conversion moves the outer “¬” to inside the inner brackets. Thus, “AG(¬(Start ∧ EG¬Heat))” becomes “AG(¬Start ∨ ¬EG¬Heat)”. The conversions are based on CTL semantics.

Function UPDATE¬ in our designed algorithm uses the parsing and normalize functions in different forms in a more abstract way.

5.5 Model Checking Functions

The model checking functions are for checking CTL semantics, such as whether “AG”, “EG”, are true or not. They are continually used for the whole process of update. Before or after each step of update, they are called to do model checking and identify error or correct states according to different semantics or update requirements. If a specification property (string form) which needs to be checked is not an atomic variable, the string should be processed by a parser to be interpreted
primary pair →
negate1 · · · false
negate2 · · · false
navigate · · · 5
operator · · · 0
operand1 · · · 0x00000000
operand2 · · · 0x00000000

nested pair1 →
negate1 · · · true
negate2 · · · false
navigate · · · 0
operator · · · 23
operand1 →
negate1 · · · false
negate2 · · · false
navigate · · · 0
varindex · · · 101
valindex · · · 1
error · · · false

operand2 →
negate1 · · · false
negate2 · · · true
navigate · · · 6
varindex · · · 103
valindex · · · 1
error · · · false

nestedpair1 · · · 0x00000000
nestedpair2 · · · 0x00000000
error · · · false

Figure 5.9: The parsed structure for string “AG(¬(Start ∧ EG¬Heat))”
into components in the parsing structures first. Then it is checked by atomic model checking functions, which deal with model checking for atomic variables only. In our model checking functions, we have checking functions with “true” or “false” results to tell whether a model satisfies a specification property with CTL semantics. To assess whether a state satisfies the required property or not, we compare the variables in a state with the variables in the required property. For example, if a state is empty (all defined variables are false), if the required property for this state is “¬Start”, then it means that this state satisfies the required feature with checking result “true”. Otherwise, the result is “false”. For particular semantics such as “EG”, its model checking function is performed for each path, where each state is checked. If all states on at least one path satisfy the required property, then it means the model checking is “true” with semantic “EG”. Besides, we also have functions which identify error or correct paths or states for particular CTL semantics, which will be used for model updating. For example, for semantics “EG”, there are functions to identify correct or error states in a model or correct or error paths in a model. The information contained in these functions is the state or path identification numbers. If model update functions use them, they can locate the error paths or states straight away to perform model update on these states or their incoming or outgoing transitions.

5.6 Model Update Functions

The model updating functions are the most important part of the implementation and demonstrate our previous theoretical results. They are called to update the model either on paths (eventually on states of the path) or states among all reachable states. If a CTL property is a string, it must be decomposed into its parsing structure. Then we identify related atomic elements in the structure to update the model. The update functions frequently call model checking functions for each step update to see whether the updated model satisfies certain features or not. If the updated model satisfies the required feature, then the update is halted and the system returns the updated model. The update obeys our minimal change rules. The resulting model could be more than one if they are not interchangeable. If the
update changes the model, the definition structure containing the model is changed as well.

The update functions include atomic updates (level 1) PU1 to PU5, which add or remove single states and their relations, or change variables. Above the atomic updates, we have the 2nd level update functions for updating the semantics of a model such as AG, EG etc. Above the 2nd level update functions, we have the outer level (level 3) update functions which are the combination of parsing, model checking and updating for cases where the specification string involves more complex semantics on multiple atomic variables. Level 2 update functions use our semantics characterization results and implicitly contain the logic of our pre-designed algorithms. In addition, a reachable state algorithm is included in update functions to solve an admissible model explosion problem. This algorithm will be introduced in detail in section 7.2.2.

If the string representing the required property is not an atomic variable, then we parse the string before doing model checking and update. For example, if the input required property is AG($\text{Start} \land \neg \text{Error}$), then all states in a model should satisfy the string after AG. The string, “$\text{Start} \land \neg \text{Error}$”, should be parsed before further update for each state on this model. This process is performed by the functions at our 2nd and 3rd level updates which call the 1st level functions. During the process, string parsing and nested model checking and update involves certain degrees of intelligent reasoning depending on the semantics and complexity of the string. The reasoning is done by update functions on the 3rd level. If a required property is in a form such as “AG($\text{Start}$)” where Start is a variable, then it can be performed by the 2nd level update functions which eventually call the 1st level update functions.

The contents of this chapter are published in [27].
Three Case Studies

Models in our model update system are predefined in C code. We use three models with different complexities to demonstrate our model updater prototype. They are the microwave oven (m-oven) [16], the Andrew File System 0 (AFS0) [84] and Andrew File System 1 (AFS1) [86] models. The microwave oven model has a total of $2^4 = 16$ states, where there are 7 reachable states and one initial state, and 4 variables with boolean values. The AFS0 model has $2^4 = 16$ states, where there are 6 reachable states and one initial state, and 4 variables each with 2 possible values. The values of variables are not true or false as those in the microwave oven model. The AFS1 model has $4 \times 3 \times 3 \times 3 \times 2 = 216$ states, where 26 states are reachable, 4 states are initial states, and each state has 5 variables. Among the variables, one variable has 4 values, 3 variables have 3 values and one variable has 2 values. The microwave oven and AFS1 models in code are designed with reachable states only. The AFS0 model in code contains all states reachable or unreachable.

The models are predefined in three C files. The microwave oven model is predefined in model.c, AFS0 in afs0.c and AFS1 in afs1.c. Our model updater is built with one of the model definition source files, with the name of the executable file reflecting the model definition. The library files are used for each executable, and have been written to be compatible for each. Any model specific processing is selectively performed using the model short name as the conditional element. (see section 5.3 state definition structure description). The model definition files use the library functions to build up the model with the following operations. Each model definition file identifies the model and its variable names and values, defines states in the model, sets data values for variables in states, and builds up relations in between states and their next and previous states. The string of the specification
property is in a char array (string). After the make file is run, the system creates three dedicated executable files which show model update results for each individual model.

6.1 The First Model – the Microwave Oven

6.1.1 Applying the Model Updater to the Microwave Oven

Example

The microwave oven example is a hardware model [16]. The Kripke structure of this model is shown in Figure 6.1, where there are 7 reachable states \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\} and 12 relations. \(s_1\) is the initial state. The set of variables are \{Start, Close, Heat, Error\} and each variable has boolean values. So, the number of total states for the microwave oven model is \(2^4 = 16\). The specification property is “\(\neg \text{EF} (\text{Start} \land \text{EG} \neg \text{Heat})\)”. The overall scenario is that the microwave oven has two major processes, one is a faulty heat process, which is \(s_1 \rightarrow s_2 \rightarrow s_5 \rightarrow s_3 \rightarrow s_1 \cdots\) and the other is a normal heat process: \(s_1 \rightarrow s_3 \rightarrow s_6 \rightarrow s_7 \rightarrow s_4 \rightarrow s_1 \cdots\). The SMV model checker has identified the faulty process (path) and our model updater can repair the false states on the faulty path. We show the model checking and update scenario in our system as follows.

model.c defines the microwave oven model. The model is stored in an instance of the model definition structure. The variables are defined as boolean. The specification is predefined in a char array (string).

First, we parse the specification string “\(\neg \text{EF} (\text{Start} \land \text{EG} \neg \text{Heat})\)” into a parsing structure. Then, we convert the structure into a new structure corresponding to specification formula AG(\(\neg (\text{Start} \land \text{EG} \neg \text{Heat})\)) to remove the front \(\neg\). The conversion is performed by a normalize function and its logic is mapped into function UPDATE_\neg in our pre-designed algorithms, which can call other parsing functions. The parsing structure of the string “AG(\(\neg (\text{Start} \land \text{EG} \neg \text{Heat})\))” is shown in Figure 5.9.

Then, we must check each state’s variables (because of AG) according to the property \(\neg (\text{Start} \land \text{EG} \neg \text{Heat})\) which is a nested calc_pair in our parsing structure.
This is performed by a model checking process for AG which is called by level 3 update functions. We select “EG¬Heat” after “∧” to update first, whose parsed elements are under “operand2” of “nested pair1” in Figure 5.9, to apply model checking functions to identify a path (or paths) for which EG is valid. In this model, any path which has each state with variable Heat false should be identified. Here, we find the paths s₁ → s₂ → s₅ → s₃ → s₁ · · · and s₁ → s₃ → s₁ · · · which are Strongly Connected Components (SCC) loops [16] satisfying EG¬Heat. Then, we check where the states have variable Start true, which is the atomic string before ∧ in the specification string and maps the elements in between “operand1” and “operand2” under “nested pair1” in the parsed structure in Figure 5.9. We identify states s₂, s₅,s₆ and s₇ with Start true by model checking functions for AG because before ∧ “Start” is atomic and the “AG” before “Start” should be mapped as the semantic symbol in front of “Start”. Now, we must identify states which have both variables Start true and Heat false because of the “∧” operator between “Start” and “EG¬Heat”. These states are s₂ and s₅. It means that the two states satisfy Start ∧ EG¬Heat. However, the “AG(¬)” before them in “AG(¬(Start ∧ EG¬Heat))” specifies that the model should not have any state which satisfies this feature. Thus, we must update s₂ and s₅.

Now, the 2nd level update function for AG (UPDATE_AG) calls atomic (1st level) update functions such as PU1-PU5. The results are three equal minimal updates: for the atomic update PU2 case, relation (s₁,s₂) is deleted; for the atomic update PU5 case, state s₂ and relations (s₁,s₂), (s₂,s₅) and (s₅,s₂) are deleted; for the PU3 case, we must normalize the part of string after “AG” before PU3 is performed. ¬(Start ∧ EG¬Heat) = ¬Start ∨ ¬EG¬Heat. The corresponding parsed structure for “¬Start ∨ ¬EG¬Heat” is as Figure 5.10.

Thus, eventually the faulty states s₂ and s₅ should be updated with either ¬Start or ¬EG¬Heat in an update function for the ∨ operator as described in the function CTL*UPDATE in our predesigned algorithms. Obviously, ¬Start is simpler thus is chosen. As we mentioned, the selection process involves certain intelligent reasoning.

After these updates, the resulting model M’= (M’,s₁) ⊨ ¬EF(Start∧EG¬Heat). The above three resulting models are all minimally changed from the original model.
and are admissible. They are not interchangeable with each other due to our minimal change rules. The updated models are shown as Figure 6.2, 6.3 and 6.4.

### 6.1.2 The simulation Results for Updating the Microwave Oven Model

We show partial screen results by running the executable file as follows:

In the beginning, the screen shows the model name, variables, states and relations in between states:

**State Machine Model**

**Short Model name is m-oven**

**Long Model name is Microwave Oven**
Variable name #1 is Start with 2 values -> false, true
Variable name #2 is Close with 2 values -> false, true
Variable name #3 is Heat with 2 values -> false, true
Variable name #4 is Error with 2 values -> false, true

State Information for 7 states is ->

<table>
<thead>
<tr>
<th>Id</th>
<th>Initial Values</th>
<th>Next Links</th>
<th>Previous Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*** false false false false</td>
<td>-&gt; 2 -&gt; 3</td>
<td>&lt;- 4 &lt;- 3</td>
</tr>
<tr>
<td>2</td>
<td>true false false true</td>
<td>-&gt; 5</td>
<td>&lt;- 1 &lt;- 5</td>
</tr>
<tr>
<td>3</td>
<td>false true false false</td>
<td>-&gt; 6 -&gt; 1</td>
<td>&lt;- 1 &lt;- 5 &lt;- 4</td>
</tr>
<tr>
<td>4</td>
<td>false true true false</td>
<td>-&gt; 3 -&gt; 1 -&gt; 4</td>
<td>&lt;- 7</td>
</tr>
<tr>
<td>5</td>
<td>true true false true</td>
<td>-&gt; 2 -&gt; 3</td>
<td>&lt;- 2</td>
</tr>
<tr>
<td>6</td>
<td>true true false false</td>
<td>-&gt; 7</td>
<td>&lt;- 3</td>
</tr>
<tr>
<td>7</td>
<td>true true true false</td>
<td>-&gt; 4</td>
<td>&lt;- 6</td>
</tr>
</tbody>
</table>

We omit parsed structure and paths here. The states which must be updated are identified as $s_2$ and $s_5$. We only demonstrate three admissible updated results as follows.

Case 1: after PU2 update on the relation between state 1 & 2

State Information for 7 states is ->

<table>
<thead>
<tr>
<th>Id</th>
<th>Initial Values</th>
<th>Next Links</th>
<th>Previous Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*** false false false false</td>
<td>-&gt; 3</td>
<td>&lt;- 4 &lt;- 3</td>
</tr>
<tr>
<td>2</td>
<td>true false false true</td>
<td>-&gt; 5</td>
<td>&lt;- 5</td>
</tr>
<tr>
<td>3</td>
<td>false true false false</td>
<td>-&gt; 6 -&gt; 1</td>
<td>&lt;- 1 &lt;- 5 &lt;- 4</td>
</tr>
<tr>
<td>4</td>
<td>false true true false</td>
<td>-&gt; 3 -&gt; 1 -&gt; 4</td>
<td>&lt;- 7</td>
</tr>
<tr>
<td>5</td>
<td>true true false true</td>
<td>-&gt; 2 -&gt; 3</td>
<td>&lt;- 2</td>
</tr>
<tr>
<td>6</td>
<td>true true false false</td>
<td>-&gt; 7</td>
<td>&lt;- 3</td>
</tr>
<tr>
<td>7</td>
<td>true true true false</td>
<td>-&gt; 4</td>
<td>&lt;- 6</td>
</tr>
</tbody>
</table>

This output demonstrates the removal of the $s_1$ to $s_2$ state transition.
Case 2: after PU5 update on states 2 & 5

State Information for 6 states is ->

<table>
<thead>
<tr>
<th>Id</th>
<th>Initial</th>
<th>Values</th>
<th>Next Links</th>
<th>Previous Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>***</td>
<td>false false false false</td>
<td>-&gt; 3</td>
<td>&lt;- 4 &lt;- 3</td>
</tr>
<tr>
<td>3</td>
<td>false</td>
<td>true false false</td>
<td>-&gt; 6 -&gt; 1</td>
<td>&lt;- 1 &lt;- 5 &lt;- 4</td>
</tr>
<tr>
<td>4</td>
<td>false</td>
<td>true true false</td>
<td>-&gt; 3 -&gt; 1 -&gt; 4</td>
<td>&lt;- 7</td>
</tr>
<tr>
<td>5</td>
<td>true</td>
<td>true false true</td>
<td>-&gt; 3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>true</td>
<td>true false false</td>
<td>-&gt; 7</td>
<td>&lt;- 3</td>
</tr>
<tr>
<td>7</td>
<td>true</td>
<td>true true true false</td>
<td>-&gt; 4</td>
<td>&lt;- 6</td>
</tr>
</tbody>
</table>

This output demonstrates the removal of $s_2$ and its associated links.

Case 3: after PU3 update on states 2 & 5

State Information for 7 states is ->

<table>
<thead>
<tr>
<th>Id</th>
<th>Initial</th>
<th>Values</th>
<th>Next Links</th>
<th>Previous Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>***</td>
<td>false false false false</td>
<td>-&gt; 3 -&gt; 22</td>
<td>&lt;- 4 &lt;- 3</td>
</tr>
<tr>
<td>22</td>
<td>false</td>
<td>false false true</td>
<td>-&gt; 55</td>
<td>&lt;- 1 &lt;- 55</td>
</tr>
<tr>
<td>3</td>
<td>false</td>
<td>true false false</td>
<td>-&gt; 6 -&gt; 1</td>
<td>&lt;- 1 &lt;- 4 &lt;- 55</td>
</tr>
<tr>
<td>4</td>
<td>false</td>
<td>true true false</td>
<td>-&gt; 3 -&gt; 1 -&gt; 4</td>
<td>&lt;- 7</td>
</tr>
<tr>
<td>55</td>
<td>false</td>
<td>true false true</td>
<td>-&gt; 3 -&gt; 22</td>
<td>&lt;- 22</td>
</tr>
<tr>
<td>6</td>
<td>true</td>
<td>true false false</td>
<td>-&gt; 7</td>
<td>&lt;- 3</td>
</tr>
<tr>
<td>7</td>
<td>true</td>
<td>true true false false</td>
<td>-&gt; 4</td>
<td>&lt;- 6</td>
</tr>
</tbody>
</table>

This output demonstrates the modification of $s_2$ and $s_5$ (re-identified as 22 and 55) with updated variable values. 22 is $s'_2$ and 55 is $s'_5$ in Figure 6.4.

6.2 Another Two Models – AFS0 and AFS1

AFS0 and AFS1 are abbreviations of different versions of the Andrew File System [84, 86]. The Andrew File System contains a series of simple to complex models from early to more recent times named AFS0, AFS1 and AFS2. They are cache
coherence protocols for distributed file systems. The SMV model checker was first successfully applied to these software protocols in 1995 by Wing and Vaziri-Farahani, which was a milestone showing that the SMV model checker applied to not only hardware verification before 1995 but also to software verification after this. We choose AFS0 [84] as one of our implementation models because AFS0 has the similar complexity as in the microwave oven model. AFS0 has a total of 16 states where 6 states are reachable states. It also has variables of similar complexity as the microwave oven example. Another reason to choose AFS0 is that it has a similar principle as AFS1 but is much simpler. AFS0 and AFS1 both consist of a client and a server module. In order to explore AFS1 well, it is a good starting point to implement AFS0 first. The AFS1 model [86] was chosen to demonstrate the updater’s ability to accommodate much more complex models. AFS1 has a total of 216 states where 26 are reachable states. Also AFS1 has more complex variables than AFS0. The microwave oven model is a hardware model while AFS0 and AFS1 models are software models. Applying our model updater prototype to these models illustrates that our model updater is applicable for handing both hardware and software examples with different complexities.

Before we go to the details of AFS0 and AFS1, we briefly review the scenario of these protocols. As described in [86], in a distributed file system, servers store files; clients store local copies of these files in their caches. Clients communicate with servers by exchanging messages and data. Clients do not communicate with each other. Each file is associated with exactly one designated “home” or authorized server. A problem arises when there are several copies of a file in a system. A file is valid if it is the most recent copy in the system. Recency is typically determined by a timestamp associated with the file. If a client updates its copy and it is the most recent update in the system, then all other copies of that file become invalid. The goal of a cache coherence protocol is to make sure a client performs work only on files it believes are valid. The validation-based technique is that the client asks the server whether its copy is valid to ensure cache coherence. Before a run a client considers all files in its cache suspect, which means that it does not have a belief on their validity. During a run clients and servers gain beliefs about cached files. At
the end of a run, they all discard their beliefs. If failures occur, beliefs are discarded but clients do not discard their cached files. Pairwise knowledge between clients and servers are used in this system and the notion of belief captures this pairwise knowledge.

6.2.1 The AFS0 Model

In the AFS0 model described in [84], the client simply requests a copy of the file from the server. The server then sends a copy to the client. The server’s belief about the file cached by the client ranges over \{valid, none\}. If the server’s belief is valid then the server thinks that there is a file in the client’s cache and it is valid; none, then the server has no belief about the existence of a file in the client’s cache or its validity. The client’s belief ranges over \{valid, nofile\}. The client’s belief is valid if the client thinks that there is a file in its cache and it is valid; nofile if it believes that there is no file in its cache. The client and server communicate by sending messages to each other. These messages range over \{fetch, val\}. The message fetch is sent by the client to the server to request a new copy of the file. The message val is from the server to the client indicating that a copy of the file has been sent.

In Figure 6.5, the finite state diagrams for AFS0 show that the client’s initial belief is nofile and the server’s initial belief is none. The client sends a fetch message to the server. The server then sends a val message to the client. At the end of the run, both the client and the server believe that the file cached by the
client is valid. It is assumed that the file is sent along with the message \textit{val}. In this figure, the labels of the transitions are received messages. Upon receipt of the message \textit{val} the client’s belief changes from \textit{nofile} to \textit{valid}. Similarly upon receipt of the message \textit{fetch}, the server’s belief changes from \textit{none} to \textit{valid}.

### 6.2.2 The AFS1 Model

AFS1 applies a validation-based technique for the client-server protocol. As described in [86], a client has two initial states: either it has no files or it has one or more files but no beliefs about their validity. If the protocol starts with the client having suspect files, then the client may request a validation for a file from the server. If the file is invalid then the client requests a new copy and the run terminates. If the file is valid, the protocol simply terminates. AFS1 is abstracted as a model with one client, one server and one file. Figure 6.6 shows the state transition diagrams with single client and server modules. The nodes are labeled with the value for the state variable, \textit{belief}; the arcs, by the name of the message received that causes the state transition. A run of the protocol begins at an initial state (one of the leftmost nodes) and ends in a final state (one of the rightmost nodes).

The client’s belief about a file has 4 possible values \{\textit{nofile, valid, invalid, suspect}\}, where “\textit{nofile}” means that the client cache is empty; \textit{valid}, if the client believes its cached file is valid; \textit{invalid} if it believes its caches file is not \textit{valid}; \textit{suspect}, if it has no belief about the validity of the file (it could be \textit{valid} or \textit{invalid}). The server’s belief about the file cached by the client ranges over \{\textit{valid, invalid, none}\}, where \textit{valid}, if the server believes that the file cached at the client is valid; \textit{invalid}, if the server believes it is not valid; \textit{none}, if the server has no belief about the existence of the file in the client’s cache or its validity.

The set of messages that the client may send to the server is \{\textit{fetch, validate}\}. The message \textit{fetch} stands for a request for a file. The \textit{validate} message is used by the client to determine the validity of the file in its cache. The set of messages that the server may send to the client is \{\textit{val, inval}\}. The server sends the \textit{val} (\textit{inval}) message to indicate to the client that its cached file is \textit{valid} (\textit{invalid}).
6.2.3 The Specification Properties

The specification properties for AFS0 and AFS1 are:

\[ \text{AG}((\text{Client.belief} = \text{valid}) \rightarrow (\text{Server.belief} = \text{valid})), \text{and } (6.1) \]

\[ \text{AG}((\text{Server.belief} = \text{valid}) \rightarrow (\text{Client.belief} = \text{valid})). \text{ (6.2) } \]

In this file system design, the client belief leads the server belief. The first property represented in formula (6.1) is true after model checking for AFS0. The 2nd property represented in formula (6.2) is false. The second specification property has been deliberately chosen to fail with this file system design. Thus, after our model updating, we do not need to consider the logic outcome of the updated models under the false specification property. From Definition 2.3, formula (6.2) is equivalent to

\[ \text{AG}((\text{Server.belief} = \neg \text{valid}) \lor (\text{Client.belief} = \text{valid})). \text{ (6.3) } \]
6.2.4 Extracting the Kripke Models of AFS0 and AFS1 from NuSMV

The Kripke model to be updated is incorporated in the source code of the CTL model updater based on the reachable states and state transitions determined from SMV analysis of the model. In the previous papers [84, 86], there are state transition diagrams (Figure 6.5 and 6.6) for AFS0 and AFS1 with client and server modules. There are also SMV model definitions as input to the SMV model checker for these models. The diagrams and SMV definitions separate the client and server as two modules in each example, thus we cannot capture what all values of variables are at certain times during transitions in a model. This feature is usually contained in Kripke structures, where all values of variables are shown at each state simultaneously. We have used NuSMV [12, 67] to derive Kripke structures for loaded models.

The resulting Kripke structures of AFS0 and AFS1 are shown in Figure 6.7 and Figure 6.8. The detailed extraction methods are presented in Appendix B. This method can also be used for extracting any other Kripke model. Here, we give a simple summary of the principle of the extracting method as follows:

First, NuSMV is started up in interactive mode with the model (AFS1) to be examined by the command “NuSMV -int AFS1.smv”. Next, NuSMV loads and processes the model using the command “go”. To display all possible states, the command “print_fair_states -v” is used. The command “print_reachable_states -v” shows reachable states in a model. We use one of the proceeding commands to serve as a means of state identification. We next retrieve the state transition relationships. The initial states are identified first by the command “pick_state -i -a”. Then one of the listed initial states is selected to start tracing its successors and paths. The command “simulate -i -a 1” is repeated until a loop (repeated states) shows up. Thus, one path has been traced. All states shown under command “simulate -i -a 1” should be selected one by one to retrieve the whole subset of paths under an initial state. Then another initial state is selected, other paths after this initial state are traced in the same fashion as those paths after the first selected initial state. Eventually all paths from all initial states in a model are recorded. Accordingly, a complete Kripke model containing reachable states is extracted from NuSMV. To
Three Case Studies

verify the trace, the command “show_traces -v” shows all states which have been traced in the above methods. To aid identifying individual states, a modification was implemented in the BddEnc.c file in the NuSMV source tree to streamline the presentation of the variable values for each state.

In the AFS0 Kripke model shown in Figure 6.7, there are 6 reachable states, where state 2 is an initial state, and 10 transitions in between the reachable states. The model contains 4 variables with two possible values. The variables are “Client.out”, which ranges over \{0, fetch\}, “Client.belief” over \{valid, no file\}, “Server.out” over \{0, val\} and “Server.belief” over \{none, valid\}. In this figure, each state contains a list of values for all variables in this model arranged in the order of #1,#2,#3 and #4 as shown in the figure.

The AFS1 Kripke model is shown in Figure 6.8. There are 26 reachable states, where states 11, 12, 13 and 14 are initial states, and 52 transitions in between the reachable states. The model contains 5 variables and each individual variable has 2 to 4 possible values. The variables are “Client.out”, which ranges over
Another Two Models – AFS0 and AFS1

{0, fetch, validate}, “Client.belief” over {valid, invalid, suspect, nofile}, “Server.out” over {0, val, inval}, “Server.belief” over {none, valid, invalid} and “Server.valid-file” over {true, false}. These values are extracted from those in the AFS1 state transition diagram shown in Figure 6.6 and each has an equivalent context there. “Client.belief” and “Server.belief” are the beliefs for client and server; “Client.out” means that the server module takes a parameter input that can be any message coming from the client; “Server.out” means that the client module takes a parameter input that can be any message coming from the server. The Server.valid-file is used when the client has a suspect file in its cache and requests a validation from the server. If an update by some other client has occurred then the server reflects that by nondeterministically setting the value of valid-file to 0; otherwise, 1 (the file cached at the client is still valid).
6.2.5 Applying the Model Updater to AFS0 and AFS1

Parsing the specification strings

The model updater has been successfully applied to the models AFS0 and AFS1. First, we describe the parsed structure of the specification properties for AFS0 and AFS1, then we implement model checking, and model update.

The specification string \( \text{AG}((\text{Client.belief}=\text{valid}) \rightarrow (\text{Server.belief}=\text{valid})) \) (formula (6.1)) is parsed into the structure as follows:

```
primary pair  ->
  negate1 ............  false
  negate2 ............  false
  navigate ............  5
  operator ............  21
  operand1 ............  0x00000000
  operand2 ............  0x00000000

nested pair1  ->
  negate1 ............  false
  negate2 ............  false
  navigate ............  0
  operator ............  25
  operand1  ->
    negate1 ............  false
    negate2 ............  false
    navigate ............  0
    varindex ............  102
    valindex ............  0
    error .............  false
  operand2 ............  0x00000000
  nestedpair1 .......  0x00000000
  nestedpair2 .......  0x00000000
  error .............  false
```
According to our previous parsing rules in section 5.4, “AG” is interpreted as “navigate · · · 5” at the 4th line of the parsed structure. The part of the string after AG is interpreted into a couple of pairs, each pair structure accommodating one named variable: “nested pair1” → “nested pair2”, where, for example, “nested pair1” represents the part of string “(Client.belief = valid)” and “nested pair2” represents “(Server.belief = valid)”. “operator · · · 21” before “nested pair1” in the parsed structure interprets the operator “implies” (→) in the string. We use “nested pair1” as the example to specify the contents inside this pair. The same principle is applied to “nested pair2”. Under “nested pair1”, a sub nested pair contains “operand1”. “operand1” is the interpretation of the part of string “(Client.belief = valid)”, where “=” is interpreted as ”operator · · · 25” before “operand1”. “operand1” contains an atom which interprets the variable index number in the variable array and the value index number in the value array. For exam-
ple, the index of variable “Client.belief” is 102 and the index of value “valid” of “Client.belief” is 0.

Models AFS0 and AFS1 satisfy the specification string in formula (6.1). Thus, we do not need to update the models for this string. The models AFS0 and AFS1 do not satisfy the specification string in formula (6.2). We will perform model update on both models in relation to the string in formula (6.2), which is $AG((\text{Server.belief}=\text{valid}) \rightarrow (\text{Client.belief}=\text{valid}))$. This string is parsed into the structure as follows:

primary pair  ->
    negate1 ............ false
    negate2 ............ false
    navigate ............ 5
    operator ............ 21
    operand1 ............ 0x00000000
    operand2 ............ 0x00000000

nested pair1  ->
    negate1 ............ false
    negate2 ............ false
    navigate ............ 0
    operator ............ 25
    operand1  ->
        negate1 ............ false
        negate2 ............ false
        navigate ............ 0
        varindex ............ 104
        valindex ............ 1
        error ............ false
        operand2 ............ 0x00000000
    nestedpair1 ....... 0x00000000
    nestedpair2 ....... 0x00000000
    error ............ false
Because this string has a similar parsed structure as the previous one, we do not describe its parsed structure in detail again.

Model Checking

After translating the specification strings into parsed structures as above, we then select the required values from the structures. The “varindex” and “valindex” in the parsed structure map the corresponding values of a variable. We then use model checking functions to compare the indexes of values of different indexes of variables in states with those in parsed structures of the specification properties. For the AFS0 and AFS1 models, both variables: “Client.belief” and “Server.belief = valid” required by the specification properties for model checking are the same #2 and #4 orders among variable sequences of these two models.

We can use either formula (6.2) or (6.3) to identify the states in the AFS0 kripke model which do not satisfy this specification property. If we use formula (6.2), we
just need to locate all states which have “Server.belief = valid”. Here these states are 5, 8, 13 and 16. Then we check whether “Client.belief = valid” or not in these states. If “Client.belief = valid”, then the state satisfies the specification property. Otherwise, the state does not. Here, we can observe that states 5 and 13 satisfy the specification property. States 8 and 16 do not. If we use formula (6.3), we just need to identify all states with either “Server.belief = ¬valid” or “Client.belief = valid”. If a state does not have either features, then it does not satisfy the specification property. Both methods can prove the correctness of each other. Since the AFS0 model extracted from NuSMV contains both reachable and non reachable states, we should filter out the non reachable states before model checking. For AFS0, states 8 and 16 are identified as false reachable states. States 2, 10, 13, 5 are identified as correct reachable states.

The specification properties to be checked for AFS1 are the same as those for AFS0. AFS1 satisfies formula (6.1) and does not satisfy formulas (6.2) and (6.3) in a similar fashion to the AFS0 case. The methods of model checking for this model are the same as those for AFS0. The AFS1 model extracted from NuSMV contains reachable states only. The false reachable states are identified as 19, 20, 23, 24, 7 and 8 straightaway after model checking. The next step is to update these states to produce admissible models which satisfy the specification property formula (6.2).

Model Update

Now, we start to update the AFS0 Kripke model to make it satisfy the specification property. According to Theorem 3.6 and its corresponding algorithm, the number of possible admissible models is \((C^2_3)^2 = 3^2 = 9\) by using the following methods: PU2, PU5 or PU3 are applied to state 8 on paths \{2, 10, 8, ⋯, 5\}; PU2, PU5 or PU3 are applied to state 16 on paths \{2, 10, 16, ⋯, 5\}. Among the 9 admissible models, \((C^2_1)^2 = 4\) admissible models only have reachable states \{2, 10\} (Figure 6.9), which are from the update of applying PU2 or PU5 to states 8 and 16. The other \(2 \times (C^1_3 \times C^3_1) - (C^1_1)^2 = 2 \times 3 - 1 = 5\) admissible models have either 5 or 6 reachable states. The combination of reachable states could be \{2, 10, 8’, 16’, 13, 5\} (the Kripke structure of this model is the same as the original AFS0), \{2, 10, 16’, 13, 5\} (Figure 6.10, where
state 160 is 16’. or \{2, 10, 8’, 13, 5\} (Figure 6.11, where state 80 is 8’).

For AFS1, the derived admissible models totaled \((C_3^3)^6 = 3^6 = 729\) according to Theorem 3.6 and its corresponding algorithm. The reason for the large number of admissible models is because the 6 false states are spread on 6 different paths (in reality, there are more than 6 false paths, but we only need to update a maximum of 6 paths. Other paths with false states partially overlap with these 6.). Then, each path has 3 possible update methods: either PU2, PU5 or PU3 applied to each state on each path once. Because there are too many admissible models, we do not list each of them here. The implementation of AFS0 and AFS1 initiates a further research topic in the next chapter which narrows down the number of admissible models according to reachable state characteristics.
Three Case Studies

Figure 6.11: The 3rd Admissible Model of AFS0
Chapter 7

Study of a Model Explosion

Problem

7.1 An Admissible Model Explosion Problem

There are many more admissible models than we expect. This phenomena is called *admissible model explosion*. We should classify the admissible models and minimize the number of admissible models. In principle, we should preserve the maximum reachable states in an admissible model to preserve the structure of the original model during an update. We first denote some terms for frequently used concepts before the analysis in this chapter. If a state is accessible by a sequence of transitions starting from an initial state, it is called a *reachable state*; if reachable states in an original model are not updated by PU1, PU2, PU3, PU4 and PU5 operations after their model is updated, they are called *unchanged states*; if the reachable states in an admissible model are also in the original model, they are called *unchanged reachable states*. Unchanged reachable states are a subset of unchanged states; unchanged reachable states are also a sub set of reachable states; reachable states and unchanged states are not subsets of each other. We use “updated\(_i\) (model name)” to denote the set of totalled reachable states in an admissible model, which will be in “case 1 (model name)” for classifying the set of all reachable states in an admissible model. “case 2 (model name)” is for classifying the set of all unchanged reachable states in an admissible model. If the set of unchanged reachable states is the largest of all state sets of admissible models, it is expressed in \(Set_{\text{max}}\), and if the smallest set, in \(Set_{\text{min}}\).
7.1.1 Classifying Admissible Models by Reachable State Characteristics

“AG” is the most important and frequently used CTL operator in model checking. We focus our method to solve an admissible model explosion problem derived from Theorem 3.6 and its corresponding algorithm, which have been used for updating all selected models. These models are checked with specification properties containing the CTL operator “AG”. Updates for other semantics containing CTL operators such as AX, AF, AU and EG will be investigated later. Before we derive a proper formal method, we analyze the three models: the microwave oven, AFS0 and AFS1 models by their reachable states characteristics.

In the microwave oven model, there are reachable states \(\{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}\), where states \(s_2\) and \(s_5\) do not satisfy their specification property. The two false states are on one path. There are 3 admissible models from update results. Two admissible models with reachable states \(\{s_1, s_3, s_4, s_6, s_7\}\) are from the updates where PU2 or PU5 are applied to the false path, and one of the admissible models with reachable states \(\{s_1, s'_2, s_3, s_4, s'_5, s_6, s_7\}\) from the update where PU3 is applied to all false states \((s_2\) and \(s_5)\) on the false path. We summarize the sets of totalled reachable states for different admissible models as follows:

case 1 (m-oven):
updated\(_1\)(m-oven) = \(\{s_1, s_3, s_4, s_6, s_7\}\), and
updated\(_2\)(m-oven) = \(\{s_1, s'_2, s_3, s_4, s'_5, s_6, s_7\}\).
Thus, updated\(_1\)(m-oven) \(\subset\) updated\(_2\)(m-oven);

case 2 (m-oven):
updated\(_1\)(m-oven) = updated\(_2\)(m-oven) - \(\{s'_2, s'_5\}\) = \(\{s_1, s_3, s_4, s_6, s_7\}\).

In AFS0, two states 8 and 16 do not satisfy their specification property. The two states are scattered on different paths. The admissible models and their reachable states have been presented in section 6.2.5. We summarize the sets of totalled reachable states for different admissible models here:

case 1 (AFS0):
§7.1 An Admissible Model Explosion Problem

Updated1(AFS0) = \{2, 10\},
Updated2(AFS0) = \{2, 10, 8', 13, 5\},
Updated3(AFS0) = \{2, 10, 16', 13, 5\}, and
Updated4(AFS0) = \{2, 10, 8', 16', 13, 5\},
where we observe:
Updated1(AFS0) ⊆ Updated2(AFS0),
Updated1(AFS0) ⊆ Updated3(AFS0),
Updated1(AFS0) ⊆ Updated4(AFS0),
Updated2(AFS0) ⊆ Updated4(AFS0), and
Updated3(AFS0) ⊆ Updated4(AFS0).

From the sets listed above, the minimum set is Updated1(AFS0) and the maximum set is Updated4(AFS0).

Case 2 (AFS0):
Set_{min} = Updated1(AFS0) = \{2, 10\},
Set_{max} = Updated2(AFS0) - \{8'\}
= Updated3(AFS0) - \{16'\}
= Updated4(AFS0) - \{8', 16'\} = \{2, 10, 13, 5\}.

Here, the minimum set is Set_{min} and the maximum set is Set_{max}.
In AFS1, 6 states do not satisfy their specification property. The 6 states are 19, 20, 23, 24, 7 and 8, which are scattered on 6 different paths. Before we examine the combination of reachable states, we divide the model into two self contained sub models for the convenience of analysis. One is the model on the left side of the Kripke model in Figure 6.8 and contains states \{11, 12, 17, 18, 19, 20, 25, 26, 15, 16\}, where 11 and 12 are initial states. We call it AFS1-1 as shown in Figure 7.1. The other sub model is on the right side of the Kripke model in Figure 6.8 and contains states \{13, 14, 6, 5, 3, 4, 1, 2, 22, 21, 23, 24, 25, 26, 7, 8, 9, 10, 15, 16\}, where 13 and 14 are initial states. We call it AFS1-2 as shown in Figure 7.2.

For AFS1-1, we have a total of \((C_3^2)^2 = 9\) admissible models. \((C_3^2)^2 = 4\) admissible models contain the same set of reachable states in case 1 (AFS1-1):

\[
\text{updated}_1 (\text{AFS1-1}) = \{11(\text{or} 12), 17, 18\};
\]

if either states 19 and 20 or their previous relations are removed by PU5 or PU2. Thus, the false paths, where states 19 and 20 are located, are all cut off. This causes the original reachable states in AFS1-1, which are successors of states 19 and 20, to be unreachable after update. The alternative \(2 \times (C_3^1 \times C_3^1) - C_1^1 = 5\) admissible
models contain the following sets of reachable states:
updated$_2$(AFS1-1) = \{11(or12), 17, 18, 19', 25, 26, 15, 16\},
updated$_3$(AFS1-1) = \{11(or12), 17, 18, 20', 25, 26, 15, 16\} and
updated$_4$(AFS1-1) = \{11(or12), 17, 18, 19', 20', 25, 26, 15, 16\},
where 19' and 20' are the updated states from states 19 and 20 updated with PU3.
The admissible models with updated$_2$(AFS1-1) are the results from 2 combinations of updates where PU3 is applied to state 19 to derive state 19', then state 20 is updated with PU2 or PU5; the admissible models with the state set updated$_3$(AFS1-1) are from updates where PU3 is applied to state 20 and PU2 or PU5 to state 19; the admissible model with the state set updated$_4$(AFS1-1) is from the update result where PU3 is applied to states 19 and 20. We observe:
updated$_1$(AFS1-1) $\subseteq$ updated$_2$(AFS1-1),
updated$_1$(AFS1-1) $\subseteq$ updated$_3$(AFS1-1) and
updated$_1$(AFS1-1) $\subseteq$ updated$_4$(AFS1-1);
updated$_2$(AFS1-1) $\subseteq$ updated$_4$(AFS1-1) and
updated$_3$(AFS1-1) $\subseteq$ updated$_4$(AFS1-1).
From these results, we can easily identify that the biggest set of reachable states is updated$_4$(AFS1-1), and the smallest is updated$_1$(AFS1-1).

Alternatively, we count unchanged reachable states and ignore all updated states with PU3 in admissible models such as states 19' and 20'. We observe

\begin{align*}
\text{case 2 (AFS1-1):} \\
Set_{min} &= \text{updated}_1(\text{AFS1-1}) = \{11(or12), 17, 18\}, \\
Set_{max} &= \text{updated}_2(\text{AFS1-1}) - \{19'\} \\
&= \text{updated}_3(\text{AFS1-1}) - \{20'\} \\
&= \text{updated}_4(\text{AFS1-1}) - \{19', 20'\} = \{11(or12), 17, 18, 25, 26, 15, 16\},
\end{align*}
where, $Set_{min} \subseteq Set_{max}$.

For model AFS1-2, the reachable states for each admissible model could be one of the sets in
\begin{align*}
\text{case 1 (AFS1-2):} \\
\text{updated}_1(\text{AFS1-2}) &= \{13(or14), 6, 5, 3, 4, 1, 2, 22, 21, 25, 26, 15, 16\}, \\
\text{updated}_2(\text{AFS1-2}) &= \{13(or14), 6, 5, 3, 4, 1, 2, 22, 21, 23', 25, 26, 15, 16\},
\end{align*}
where the admissible model with the maximum set of reachable states contains
updated\textsubscript{16}(AFS1-2) and the admissible model with the minimum set of reachable
states contains updated\textsubscript{1}(AFS1-2). The admissible model with updated\textsubscript{16}(AFS1-2)
is updated using the method where PU3 is applied to all false states on all false
paths in the admissible model. The admissible model with updated\textsubscript{1}(AFS1-2) are
updated with the method where PU2 or PU5 are applied to all false paths. All other
admissible models are from the update where PU3 is applied to either state 7 or 8
at least once.

Next, in case 2 (AFS1-2), all states updated with PU3 in an admissible model are
removed from the reachable state sets in case 1 (AFS1-2). The remaining reachable
states in each model can be divided into two groups in
case 2 (AFS1-2):
\[
Set_{\text{min}} = \text{updated}\textsubscript{1}(AFS1-2)
\]
\[
= \text{updated}\textsubscript{2}(AFS1-2) - \{23'\}
\]
\[
= \text{updated}\textsubscript{3}(AFS1-2) - \{24'\}
\]
\[
= \text{updated}\textsubscript{6}(AFS1-2) - \{23', 24'\}
\[\text{Set}_{\text{max}} = \text{updated}_4(AFS1-2) - \{7\}\]
\[= \text{updated}_5(AFS1-2) - \{8\}\]
\[= \text{updated}_7(AFS1-2) - \{23', 7'\}\]
\[= \text{updated}_9(AFS1-2) - \{23', 8'\}\]
\[= \text{updated}_9(AFS1-2) - \{24', 7'\}\]
\[= \text{updated}_{10}(AFS1-2) - \{24', 8'\}\]
\[= \text{updated}_{11}(AFS1-2) - \{7', 8'\}\]
\[= \text{updated}_{12}(AFS1-2) - \{23', 24', 7'\}\]
\[= \text{updated}_{13}(AFS1-2) - \{23', 24', 8'\}\]
\[= \text{updated}_{14}(AFS1-2) - \{23', 7', 8'\}\]
\[= \text{updated}_{15}(AFS1-2) - \{24', 7', 8'\}\]
\[= \text{updated}_{16}(AFS1-2) - \{23', 24', 7', 8'\}\]
\[= \{13(\text{or}14), 6, 5, 3, 4, 1, 2, 22, 21, 25, 26, 9, 10, 15, 16\}, \quad \text{and}\]
\[\text{Set}_{\text{max}} - \text{Set}_{\text{min}} = \{9, 10\}.\]

We can extract much more information from updating AFS1-2 than previous models. Our strategy is to preserve all models among \(\text{Set}_{\text{max}}\), where states 9 and 10 are protected during the update. Thus, updates which do not remove states 7 and 8 (for case PU5) or their previous relations (for case PU2) at the same time are selected.

### 7.1.2 Analyzing Admissible Models

We have illustrated 3 examples by classifying the sets of reachable states in the admissible models which are derived from Theorem 3.6. The principle of narrowing down the admissible models is to identify those preserving the maximum reachable states. We choose the admissible models with the largest set of reachable states and discard the admissible models with the smallest set of reachable states. We have classified the sets of reachable states with two different methods (case 1 and case 2) for each example. In case 1, we compare all reachable states of the admissible models. For each example in case 2, the states updated by PU3 are not considered within the set of reachable states of admissible models. We only compare the unchanged
reachable states in the sets of reachable states.

In the microwave oven model, the 2 false states belong to one path. Thus, even if we have 3 different update methods for each path according to Theorem 3.6, we only have a total of 3 different admissible models for this model. According to case 1 (m-oven), we have an admissible model with state set updated$_2$(m-oven), which contains maximum reachable states. This model is derived from the update method where PU3 is applied to all false states on a path. This method only preserves one admissible model out of a total of 3. According to case 2 (m-oven), an extra 2 admissible models with state set updated$_1$(m-oven) are derived from the update where PU5 or PU2 are applied to state $s_2$ or its previous relation. The identification method in case 2 (m-oven) preserves all 3 admissible models.

In the AFS0 model, after model checking, there are two false states on different paths. False states scattered on different paths increase the number of admissible models. The number of admissible models derived from Theorem 3.6 is $(C_3^1)^2 = 9$. According to case 1 (AFS0), the admissible model which contains the maximum reachable states has state set updated$_4$(AFS0). This model is derived from the update method where PU3 is applied to all false states. According to case 2 (AFS0), the narrowed down admissible models are those with reachable state sets updated$_2$(AFS0), updated$_3$(AFS0) and updated$_4$(AFS0). These admissible models are the result of applying PU3 to at least one of the 2 false paths (Each false state is on a different path). The number of narrowed admissible models is $C_1^1 \times C_1^3 + C_1^3 \times C_1^1 - C_1^1 \times C_1^1 = 3 + 3 - 1 = 5$. In this formula, $C_1^1 \times C_1^3$ is the number of updated models for the combination of update operations when PU3 is applied to state 8 and PU2, PU3 or PU5 are applied to state 16; $C_1^3 \times C_1^1$ is the number of updated models for the combination of update operations when PU3 is applied to state 16, and PU2, PU3 or PU5 are applied to state 8; $C_1^1 \times C_1^1$ is the duplicate number between the previous two numbers if PU3 is applied to state 8 and 16 at the same time. $(C_1^2)^2 = 4$ admissible models are discarded. These models are derived by applying either PU2 or PU5 to states 8 and 16, thus the models do not preserve the maximum reachable states.

In AFS1-1, the two false states occur scattered on different paths, similar to
AFS0. In case 1 (AFS1-1), the biggest set is updated$_4$(AFS1-1). The admissible model with this set is derived from the update where PU3 is applied to all false states on different paths. With this method, only this admissible model is preserved. All other admissible updates for AFS1-1 are discarded. In case 2 (AFS1-1), the admissible models with state sets updated$_2$(AFS1-1), updated$_3$(AFS1-1) and updated$_4$(AFS1-1) have maximum unchanged reachable states. In this case, these 5 admissible models are preserved out of a total of 9 admissible models.

In AFS1-2, the outcome is more complex than that of AFS0 and AFS1-1. The 4 false states are scattered on 4 different paths. There are $(C^4_1)^4 = 81$ admissible models after update. According to case 1 (AFS1-2), the narrowed admissible model which preserves maximum reachable states has state set update$_{16}$(AFS1-2) derived from PU3 applied to all false states. According to case 2 (AFS1-2), the narrowed admissible models which contain maximum unchanged reachable states have state sets update$_4$(AFS1-2), update$_5$(AFS1-2), update$_7$(AFS1-2), \ldots, and update$_{16}$(AFS1-2). According to this method, $(C^2_1)^2 \times (2 \times (C^1_1 \times C^3_1) - (C^1_1)^2) = 45$ admissible models preserve the maximum unchanged reachable states, where $2 \times (C^1_1 \times C^3_1) - (C^1_1)^2$ is the number of combinations of different update results if PU3 is applied to at least one of states 7 or 8, then PU2, PU3 or PU5 are applied to the other state; $(C^3_1)^2$ is the number of combinations of different updates, PU2, PU3 and PU5, on states 23 and 24. $(C^2_1)^2 \times (C^2_2)^2 = 36$ admissible models do not preserve the maximum unchanged reachable states, which is from applying either PU2 or PU5 to states 7 and 8.

For AFS1, the number of admissible models is $(C^3_1)^6 = 3^6 = 729$. By using the method in case 2, the updated models which can preserve the maximum unchanged reachable states are narrowed down to $(2 \times (C^1_1 \times C^3_1) - (C^1_1)^2)^2 \times (C^2_1)^2 = 2 \times 5 \times 3^2 = 225$, where $(2 \times (C^1_1 \times C^3_1) - (C^1_1)^2)^2$ is the number of combinations of different update operations on a pair of states 19 and 20 in sub model AFS1-1 and a pair of states 7 and 8 in sub model AFS1-2 respectively; If PU3 is applied to one of the states (path) between the pair, then PU2, PU3 and PU5 can be applied to the other state (path); $(C^1_1)^2$ after “−” sign is the duplicate number of combinations of updates where PU3 is applied to both states of each pair; $(C^3_1)^2$ is the number of combinations of
different update operations, PU2, PU3 and PU5, on states 23 and 24. The number of admissible models which do not preserve the maximum unchanged reachable states is $2 \times ((\binom{C_2}{1})^2 \times (\binom{C_3}{1})^2) - (\binom{C_2}{1})^2 \times (\binom{C_3}{1})^2 = 2 \times (4 \times 9^2) - 4^2 \times 9 = 504$, where $(\binom{C_2}{1} \times C_2^2) \times (\binom{C_3}{1} \times C_3^2)$ is the number of admissible models for two cases: if states 19 and 20 are updated with either PU2 or PU5, or if states 7 and 8 are updated with either PU2 or PU5; $((\binom{C_2}{1})^2 \times (\binom{C_3}{1})^2$ is the duplicate number of admissible models which result from the previous two cases where all 4 states 19, 20, 7 and 8 are either updated with PU2 or PU5. After using the method in case 2, 504 admissible models have been discarded because they don’t preserve the maximum unchanged reachable states. To confirm the correctness of the above calculation, the total number of admissible models is $225 + 504 = 729$.

From these results, we observe that under case 1, the number of admissible models are narrowed down to a final admissible model where all false states are updated with PU3 only. All other admissible models are discarded. Compared with the total number of admissible models, the method in case 2 can accommodate more than one but fewer admissible models, where for each model at least one of the false paths consisting of all false states are updated with PU3. After model update, this approach preserves admissible models with all original unchanged reachable states which succeed the original false states. With this method, we do not focus on which update methods have been used in the update, but we are concerned whether this particular update method affects the reachability of the states which are not updated.

We use the following approach to decide which method we will adopt to narrow down the admissible models. The admissible models with a minimal set of reachable states should be discarded to avoid an update consisting of only PU2 and PU5 which remove state transitions and states. The function and structure of the original model should be protected as much as possible. An updated system should have more than one admissible model, as this gives options which can be adjusted to suite the technical environment. We consider that the comparison method in case 1 discards too many admissible models and only preserves one admissible model. This approach provides a real system with no option, although it can protect the original states and
the structure of the original model. Thus, we consider that the comparison method in case 2 is more reasonable. This method protects the structure of the original model and also provides more options of admissible models. Accordingly, case 2 is our ideal choice.

**Observation 1** Any admissible model updated with PU2 or PU5 has the risk of not retaining its maximum unchanged reachable states, which are successors of the false states updated with PU2 or PU5 in the original model.

### 7.2 Solving the Admissible Model Explosion Problem

#### 7.2.1 Minimal Change with Maximal Reachable States

Given a Kripke model \( M = (S, R, L) \) and \( s_0 \in S \), and let \( M = (M, s_0) \). We say that \( s^r \) is a reachable state of \( M \), if there is a path in \( M = (S, R, L) \) of the form \( \pi = [s_0, s_1, \ldots] \) where \( s^r \neq s_0 \) and \( s^r \in \pi \).\(^1\) We use \( RS(M) \) to denote the set of all reachable states of \( M \). An original model \( M \) contains reachable states \( RS(M) = \{ s^u | s^u \text{ is a state that will not be changed during an update} \} \cup \{ s_{\text{update}} | s_{\text{update}} \text{ is a state that will be updated during an update} \} = S^u \cup S_{\text{update}}. \)

It is easy to see that we can express \( S^u = S^u_{\text{pre}} \cup S^u_{\text{succ}} \), where \( S^u_{\text{pre}} \) are the set of unchanged reachable states preceding all states in \( S_{\text{update}} \) from an initial state, and \( S^u_{\text{succ}} \) are the set of states succeeding states in \( S_{\text{update}} \). If the update \( \text{Update}_\phi(M) \) for \( M \) to satisfy \( \phi \) preserves \( S^u_{\text{succ}} \) and the update is admissible, then these final admissible models are called committed models. Now we propose a refined CTL model update principle which can significantly reduce the number of admissible models:

**Definition 7.1** (Minimal change with maximal reachable states) Given a CTL Kripke model \( M = (S, R, L) \), \( M = (M, s_0) \) where \( s_0 \in S \), and a CTL formula

\(^1\)Requiring \( s^r \neq s_0 \) is only from a technical consideration for our algorithm design.
\( \phi \), a resulting model \( \text{Update}(\mathcal{M}, \phi) \) is called committed with respect to the update of \( M \) to satisfy \( \phi \), if the following conditions hold: (1) \( \text{Update}(\mathcal{M}, \phi) = \mathcal{M}' = (\mathcal{M}', s_0') \) is admissible; and (2) there is no other resulting model \( \mathcal{M}'' = (\mathcal{M}'', s_0'') \) such that \( \mathcal{M}'' \models \phi \) and \( \mathcal{R}(\mathcal{M}) \cap \mathcal{R}(\mathcal{M}') \subset \mathcal{R}(\mathcal{M}) \cap \mathcal{R}(\mathcal{M}'') \).

We use AFS0 as an example. There are 9 admissible models of AFS0. 5 admissible models out of the 9 have maximum unchanged reachable states \( \mathcal{R}(\text{AFS0}, 2) = \{2, 10, 13, 5\} \). The Kripke structures containing only unchanged reachable states of these 5 admissible models are shown in Figure 6.10 and 6.11, and Figure 6.7, where states 8 and 16 should be 8' and 16' instead. All other admissible models have fewer unchanged reachable states than these models.

### 7.2.2 The Reachable State Algorithm

We have three options to update an original model to a committed model:

Option 1: We derive all admissible models \( M' = \text{update}_{AC}(M, s_0) \) first. Then, we narrow down the admissible models by comparing the unchanged reachable states in these models with the original model \( M \). The final committed models are those that have the same number of original unchanged reachable states as in \( M \) out of all admissible models. In this method, the maximal unchanged reachable state principle is performed after the admissible models are derived.

Option 2: Theorem 3.6 should be re-characterized under the new constraint of the maximal reachable state principle. If we update a false path with Theorem 3.6, we should examine whether this method causes the exclusion of original reachable states (searching the successors of the updated states). If it does, then an alternative atomic update method should be tried. All possible sequences of false path updates should be performed to derive the full set of committed models.

Option 3: This method is the same as option 2 but without the consideration of order of false paths. The derived committed models for this option could be a subset of the full set of committed models.

The major reason to finalize committed models is to reduce the admissible model explosion effect and simplify the update procedure. It is obvious that option 1 cannot reduce the complexity of the update process. Thus, it should be abandoned.
Option 2 and 3 are more reasonable in that they simplify the update. The only difference between option 2 and 3 is that option 2 considers all committed models derived from all possible updates which preserve maximum reachable states and the final committed models are a complete set. However, the final committed models in option 3 are a subset of the resulting committed models in option 2. Option 3 is the most appropriate choice, as it provides the simplest algorithm, which in turn simplifies coding and reduces the machine processing time. It is less important that the committed models are a complete set. Thus, option 3 is chosen as our final algorithm to derive committed models. Before starting the reachable state algorithm, we describe some notions used in the algorithm. An original model $\mathcal{M} = (M, s_0)$ is updated to a model $\mathcal{M}' = (M', s_0)$. During the update, if some reachable states in $\mathcal{M}$ are updated with operations PU3 or PU5 which involve replacing or removing original states in an original model, these replaced or removed states in the original model are expressed as $S_{\text{update}}$, where $\text{update}$ can be a combination of PU3 and PU5; for example, $S_{\{\text{PU3,PU5}\}}$ represents the states in $\mathcal{M}$ which will be updated by operations PU3 and PU5. If some reachable states in $\mathcal{M}'$ are from the application of updates PU3 or PU4 on the original model $\mathcal{M}$, the set of these states are expressed as $S'_{\text{update}}$. For example, $S'_{\text{PU3}}$ represents states in the updated model $\mathcal{M}'$ which have been updated with PU3. Because PU1 and PU2 do not involve changing states, we do not need to denote changed states with these updates. We will use $\text{RS}(\mathcal{M}) - S_{\{\text{PU3,PU5}\}}$ to denote the unchanged reachable states in $\mathcal{M}$ and $\text{RS}(\mathcal{M}') - S'_{\{\text{PU3}\}}$ to represent unchanged reachable states in updated model $\mathcal{M}'$. However, PU1 and PU4 can increase the number of reachable states in an updated model compared with those in the original model. Thus, if a model involves update PU1 and PU4, the notation $\text{RS}(\mathcal{M}') - S'_{\{\text{PU1,PU4}\}}$ may not represent the unchanged reachable states in an updated model, but may be the total number of unchanged states and extra reachable states from update PU1 and PU4. Thus, if a model has been updated with PU1 or PU4, we should not use this notion. Now, we describe the reachable state algorithm to identify committed models as follows:

$$\text{Update}_{AG}(\mathcal{M}, \phi) \quad /\mathcal{M} \not\models AG\phi. \text{ Update } \mathcal{M} \text{ to satisfy } AG\phi./$$
if ℳ₀ = (ℳ, s₀) ⊭ ϕ, then PU3 is applied to s₀, else
(1) applying PU3 to all sᵢ ⊭ ϕ in ℳ;
(2) select a path π = [s₀, s₁, ⋯], where ∃s ∈ π, such that ℳᵢ = (ℳ, sᵢ) ⊭ ϕ;
   select the earliest state sᵢ ∈ π such that (ℳ, sᵢ) ⊭ ϕ;
   perform the following procedure:
   (2.1) applying PU2 to remove relation (sᵢ₋₁, sᵢ), obtain result ℳ', or
   (2.2) applying PU5 to remove state sᵢ and
       its associated relations, obtain result ℳ',
iff all unchanged reachable states in ℳ are still reachable in ℳ':
RS(ℳ) − S_{PU3,PU5} = RS(ℳ') − S'_{PU3}, else
(2.3) applying PU3 on all sᵢ ⊭ ϕ on π,
if ℳ' |= AGϕ, return ℳ'; else call {Update_AG(ℳ', ϕ)};

This algorithm gives two ways to derive committed models. One way is to apply
PU3 to all false states in a model. Another way is to apply PU5 or PU2 to the
first false state or its incoming transition on a path if the update can retain the
unchanged reachable states in the updated model. Otherwise, PU3 is applied to all
false states on this path. After a path is updated, the updated model is repeatedly
checked as to whether it satisfies the required property AGϕ. If it does not, the
function Update_AG recursively calls itself to update other paths until the updated
model satisfies the specification property. The final updated model is a committed
model.

7.3 Implementing the Reachable State Algorithm

7.3.1 Coding the Reachable State Algorithm

To avoid eliminating unchanged reachable states, we have implemented a reachable
state algorithm in code to embed the algorithm into the model updater as shown
in Figure 5.2. After all false reachable states are stored in the false reachable array
and all unchanged reachable states are stored in the unchanged reachable array, we
§7.3 Implementing the Reachable State Algorithm

apply PU5 to the first false reachable state (actually the first false state on a path). Next, we check if all unchanged reachable states are still reachable. If they are, then PU5 is an appropriate update for the first false reachable state (the first false path). Otherwise, PU3 should be performed on this false state (or all false states on a path). Then, PU5 is applied to the next false state in the false reachable array (actually the first false state on the 2nd false path). If all unchanged reachable states are still preserved, then PU5 is a suitable update on this false state (or this false path). Otherwise, PU3 should be applied to this false state (or all false states on this false path). The process continues until the last false reachable state (or last false path) in the false array is considered. Eventually, the updated model is one of the committed models. If we change the sequence of false states (actually false paths) in the false array, the other committed models will be derived. We do not need to perform our reachable state algorithm coding for the case where all false states in the original model are updated by PU3 to derive the committed model, because updating with PU3 does not create any unreachable states.

The method used to select reachable states is to check whether the total number of previous states “numprev” in a state structure shown in Figure 5.4 is 0 or not. If it is 0, then this state is not reachable. However, this method cannot be applied to update PU2. PU2 just removes the incoming relations of a false state and preserves the false state itself. After this update, the states after the false state or the false state itself could be unreachable starting from an initial state, but the states after the false states still have “numprev” being not 0, because the false state is still a previous state of its successors. However, PU2 and PU5 perform exactly the same function and derive the same updated models according to the concept of reachable states. Thus, we only need to perform PU5 to show both update PU2 and PU5.

Now, we apply the model updater with the inclusion of the reachable state algorithm to the AFS0 and AFS1 models. For AFS0 as shown in Figure 6.7, PU2 or PU5 are applied to state 8 first. Then, the intermediate updated model is checked as to whether it contains maximum unchanged reachable states. The checking result shows that the reachability of the unchanged states being successors of state 8 are preserved after this update. Next, PU2 and PU5 are applied to state 16. Then,
the updated model is checked again to identify all unchanged reachable states. Unfortunately, the successors (unchanged reachable states) of state 16 are missing. Thus, PU3 should be applied to state 16 instead. The admissible models from this method preserve all original unchanged reachable states. So, these admissible models are committed models such as shown in Figure 6.10. If we process the same principle but starting with state 16, we will get other admissible models such as shown in Figure 6.11.

For the AFS1 model, there are 4 initial states which can be grouped as 2 pairs, each member of the pair producing identical results. The above update methods are performed after this model is split into 2 sub models: AFS1-1 and af1-2. As shown in Figure 7.1, AFS1-1 contains states 11, 12, 17, 18, 19, 20, 25, 26, 15 and 16. As shown in Figure 7.2, AFS1-2 contains states 13, 14, 6, 5, 3, 4, 1, 2, 22, 21, 23, 24 7, 8, 9, 10, 25, 26, 15 and 16. The technique to separate AFS1 is to use the path information to sort out states for each sub model, because paths start from the initial states and end at the final states. For AFS1-1, we select all paths starting with initial state 11 and 12 and put all states on these paths into an array. Then, we filter out the repeated states from this array. The final non repeated states in this array are all states for AFS1-1. Also, we remove links (22,25), (23,25) and (24,25) for state 25, and (22,26), (23,26) and (24,26) for state 26 to isolate AFS1-1 from AFS1-2 to perform our reachable state algorithm. After the algorithm is performed, the links are added again for the AFS1-2 sub model. Strictly, we also perform the same procedure with relations (9,15), (10,15), (9,16) and (10,16). However, the reachability of states 15 and 16 in AFS1-1 is the same even if these relations are removed. So, we omit these procedures. For the AFS1-2 sub model, we select all paths starting with initial states 13 and 14. Then we perform the same technique as for AFS1-1 to store all states on these paths into an array. Then, we select all non repeated states in this array. These non repeated states are for the AFS1-2 sub model. Also, we remove the links (19,25) and (20,25) for state 25 and (19,26) and (20,26) for state 26 for AFS1-2, such that the AFS1-2 sub model is isolated to perform the reachable state algorithm. We do not need to perform the same procedure with relations (25,15), (26,15), (25,16) and (26,16) for states 15 and 16
in AFS1-2 for the same reason that applied for AFS1-1. After the AFS1 model is separated into two sub models, these two sub models are fed into the reachable state algorithm similar to the AFS0 model.

### 7.3.2 Simulation Results forUpdating AFS0 and AFS1

The **AFS0 Model**

The original information about the AFS0 model is presented on the screen:

State Machine Model

- Short Model name is AFS0
- Long Model name is Andrew File System Version 0

Variable name #1 is Client.out with 2 values -> 0, fetch
Variable name #2 is Client.belief with 2 values -> valid, nofile
Variable name #3 is Server.out with 2 values -> 0, val
Variable name #4 is Server.belief with 2 values -> none, valid

State Information for 16 states is ->

<table>
<thead>
<tr>
<th>Id</th>
<th>Initial</th>
<th>Values</th>
<th>Next Links</th>
<th>Previous Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>valid</td>
<td>0</td>
<td>none</td>
</tr>
<tr>
<td>2</td>
<td>***</td>
<td>nofile</td>
<td>0</td>
<td>none -&gt; 10 -&gt; 2 &lt;- 2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>valid</td>
<td>val</td>
<td>none</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>nofile</td>
<td>val</td>
<td>none</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>valid</td>
<td>0</td>
<td>valid -&gt; 5     &lt;- 8 &lt;- 13 &lt;- 16 &lt;- 5</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>nofile</td>
<td>0</td>
<td>val</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>valid</td>
<td>val</td>
<td>valid</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>nofile</td>
<td>val</td>
<td>valid -&gt; 5     &lt;- 10</td>
</tr>
<tr>
<td>9</td>
<td>fetch</td>
<td>valid</td>
<td>0</td>
<td>none</td>
</tr>
<tr>
<td>10</td>
<td>fetch</td>
<td>nofile</td>
<td>0</td>
<td>none -&gt; 8 -&gt; 16 &lt;- 2</td>
</tr>
<tr>
<td>11</td>
<td>fetch</td>
<td>valid</td>
<td>val</td>
<td>none</td>
</tr>
<tr>
<td>12</td>
<td>fetch</td>
<td>nofile</td>
<td>val</td>
<td>none</td>
</tr>
</tbody>
</table>
13 fetch valid 0 valid -> 5 <- 8 <-16
14 fetch nofile 0 valid
15 fetch valid val valid
16 fetch nofile val valid -> 5 -> 13 <- 10

Number of paths is 5
path 1 states -> 2 -> 10 -> 8 -> 5 -> 5
path 2 states -> 2 -> 2
path 3 states -> 2 -> 10 -> 16 -> 5 -> 5
path 4 states -> 2 -> 10 -> 8 -> 13 -> 5 -> 5
path 5 states -> 2 -> 10 -> 16 -> 13 -> 5 -> 5

This model prints out all reachable and unreachable states. The reachabilities of states are shown by the previous links of each state. The update results are:

Case 1: state 8 is updated with PU2 or PU5, state 16 is updated with PU3:

Variable name #1 is Client.out with 2 values -> 0, fetch
Variable name #2 is Client.belief with 2 values -> valid, nofile
Variable name #3 is Server.out with 2 values -> 0, val
Variable name #4 is Server.belief with 2 values -> none, valid

State Information for 15 states is ->

<table>
<thead>
<tr>
<th>Id</th>
<th>Initial</th>
<th>Values</th>
<th>Next Links</th>
<th>Previous Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>valid</td>
<td>0</td>
<td>none</td>
</tr>
<tr>
<td>2</td>
<td>***</td>
<td>nofile</td>
<td>0</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-&gt; 10 -&gt; 2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>valid</td>
<td>val</td>
<td>none</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>nofile</td>
<td>val</td>
<td>none</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>valid</td>
<td>0</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;- 13 &lt;- 5 &lt;- 160</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>nofile</td>
<td>0</td>
<td>valid</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>valid</td>
<td>val</td>
<td>valid</td>
</tr>
<tr>
<td>9</td>
<td>fetch</td>
<td>valid</td>
<td>0</td>
<td>none</td>
</tr>
<tr>
<td>10</td>
<td>fetch</td>
<td>nofile</td>
<td>0</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-&gt; 160 &lt;- 2</td>
</tr>
<tr>
<td>11</td>
<td>fetch</td>
<td>valid</td>
<td>val</td>
<td>none</td>
</tr>
</tbody>
</table>
Implementing the Reachable State Algorithm

Number of paths is 2
path 1 states -> 2 -> 10
path 2 states -> 2 -> 2

The corresponding Kripke structure of this committed model is shown in Figure 6.10. Alternatively, if we randomly change the sequence of false reachable states in the array, the implementation result will be

Case 2:

State Information for 15 states is ->

<table>
<thead>
<tr>
<th>Id</th>
<th>Initial</th>
<th>Values</th>
<th>Next Links</th>
<th>Previous Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>valid 0</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>***</td>
<td>nofile 0</td>
<td>none</td>
<td>10 -&gt; 2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>valid</td>
<td>val none</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>nofile 0</td>
<td>val none</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>valid</td>
<td>0 valid</td>
<td>&lt;- 13 &lt;- 5 &lt;- 80</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>nofile 0</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>valid</td>
<td>val valid</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>fetch</td>
<td>valid</td>
<td>0 none</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>fetch</td>
<td>nofile 0</td>
<td>none</td>
<td>80</td>
</tr>
<tr>
<td>11</td>
<td>fetch</td>
<td>valid</td>
<td>val none</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>fetch</td>
<td>nofile 0</td>
<td>val none</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>fetch</td>
<td>valid</td>
<td>0 valid</td>
<td>&lt;- 80</td>
</tr>
<tr>
<td>14</td>
<td>fetch</td>
<td>nofile 0</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>fetch</td>
<td>valid</td>
<td>val valid</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>0</td>
<td>valid</td>
<td>val valid</td>
<td>5 -&gt; 13</td>
</tr>
</tbody>
</table>

12 fetch nofile val none
13 fetch valid 0 valid -> 5 <- 160
14 fetch nofile 0 valid
15 fetch valid val valid
160 fetch valid val valid -> 5 -> 13 <- 10
where state 8 is updated with PU3 into state 80. State 16 is updated with PU2 or PU5. State 16 is no longer shown on the screen after the update. The corresponding Kripke structure of this committed model is shown in Figure 6.11. The other alternative admissible update is where both state 8 and 16 are updated with PU3. Because updating with PU3 does not create any unreachable states, we do not need to implement reachability checking for this update option. We only implement the cases which represent the marginal condition to avoid the risk of reachable states being eliminated.

The AFS1 Model

The original information about the AFS1 model before update is presented on the screen:

State Machine Model
Short Model name is AFS1
Long Model name is Andrew File System Version 1

Variable name #1 is Client.out with 3 values->0,fetch,validate
Variable name #2 is Client.belief with 4 values->valid,invalid,suspect,nofile
Variable name #3 is Server.out with 3 values->0,val,inval
Variable name #4 is Server.belief with 3 values->none,valid,invalid
Variable name #5 is Server.valid-file with 2 values->false,true

State Information for 26 states is ->

<table>
<thead>
<tr>
<th>Id</th>
<th>Initial Values</th>
<th>Next Links</th>
<th>Previous Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>validate invalid 0 invalid true</td>
<td>-&gt;21-&gt;22 &lt;- 3 &lt;- 4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>validate invalid 0 invalid false</td>
<td>-&gt;21-&gt;22 &lt;- 3 &lt;- 4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>validate suspect inval invalid true</td>
<td>-&gt;1 -&gt; 2 &lt;- 6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>validate suspect inval invalid false</td>
<td>-&gt;1 -&gt; 2 &lt;- 6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>validate suspect 0 none true</td>
<td>-&gt;7 -&gt; 8 &lt;-13&lt;-14</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>validate suspect 0 none false</td>
<td>-&gt;3 -&gt; 4 &lt;-13&lt;-14</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>validate suspect val valid true</td>
<td>-&gt;9 -&gt;10 &lt;- 5</td>
<td></td>
</tr>
</tbody>
</table>
§7.3 Implementing the Reachable State Algorithm

Because the number of paths is large, we do not list them in detail here. This model is updated under our reachable state algorithm and one of the results is as follows:

State Information for 22 states is ->

<table>
<thead>
<tr>
<th>Id</th>
<th>Initial Values</th>
<th>Next Links</th>
<th>Previous Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>validate invalid 0  invalid true</td>
<td>-&gt;21-&gt;22  &lt;-3&lt;-4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>validate invalid 0  invalid false</td>
<td>-&gt;21-&gt;22  &lt;-3&lt;-4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>validate suspect inval invalid true</td>
<td>-&gt; 1-&gt; 2  &lt;-6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>validate suspect inval invalid false</td>
<td>-&gt; 1-&gt; 2  &lt;-6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>validate suspect 0  none true</td>
<td>-&gt;80  &lt;-13&lt;-14</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>validate suspect 0  none false</td>
<td>-&gt; 3-&gt; 4  &lt;-13&lt;-14</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>validate valid 0  valid true</td>
<td>-&gt;15-&gt;16  &lt;-80</td>
<td></td>
</tr>
</tbody>
</table>
We do not list the updated paths in detail here because of the large number. In this result, state 19 is updated with PU2 or PU5, state 20 is updated with PU3 and the new substituted state is 200. State 23 and 24 are updated with PU2 or PU5. State 7 is updated with PU2 or PU5. State 8 is updated with PU3 and the new substituted state is 80. The Kripke structure of this committed model is as Figure 7.3. The alternative updates could be the following cases:

case a: all false states are updated with PU3, or

case b: state 19 is updated with PU3 and state 20 is updated with PU2 or PU5, state 7 is updated with PU3 and state 8 with PU2 or PU5, and states 23 and 24 are updated with PU2 or PU5, or

case c: One of states 23 and 24 are updated with PU3 and the other by PU2 or PU5, the update methods of other states can be either case a or b. We do not list them in this implementation description to save space. Their corresponding state sets can be found in section 7.1.1.
7.4 Characterizations and Complexities

7.4.1 Characterizations

According to Observation 1, any semantic characterization involving update PU2 or PU5 needs to be constrained by the maximal reachable state principle to result in committed models. Semantics involving PU2 or PU5 in their characterizations are AG, AX, AF, AU and EG. We re-characterize updates for these semantics in Chapter 3 as follows:

**Theorem 7.1** Let $M = (S, R, L)$ be a Kripke model and $M = (M, s_0) \not\models AG\phi$, where $s_0 \in S$ and $\phi$ is a propositional formula. Then an admissible updated model $M' = Update(M, AG\phi)$ can be obtained by the following: for each path starting from $s_0$, $\pi = [s_0, \cdots, s_i, \cdots]$:

1. if for all $s < s_i$ in $\pi$ where $s \models \phi$ but $s_i \not\models \phi$, PU2 is applied to remove relation $(s_{i-1}, s_i)$, or PU5 is applied to remove $s_i$ and its associated relations,
iff reachable states $RS(\mathcal{M})-S_{\{PU3,PU5\}}$ are still reachable in the new model $\mathcal{M}^\prime$:

$$RS(\mathcal{M})-S_{\{PU3,PU5\}}=RS(\mathcal{M}^\prime)-S^\prime_{\{PU3\}}$$

else,

2. $PU3$ is applied to all states $s_i$ in $\pi$ not satisfying $\phi$ to substitute $s$ with $s^* |\models \phi$ and $Diff(s,s^*)$ is minimal.

Theorem 7.1 is formalized from its corresponding reachable state algorithm in section 7.2.2 which constrains Theorem 3.6. Theorem 7.1 characterizes a few typical situations of generating committed models for updates with formula $AG\phi$. Basically, the theorem says that to obtain a committed model to satisfy $AG\phi$, we need to perform certain update operations on each false path starting from $s_0$, where some states do not satisfy $\phi$. We may apply $PU2$ or $PU5$ to remove the tail of a path where $\exists s \not|\models \phi$ if the update still retains all other reachable states in $\mathcal{M}$. Otherwise, we can always safely apply $PU3$ on all states which do not satisfy $\phi$ along a path to do substitution. The committed models could be derived by combinations of paths updated with case 1 and 2 or case 2 only which yield a committed model with all false states in the original model updated with $PU3$.

Proof. In this theorem, after update in case 1, for some false paths, the part succeeding the first false state on a false path is removed. The remaining path contains all true states satisfying specification properties $\phi$. After update in case 2, for the rest of the false paths, all false states on a false path are updated and become true states satisfying $\phi$. Also, after update, all unchanged reachable states in the original model are still reachable in updated models. Thus, after update in case 1 and case 2, the updated models are committed models.

Case 1 excluding the reachable state principle constraint and case 2 are the same as those in Theorem 3.6. We have proved that they are equal minimal changes according to Definition 2.6 and result in admissible models according to Definition 2.7. However, under the constraint of the reachable state principle for case 1, some updates derived from Definition 2.6 which cannot retain the maximal unchanged reachable states should not be processed. In Theorem 7.1, the minimal change rules are based on Definition 7.1 which contains Definitions 2.6 and 2.7.
Accordingly, update involving PU2 or PU5 in case 1 which is constrained by the reachable state principle is an equal minimal change as the update involving PU3 in case 2. Both retain maximal unchanged reachable states and result in committed models.

\[\square\]

**Theorem 7.2** Let \( M = (S, R, L) \) be a Kripke model, and \( \mathcal{M} = (M, s_0) \) \((s_0 \in S)\) and \( \mathcal{M} \not\models AF\phi \) where \( \phi \) is a propositional formula. Then a committed model \( \mathcal{M}' = Update(\mathcal{M}, AF\phi) \) can be obtained by the following operations\(^2\): for each path \( \pi = [s_0, \ldots, s_i, \ldots] \) where \( \forall s_i \in \pi \ (i = 1, \cdots), s_i \not\models \phi \):

1. PU3 is applied to a state \( s_i \in \pi \ (i > 0) \) to substitute \( s_i \) with \( s^* \) such that \( s^* \models \phi \) and \( Diff(s_i, s^*) \) is minimal; or

2. PU2 or PU5 is applied to remove relation \( (s_0, s_1) \) or state \( s_1 = succ(s_0) \) respectively, only if \( RS(\mathcal{M}) - S\{PU3,PU5\} = RS(\mathcal{M}') - S\{PU3\} \).

Theorem 7.2 constrains Theorem 3.4 in Chapter 3. Theorem 7.2 characterizes two typical situations of generating committed models for updates with formula \( AF\phi \). Basically, the theorem says that to obtain a committed model to satisfy \( AF\phi \), we need to perform certain update operations on each path starting from \( s_0 \), where all states from \( s_0 \) do not satisfy \( \phi \). In particular, we can always safely apply PU3 once to do a substitution along the path. On the other hand, we may also apply PU2 or PU5 to remove \( s_1 = succ(s_0) \) if the update still retains all other reachable states in \( \mathcal{M} \).

**Theorem 7.3** Let \( \mathcal{M} = (S, R, L) \) be a Kripke model and \( \mathcal{M} = (M, s_0) \not\models AX\phi \), where \( s_0 \in S \) and \( \phi \) is a propositional formula. Then an admissible updated model \( \mathcal{M}' = Update(\mathcal{M}, AX\phi) \) can be obtained by the following:

1. PU2 is applied to delete some relation(s) \( (s_0, succ(s_0, \neg\phi)) \), PU5 to delete rest some state(s) \( succ(s_0, \neg\phi) \), iff reachable states \( RS(\mathcal{M}) - S\{PU3,PU5\} \) are still

---

\(^2\)Actually, here we mean a sequence of such operations since we need to check each path in which all states after \( s_0 \) do not satisfy \( \phi \).
reachable in the new model $\mathcal{M}'$: \( RS(\mathcal{M}) - S_{\{PU3,PU5\}} = RS(\mathcal{M}') - S'_{\{PU3\}} \), else PU3 is applied to substitute state(s) \( \text{succ}(s_0, \neg \phi) \);

2. if \( \neg \exists \text{succ}(s_0) \) and \( \forall s_i \not\models \phi \), where \( s_i \in S \), then PU4 is applied to next to \( s_0 \) once to add a state \( s^* \models \phi \);

3. if \( \neg \exists \text{succ}(s_0) \) and \( \exists s_i \models \phi \), then PU1 is applied to connect \( (s_0, s_i) \) once.

Theorem 7.3 is based on Theorem 3.2 and is a subset of the latter. The maximal reachable state principle is for constraining the first case in Theorem 7.3: PU5 and PU2 are applied to delete false states succeeding the initial state or delete transitions preceding these false states, if the unchanged reachable states in the original model are still reachable after these updates. Otherwise, PU3 is applied to false states succeeding the initial state to derive committed models. For case two, if there are not successors of the initial state and there are no true states in a model, then, PU4 is applied to add a true state succeeding the initial state. For case three, if there are not successors of the initial state but there are some true states in a model, then, PU1 is applied to connect a transition from the initial state to a true state in the model. These two cases do not involve PU2 and PU5, thus they are not constrained by the reachable state algorithm.

**Theorem 7.4** Let \( M = (S, R, L) \) be a Kripke model, \( \mathcal{M} = (M, s_0) \not\models EG\phi \), where \( s_0 \in S \) and \( \phi \) is a propositional formula. Then an admissible updated model \( \mathcal{M}' = \text{Update}(\mathcal{M}, EG\phi) \) can be obtained by the following: Select a path \( \pi = [s_0, s_1, \cdots] \) from \( M \) which contains minimal number of states not satisfying \( \phi \), then

1. if for all \( s' \in \pi \) such that \( s' \not\models \phi \), there exist \( s_i, s_j \in \pi \) satisfying \( s_i < s' < s_j \) and \( \forall s \leq s_i \) and \( \forall s \geq s_j \), such that \( s \models \phi \), then PU1 is applied to add a relation \( (s_i, s_j) \), or PU4 is applied to add a state \( s^* \models \phi \) and new relations \( (s_i, s^*) \) and \( (s^*, s_j) \);

2. \( \exists s_i \in \pi \), such that \( \forall s \leq s_i, s \models \phi \) and \( \exists s_k \in \pi', \) where \( \pi' = [s_0, \cdots, s_k, \cdots] \), such that \( \forall s \geq s_k, s \models \phi \), then PU1 is applied to connect \( s_i \) and \( s_k \);

3. if \( \exists s_i \in \pi \) (\( i > 1 \)) such that for all \( s' < s_i, s' \models \phi \), \( s_i \not\models \phi \), then,
   a. PU1 is applied to connect \( s_{i-1} \) and \( s' \) to form a new transition \( (s_{i-1}, s') \);
b. if $s_i$ is the only successor of $s_{i-1}$, then PU2 is applied to remove relation $(s_{i-1}, s_i)$, or PU5 is applied to remove state $s_i$ and its associated relations, iff reachable states $RS(M) - S_{\{PU3,PU5\}}$ are still reachable in the new model $M'$: $RS(M) - S_{\{PU3,PU5\}} = RS(M') - S_{\{PU3\}}$, else

4. if $\exists s' \in \pi$, such that $s' \models \phi$, then PU3 is applied to substitute all $s'$ with new state $s^* \models \phi$ and $\text{Diff}(s, s^*)$ to be minimal.

Theorem 7.4 is based on Theorem 3.5 and is a subset of the latter. Theorem 7.4 consists of 4 cases. Only case 3 needs to be constrained by the maximal reachable state principle because it involves PU2 and PU5. This theorem updates one false path to derive a committed model. For case 1, if true states exist on the beginning and end of a path and false states are in between these true states on this path, then, we just need to bypass the part of the path containing the false states by applying PU1 or PU4 to two true states at the beginning and end parts of the path respectively. Then, the updated model containing the extra new path, which avoids the false states on the old path, is a committed model. For case 2, if the beginning part of one path contains all true states and the end part of another path contains all true states, then, we just need to applied PU1 to connect these two parts on different paths. After the connection, a new path containing all true states is formed. For case 3, a. if the first part of a path contains all true states, we just need to connect the last true state on this part to any other true state before the last state on this path to form a new path which contains Strongly connected components (SCCs); b. PU2 or PU5 are applied to cut the transition preceding the first false state or the false state itself on a path, if the updated model can preserve the unchanged reachable states. Otherwise, PU3 is applied to all false states on this path. For case 4, all false states on a false path are updated with PU3 to derive a committed model.

**Theorem 7.5** Let $M = (S, R, L)$ be a Kripke model and $\mathcal{M} = (M, s_0) \not\models A[\phi U \psi]$, where $s_0 \in S$ and $\phi$ and $\psi$ are propositional formulas. Then an admissible updated model $M' = \text{Update}(M, A[\phi U \psi])$ can be obtained by the following: for each false path starting from $s_0$, $\pi = [s_0, \ldots, s_i, \ldots, s_j, \ldots]$: 
1. If $s_0 \not\models \phi$, then, PU3 is applied to $s_0$ to substitute a new state $s'_0 \models \phi$, then,

2. If $\forall s \leq s_i, s \models \phi, s' \not\models \phi \lor \psi, \exists s_j \models \psi \lor \phi$, where $s_i < s' < s_j$, then
   
   if $\text{Diff}(s_{i+1}, s^*) \leq \text{Diff}(s_{i+1}, s'')$, where $s^* \models \psi$ and $s'' \models \phi$,
   
   then, PU3 is applied to substitute a new state $s*$ on $s_{i+1}$.
   
   else,

   PU3 is applied to substitute new states $s'' \models \phi$ on all $s'$;

3. If $\forall s_i \models \phi, \not\exists s_j \models \psi$ and $\exists s_j \models \phi \lor \neg \psi$, then

   PU3 is applied to substitute a new state $s^* \models \psi$ on $s_{i+1}$.

4. If $\mathcal{M} \models E[\phi U \psi]$ and $\mathcal{M} \not\models A[\phi U \psi]$, then

   PU2 is applied to remove transition $(s_0, \text{succ}(s_0))$ or

   PU5 is applied to remove state $\text{succ}(s_0)$,

   iff reachable states $RS(\mathcal{M}) - S_{\{PU3, PU5\}}$ are still reachable in the new model $\mathcal{M}'$: $RS(\mathcal{M}) - S_{\{PU3, PU5\}} = RS(\mathcal{M}') - S'_{\{PU3\}}$.

Theorem 7.5 is from Theorem 3.8 by constraining PU2 and PU5 in Theorem 3.8. The first three cases in Theorem 7.5 are the same as those in Theorem 3.8. The first case is for providing a precondition for case 2 and 3. Case 2 and 3 update the state(s) which does not satisfy either $\phi$ or $\psi$ succeeding a sequence of states from the initial state which satisfy $\phi$. The update in first three cases are PU3 only. However, for case 4, the update cuts off false paths after initial state(s). Case 4 involves PU2 and PU5, thus, it needs to be constrained by using the reachable state principle. If the updates in case 4 cause an unchanged state in the updated model to be unreachable, the updates in case 4 should not be performed.

We can observe that all theorems above are the direct results of their compatible theorems in Chapter 3 constrained by maximal reachable state rules in Definition 7.1. These results are not particularized enough. To illustrate the results of the detailed theorems, we use Theorem 7.2 as an example and its particularized theorem is as follows.
Firstly, we introduce some useful notions. We call $\pi = [s_0, \cdots]$ a false path with respect to formula $\text{AF}\phi$ (here $\phi$ is a propositional formula) if for each state $s \in \pi$, $s \not\models \phi$; otherwise, $\pi$ is called a valid path with respect to $\text{AF}\phi$.

**Theorem 7.6** Let $M = (S, R, L)$ be a Kripke model, and $M = (M, s_0) \not\models \text{AF}\phi$, where $s_0 \in S$ and $\phi$ is a propositional formula. A committed model $M' = \text{Update}(M, \text{AF}\phi)$ can be obtained by the following operations: for each false path $\pi = [s_0, s_1, \cdots]$

1. if there is no other false path $\pi'$ sharing any common state with $\pi$, then PU3 is applied to any state $s_i \in \pi$ ($i > 0$) to substitute $s_i$ with $s^*$ such that $s^* \models \phi$ and $\text{Diff}(s, s^*)$ is minimal; otherwise, this operation is only applied to a shared state $s_j$ ($j > 0$) in maximum number of false paths;

2. PU2 is applied to remove relation $(s_0, s_1)$, if $s_1$ also occurs in another valid path $\pi'$, where $\pi' = [s_0, s'_1, \cdots, s'_k, s_1, s'_{k+1}, \cdots]$ and there exists some $s'_i$ ($1 \leq i \leq k$) such that $s'_i \models \phi$.

**Proof.** We first prove Result 1. Consider a false path $\pi = [s_0, \cdots, s_i, s_{i+1}, \cdots]$. Since each state in $\pi$ does not satisfy $\phi$, we need to (minimally) change one state $s$ along this path so that the new state $s^*$ satisfies $\phi$ (i.e., apply PU3 once). If there is no other false path that shares any states with $\pi$, then we can apply PU3 on any state in path $\pi$. In this case, only one reachable state in the original model with respect to this path is changed to satisfy $\phi$. Thus, the resulting model retains a maximal set of unchanged states.

If there are other false paths sharing a common state with $\pi$, for instance, suppose that there is only another false path $\pi' = [s_0, \cdots, s'_{i-1}, s_i, s'_{i+1}, \cdots]$ sharing a common state $s_i$ with $\pi$, applying PU3 to any state rather than $s_0$ and $s_i$ in $\pi$ may not retain a maximal set of unchanged reachable states. For example, if a state in $\pi$ which is not $s_0$ or $s_i$ has already been changed to make $\pi$ valid, then any change to any state in path $\pi'$ to make $\pi'$ valid will not retain a maximal set of unchanged reachable states in the original model. However, if we change $s_i$ which is in both $\pi$ and $\pi'$ to make both paths valid at the same time, then, the change can retain the maximal set of unchanged reachable states with respect to path $\pi$ and $\pi'$ after the update.
In order to remain a maximal set of unchanged reachable states in the original model, we must consider all states in $\pi$ that are also in other false paths. In this case, we only need to apply PU3 operation to one state $s_j$ in $\pi$ that is shared by a maximal number of false paths. In this way, changing $s_j$ to satisfy $\phi$ will also minimally change other false paths to be valid at the same time. Consequently, we retain a maximal set of unchanged reachable states in the original model.

Now we prove Result 2. Let $\pi = [s_0, s_1, s_2, \cdots]$ be a false path. According to the condition, there is a valid path $\pi'$ of the form $\pi' = [s_0, s'_1, \cdots, s'_k, s_1, s'_{k+1}, \cdots]$], where for some $s'_i \in \pi'$ ($1 \leq i \leq k$), $s'_i \models \phi$. Note that the third path, formed from $\pi$ and $\pi'$, $\pi'' = [s_0, s'_1, \cdots, s'_k, s_1, s_2, \cdots]$ is also valid. Applying PU2 on relation $(s_0, s_1)$ will simply eliminate the false path $\pi$ from the model. Under the condition, it is easy to see that this operation does not actually affect the state reachability in the original model because the valid path $\pi''$ will connect $s_1$ and all states in path $\pi$ are still reachable from $s_0$ but through path $\pi''$. This is described in Figure 7.4 as follows.

\[\square\]

Theorems 7.3, 7.4 and 7.5 can be proved under a similar principle as that for Theorem 7.1 and 7.6 (or Theorem 7.2). Therefore, we omit proofs for these theorems.

### 7.4.2 Complexities

Now we consider the computational issue related to the generation of committed models during a CTL model update. First we have the following general results.
Lemma 7.1 Given a CTL Kripke model $M = (S, R, L)$, $M = (M, s_0)$, where $s_0 \in S$, a CTL formula $\phi$, and two admissible results $M' = (M', s_0')$ and $M'' = (M'', s_0'')$ from the update of $M = (M, s_0)$ to satisfy $\phi$, checking whether $\text{RS}(M) \cap \text{RS}(M') \subset \text{RS}(M) \cap \text{RS}(M'')$ can be achieved in polynomial time.

Proof. For a given $M = (S, R, L)$, we can view $M$ as a directed graph $G(M) = (S, R)$, where $S$ is the set of vertices and $R$ represents all edges in the graph. Obviously, the problem of finding all reachable states from $s_0$ in $M$ is the same as that of finding all reachable vertices from vertex $s_0$ in graph $G(M)$, which can be obtained by computing a spanning tree with root $s_0$ in $G(M)$. It is well known that a spanning tree can be computed in polynomial time [69]. Therefore, all sets $\text{RS}(M)$, $\text{RS}(M')$, and $\text{RS}(M'')$ can be obtained in polynomial time. Also, $\text{RS}(M) \cap \text{RS}(M') \subset \text{RS}(M) \cap \text{RS}(M'')$ can be checked in polynomial time. $\square$

Theorem 7.7 Given two CTL Kripke models $M = (S, R, L)$ and $M' = (S', R', L')$, where $s_0 \in S$ and $s_0' \in S'$, and a CTL formula $\phi$. Deciding whether $(M', s_0')$ is a committed result from the update of $(M, s_0)$ to satisfy $\phi$ is co-NP-complete.

Proof. Since every committed result is also an admissible one, from Theorem 7.7, the hardness holds. For the membership, we need to check (1) whether $(M', s_0')$ is admissible; and, (2) a resulting model $M''$ does not exist such that $(M'', s_0'') \models \phi$ and $\text{RS}(M) \cap \text{RS}(M') \subset \text{RS}(M) \cap \text{RS}(M'')$. From Theorem 3.9, checking whether $(M', s_0')$ is in co-NP. For (2), we consider its complement: a resulting model $(M'', s_0)$ exits such that $(M'', s_0'') \models \phi$ and $\text{RS}(M) \cap \text{RS}(M') \subset \text{RS}(M) \cap \text{RS}(M'')$. From Lemma 7.1, we can conclude that the problem is in NP. Consequently, the original problem of checking (2) is in co-NP. $\square$

In the following, we present some interesting update cases where committed update results may or may not be efficiently computed.

Theorem 7.8 Let $M = (S, R, L)$ be a CTL Kripke model and $\phi$ a CTL formula. The following results hold.

1. If a committed result $\text{Update}((M, s_0), \phi)$ $(s_0 \in S)$ can be obtained by only applying PU1, PU2, PU4 and PU5, then $\text{Update}((M, s_0), \phi)$ can be computed in polynomial time;
2. If $\phi$ is of the form $AG\psi$ and there are two states $s, s' \in S$ such that $s \not\models \psi$, $s' \models \psi$, and any path from $s_0$ to $s'$ contains $s$, then a committed result can only be obtained by applying $PU3$. In this case, deciding whether an update result $Update((M, s_0), \phi)$ is committed is co-NP-complete.

The contents of this chapter are partially contained in [25].
Chapter 8

Conclusions and Future Research

8.1 CTL Model Update

Model checking is an automatic verification tool for system diagnosis. The theory and application of model checking has been extensively studied. However, model checking cannot be used for system modification. The research in this dissertation has filled this gap and extended the model checking function from automatic diagnosis to automatic repair. The theory of CTL model update has been proposed and then implemented. This represents a significant advancement in model checking research. The prototype of the model updater can handle both software and hardware models with reasonable complexity. Our research is a launch point for a future universal model update compiler, which will be based on our existing theoretical results and extend our implementation functions. More detailed summary of our research is as follows.

8.1.1 Minimal Change

CTL model update should be constrained by minimal change rules. Thus, minimal change rules are the major criteria for model update. We have extracted atomic update operations PU1-PU5 from Kripke models as the foundation of CTL model update. Then we have proposed closeness ordering and admissible update as minimal change rules. The closeness ordering determines the comparison methods according to sets of states or relations in different updated models to rule out non minimally changed models. The updated models which are selected according to minimal change rules are called admissible models.
8.1.2 Characterizations and Complexities

Specification properties are abstracted and expressed in the form of CTL semantics. Models to be checked should satisfy these specification properties. Thus, formalizing update with minimal change for each CTL semantic is essential, providing great convenience for updating models satisfying CTL semantics. Updates with minimal change rules for CTL semantics are characterized and proved. The complexity of admissible model determination is proved as \( \text{co-NP-complete} \). If an admissible update is produced by primitive updates PU1, PU2, PU4 and PU5 only without path selection, then, this result can be computed in polynomial time.

8.1.3 Algorithms

The designed CTL model update algorithms based on characterizations guide the implementation of theCTL model updater. These algorithms are designed in pseudo code with recursive calls in similar manner to the labelling algorithms implemented in the SAT function in the SMV model updater. Our algorithms contain a main function and subfunctions according to the CTL syntax in Definition 2.2 and semantics in Definition 2.3. The main function calls sub functions or itself; the subfunctions call the main function or themselves if further updates are needed. Updates are eventually finalized by performing atomic updates PU1, PU2, PU3, PU4 or PU5. The algorithms terminate if an updated model satisfies the required specification properties.

8.1.4 The Prototype and Case studies

Based on the previous theoretical results, a prototype of the CTL model updater is implemented. This prototype is a stand alone model update utility. It consists of model checking and update functions. In addition, it also contains parsing functions. A group of reasonable size library functions which could be used by any function provides great convenience for the model updater package. This prototype has been successfully implemented on three well known models: the microwave oven and Andrew File System models. The implementation proves our two major aims for
our model update research: the prototype is an automatic verification tool and is for error repair.

8.1.5 Study of the Model Explosion Problem

Another interesting theoretical result in addition to minimal change is the study of the model explosion problem. From our initial minimal change rules, too many updated models are produced in some circumstances. This problem was identified during the implementation process. Eliminating unnecessary updated models can solve this problem. A proper solution is achieved by employing the reachable state algorithm. The final updated model is called the committed model. This solution is needed for updates of the AFS0 and AFS1 models. The number of admissible models for the AFS0 model is reduced from 9 to 5; the number of admissible models for the AFS1 model is reduced from 729 to 225. Further characterizations constrained by the reachable state algorithm and the related complexity analysis are presented.

8.2 LTL Model Update

LTL is another temporal logic approach which expresses individual computation paths rather than branch computation paths as is the case for CTL. We explored LTL model update first. The research was published in the conference ISMIS 2005 [30]. However, we restricted our investigation of LTL model update because CTL model update research was our major focus. We have included LTL model update in appendix A as a supplement of our CTL model update research. We will extend this research by improving the minimal change rules and using a real time model in the future. For now, we present the summary of LTL model update as follows.

A similar pattern to CTL model update is followed for the theoretical research for LTL model updates. LTL model update with minimal change and characterizations are explored first. The atomic operations for LTL model update are identified as state addition, state deletion and state substitution. Then the minimal change rules according to the set of relations in updated models are proposed. Based on this, the minimally changed models are selected. According to LTL syntax and semantics,
some characterizations are deduced to update models which are required to satisfy these particular semantics.

8.3 Limitations and Future Research

8.3.1 Theoretical Research

**Limited comparison methods for minimal change** According to our current minimal change rules, the updated models which can be compared are those which are updated with the same primitive operations and all changed states or transitions should be subsets of each other. Otherwise, the updated models cannot be compared. For example, if the model $M$ in Figure 2.7 is updated to two models $M_1'$ and $M_2'$ where the first update implies $\text{Diff}(M, M_1') = (\{(s_0, s_2)\}, \emptyset, \emptyset, \emptyset)$ and the second update implies $\text{Diff}(M, M_2') = (\emptyset, \emptyset, \emptyset, \emptyset, \{s_3\})$, then, these two updated models cannot be compared using our current minimal change rules and both are equally minimally changed models. From both $\text{Diff}$, we can observe that the first updated model $M_1'$ is updated by adding a transition $(s_0, s_2)$ on the original model $M$; the second updated model $M_2'$ is updated by removing state $s_3$ from $M$. If we have a third updated model $M_3$ which implies $\text{Diff}(M, M_3) = (\{(s_1, s_0)\}, \emptyset, \emptyset, \emptyset)$, this model is updated by adding a transition $(s_1, s_0)$ into the original model $M$. $M_1'$ and $M_3$ are updated with the same primitive operation $\text{PU}_1$, however, they are not compatible because the changed transitions $(s_0, s_2)$ in $M_1'$ and $(s_1, s_0)$ in $M_3$ are not subsets of each other. Thus, the current minimal change rules cannot compare all updated models if they are not updated with common primitive operations and if their changed components are not subsets of each other. This is one major reason contributing to the updated model explosion problem. If a significant number updated models cannot be compared with minimal change rules, then, all these models become minimally changed models. To solve this problem, we should provide a wider range of comparison rules for updated models. One approach we can consider is to quantify incomparable primitive operations and changed components in updated models which do not have common features, then compare these quantified components to yield a narrow comparison result. If this idea really works,
the updated model explosion problem could be significantly resolved.

**Decomposing updated models** If we need to compare a set of updated models from an original model, we must to decompose the updated models according to different primitive updates. Then, we can identify the more minimally changed models. For example, in Figure 2.7, if we are given $M_1$ and $M_2$ and we do not know which primitive operations are applied to the original model $M$ to derive these two models, we need to analyze $M_1$ and $M_2$ to discover the ownership of the changed components in these updated models. Then we classify the changes according to primitive operations. In $M_1$ of Figure 2.7, the transition $(s_0, s_2)$ is an added component which is derived from the application of PU1 to the original model $M$. In $M_2$, the transitions $(s_0, s_2)$ and $(s_1, s_0)$ are added components by applying PU1 to $M$; $s_3$ and its associated transitions are removed by applying PU5 to $M$. It is obvious, $M_1$ is a more minimally changed model than $M_2$. Because these updated models are very simple, it is easy for us to identify the changes. However, a real system model most likely will consist of a large number of states and transitions. Identifying changes among large models needs standardized methods. Decomposition of an updated model would need skills such as pattern recognition.

**The model explosion problem** We have used the reachable state algorithm to study the model explosion problem. However, the solution is not perfect. There are still many updated models which cannot be ruled out from the committed models. The theoretical outcome for the reachable state algorithm is not fully analyzed. Two areas of investigation need further work. One is to solve the comparison methods for minimal change which has been described earlier in this section. The other is to improve the reachable state algorithm. We are considering an improvement of the reachable state algorithm as part of our future work. If two states are preserved in an update and there was a path between them in the original model, then there is still a path between them in the updated model. For instance, in Figure 7.3, there is a path from state 21 to 26. Because these two states are preserved, there must be a path between them in all committed models. This would reduce the
number of admissible models even more and would rule out the model in Figure 7.3. This improved reachable state algorithm in fact provides the reachability condition from all states in a model rather than from initial states only. The reachable state algorithm could be further analyzed with graph theory.

8.3.2 A Proposed SMV Model Updater

In addition to solving our previous theoretical research limitations, we have limitations in our model updater prototype as well. This prototype is dedicated to three models. Thus, it cannot be used for updating other models without further modification. It does not use SMV as the input specification language and is not standardized. The most complex model for implementation is AFS1. Thus, this prototype is not tested with models more complex than AFS1. The state explosion problem is not considered in this prototype either and this might be a major barrier in a formal model updater.

Based on the limitations of our prototype, we have identified three directions for our future research to implement a formal SMV model updater. One direction is the integration of the SMV model checker and our model updater to form a universal compiler. Another direction is to investigate abstraction for model update. The third direction is to explore partial model update. The latter two directions are for solving a potential state explosion problem in the future model updater. We specify each of them in more details as follows.

The integration of the SMV model checker and our model updater consists of designing and implementing a formal compiler similar to the existing SMV model checker, but also incorporating the automatic error repairing function. The input will be the SMV specification language rather than the source code defined models as in our model updater prototype. The output will be a text file similar to our prototype and the original SMV compiler to represent corrected Kripke models. Thus, the model checking functions and parsers in the SMV model checker are also used for the proposed model updater. For the integration, the first consideration is the interface. One way is to use the information reported by counterexamples. In this case, the advantage is that the interface is clear and the SMV compiler is not decom-
posed involving a lot of work. However, counterexamples only report final results. All intermediate data of the model checking phase should be recovered according to the information from counterexamples. Another approach to the interface is to decompose the original SMV model updater to embed our model update functions into it. This way we do not need to reverse the information reported by counterexamples, but the task to decompose the SMV model updater and find interfaces for our model update function internal of the SMV model updater is a big challenge, which involves exploring the relationship of the counterexample transition process and the SMV model checking process. The integration relies on adopting one of these approaches.

Once the model updater is implemented as an SMV model updater, state explosions will occur as they do in the SMV model checker. So, it is wise to anticipate the problem before the SMV model updater is implemented. One way to solve this problem is to refine abstraction for model update to do system level “approximate” reasoning rather than “exact” reasoning. To limit a potential state explosion, a complex system should be abbreviated to a simple Kripke model not containing many state components. For a complex system, it is not easy to abstract into a very simple model. However, the abstracted model can be refined from the lower level where there are more states in a model containing more detailed information of a system to a higher level model with fewer states and containing less detailed information of a system. The existing techniques related to this research could be refinement and other abstraction skills as mentioned in section 1.2.3. Another way to solve the state explosion problem is to process partial model update. In this way, only partial states and transitions of a model are updated at a time, which reduce the number of states at the time the model update is executed. The existing technique related to this research could be the Cadence SMV model checker mentioned in section 1.2.2, which uses compositional model checking techniques.

In summary, the research in this dissertation was completed within 3.5 years, which involved both theoretical research and practical implementation (over 6000 lines of code). The former provides theoretical functions for the latter. The latter
proves the soundness and robustness of the former. The combination of both is a self contained complete system. However, the research is not complete. There are still remaining unsolved problems to be addressed by our future research. The research in this dissertation serves as a foundation for our next stage research to implement a formal SMV model updater. Based on the research in this dissertation, we are confident that our next stage research can complete a model update system.
Appendix A

LTL Model Update

A.1 LTL Model Checking: An Overview

In the context of model checking, a designed system is represented by the CTL or LTL Kripke structure $M$. Then $M$ should satisfy a certain property $\phi$. The function of a model checker is to check whether $M$ really satisfies $\phi$. If it does not, the counterexamples [16] report the errors. The describing languages to perform the model checking are, for example, SMV for LTL and CTL, and Promela for LTL.

Parallel to CTL model checking, LTL model checking is extensively studied. One of the main streams is to perform LTL model checking through SMV. In the early stages, Clarke et al. in [16] translated LTL to CTL model checking with fairness constraints, adding a translator into SMV. In recent years, new model checkers such as NuSMV, Cadence SMV and MCK [38] use similar functions of Clarke’s LTL-to-CTL translation. Another main stream for LTL model checking is to translate LTL into automata, such as the current popular LTL model checker SPIN. For SPIN, Promela is its specification language in contrast to SMV. SPIN supports the design and verification of asynchronous process software systems. It performs on-the-fly model checking. We have investigated that LTL model checking is well developed for research purposes and perceived as a potential industrial technique. However, model update based on LTL systems has not been explored by researchers yet. We address in the following sections detailed ideas and methods for LTL model updating.

151
A.2 LTL Syntax and Semantics

Linear-time temporal logic (LTL) is closely related to CTL in that it has similar expressive mechanisms, such as an Until connective. Unlike CTL, however, its formulas have meanings on individual computation paths, i.e. there are no explicit path quantifiers E and A. In this sense LTL appears to be less expressive than CTL. However, LTL allows one to nest boolean connectives and modalities in a way not permitted by CTL; in that sense it appears more expressive. In fact, it is neither more nor less expressive. The definitions of LTL Kripke model and its syntax and semantics are listed as follows.

Definition A.1 [16] Let $AP$ be a set of atomic propositions. A Kripke model $M$ over $AP$ is a four tuple $M = (S, S_0, R, L)$ where

1. $S$ is a finite set of states.
2. $S_0 \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is a transition relation.
4. $L : S \rightarrow 2^{AP}$ is a function that assigns each state with a set of atomic propositions.

Definition A.2 [50] Linear-time temporal logic (LTL) has the following syntax given in Backus naur form:

$$\phi ::= p | (\neg \phi) | (\phi \land \phi) | (\phi \lor \phi) | (G\phi) | (F\phi) | (X\phi)$$

where $p$ is any propositional atom.

An LTL formula is evaluated on a path, a sequence of states, or a set of paths. Consider the path $\pi \overset{def}{=} [s_0, s_1, s_2, \cdots]$; we write $\pi^i$ for the suffix starting at $s_i$, i.e. $\pi^i$ is $[s_i, s_{i+1}, \cdots]$, and $\pi^{[i]}$ for the prefix ending at $s_i$, i.e. $\pi^{[i]} = [s_0, \cdots, s_{i-1}, s_i]$. We write $(s_i, s_{i+1}) \subseteq \pi$ and $s_i \in \pi$, if $\pi = [s_0, \cdots, s_i, s_{i+1}, \cdots]$.

Definition A.3 [50] Let $M = (S, S_0, R, L)$ be a Kripke model. Then a path of $M$ is $\pi = [s_0, s_1, s_2, \cdots]$, where for all $i = 0, 1, \cdots$, we have $s_0 \in S_0$, and $(s_i, s_{i+1}) \in R$. We define when $\pi$ satisfies an LTL formula via the satisfaction relation $\models$ for LTL formulas as follows:
A.3 The LTL Kripke Model Update: Formal Definitions

Model update is a new concept for system verification and modification. If a designed system \( M \) does not satisfy a property \( \phi \) after model checking (See Figure A.1.), the model updater automatically corrects errors in \( M \), updating \( M \) to the new model \( M' \). Thus, \( M' \models \phi \). Model update is implemented by the model updater. Now
Figure A.1: The relationship of a model checker and a model updater.

we define three update operations for LTL model update on the basis of state of an LTL Kripke model, which is a Kripke model with syntax in Definition A.2 and semantics in Definition A.3.

Definition A.4 (Simple Modification) Given an LTL Kripke structure, $M = (S, S_0, R, L)$ over $AP$, where $S_0 \subseteq S$. A simple modification on $M$ is defined as one of the following three single operations: ¹

- **State Addition**: $\text{add}(M, s_{\text{new}}, s_k, s_{k+1}) = M' = (S', S'_0, R', L')$ over $AP$, where $S' = S \cup \{s_{\text{new}}\}$;

  $S'_0 = \begin{cases} 
  S_0 \cup \{s_{\text{new}}\} - \{s_{k+1}\}, & \text{if } s_{k+1} \in S_0 \text{ and } s_{\text{new}} \in S'_0; \\
  S_0, & \text{otherwise}; 
  \end{cases}$

  $R' = \begin{cases} 
  R \cup \{(s_{\text{new}}, s_{k+1})\}, & \text{if } s_{k+1} \in S_0; \\
  R \cup \{(s_k, s_{\text{new}})\}, & \text{if } (s_k, s_{k+1}) \notin R; \\
  R \cup \{(s_k, s_{\text{new}}), (s_{\text{new}}, s_{k+1})\} - \{(s_k, s_{k+1})\}, & \text{otherwise}; 
  \end{cases}$

  $L' : S' \rightarrow 2^{AP}$, where $\forall s \in S'$, if $s \in S$, then $L'(s) = L(s)$, else $L'(s_{\text{new}}) = \tau(s_{\text{new}})$, where $\tau$ are a set of true variables assigned in $s_{\text{new}}$.

- **State Deletion**: $\text{delete}(M, s_k) = M' = (S', S'_0, R', L')$ over $AP$, where $S' = S - \{s_k\}$;

  $S'_0 = \begin{cases} 
  S_0 - \{s_k\}, & \text{if } s_k \in S_0; \\
  S_0, & \text{otherwise}; 
  \end{cases}$

¹In the following statement, $s_{k-1}$, $s_k$ and $s_{k+1}$ are refered to the $k-1$-th, $k$-th and $k+1$-th states of a path in $M$. If $s_k$ is the first state of a path, $s_{k-1}$ is assumed not to exist on this path. If $s_k$ is the last state of a path, $s_{k+1}$ is assumed not to exist on this path.
§A.3 The LTL Kripke Model Update: Formal Definitions

\[ R' = \begin{cases} 
R - \{(s_k, s_{k+1})\}, & \text{if } s_k \in S_0 \text{ and } (s_k, s_{k+1}) \in R; \\
R - \{(s_{k-1}, s_k)\}, & \text{if } (s_k, s_{k+1}) \notin R \text{ and } (s_{k-1}, s_k) \in R; \\
(R - \{(s_{k-1}, s_k), (s_k, s_{k+1})\}) \cup \{(s_{k-1}, s_k)\}, & \text{otherwise}; 
\end{cases} \]

\[ L' : S' \rightarrow 2^\text{AP}, \quad L' = L \text{ for all } s \in S \cap S' \text{ (note } S' \subset S). \]

- **State Substitution:** \( \text{sub}(M, s_{\text{new}}, s_k) = M' = (S', S_0', R', L) \) over \( \text{AP} \), where \( S' = (S - \{s_k\}) \cup \{s_{\text{new}}\}; \)

\[ S_0' = \begin{cases} 
(S_0 - \{s_k\}) \cup s_{\text{new}}, & \text{if } s_k \in S_0; \\
S_0, & \text{otherwise}; 
\end{cases} \]

\[ R' = \begin{cases} 
(R - \{(s_k, s_{k+1})\}) \cup \{(s_{\text{new}}, s_{k+1})\}, & \text{if } s_k \in S_0 \text{ and } (s_k, s_{k+1}) \in R; \\
(R - \{(s_{k-1}, s_k)\}) \cup \{(s_{k-1}, s_{\text{new}})\}, & \text{if } (s_k, s_{k+1}) \notin R \text{ and } (s_{k-1}, s_k) \in R; \\
(R - \{(s_{k-1}, s_k), (s_k, s_{k+1})\}) \cup \{(s_{k-1}, s_{\text{new}}), (s_{\text{new}}, s_{k+1})\}, & \text{otherwise}; 
\end{cases} \]

\[ L'(s_{\text{new}}) = \tau(s_{\text{new}}), \text{ where } \tau \text{ are a set of true variables assigned in } s_{\text{new}}. \]

A simple modification \( \delta \) denotes one of the above three single operations. \( \delta(M) \) denotes the resulting LTL Kripke model after the simple modification \( \delta \) is applied on \( M \). For Definition A.4, we illustrate “State addition” and “State deletion” operations in detail. “State substitution” is actually the combination of the former two operations and for brevity its illustration is omitted.

In Figure A.2, the original LTL model is \( M = (S, S_0, R, L) \), where \( S = \{s_0, s_1, s_2, s_3\} \), \( S_0 = \{s_0\}, R = \{(s_0, s_1), (s_1, s_2), (s_2, s_3)\} \) and \( L \) assigns all states \( s_0, s_1, s_2 \) and \( s_3 \) in \( M \) with \( \{a\}, \{b\}, \{c\}, \text{ and } \{d\} \) respectively. We suppose that there are three different simple modifications under the addition operation \( \text{add}(M, s_{\text{new}}, s_k, s_{k+1}) \) as (A), (B) and (C) in this figure.

In (A), \( s_k = s_1 \) and \( s_{k+1} = s_2 \); the modified LTL model is \( M' = (S', S_0', R', L') \), where \( S' = \{s_0, s_1, s_2, s_3, s_{\text{new}}\}, S_0' = \{s_0\}, R' = \{(s_0, s_1), (s_1, s_{\text{new}}), (s_{\text{new}}, s_2), (s_2, s_3)\} \) and \( L' \) assigns all states \( s_0, s_1, s_2, s_3 \) and \( s_{\text{new}} \) in \( M' \) with \( \{a\}, \{b\}, \{c\}, \text{ and } \{d\} \) respectively.

In (B), \( s_k = \emptyset \) and \( s_{k+1} = s_0 \); the modified LTL model is \( M' = (S', S_0', R', L') \), where \( S' = \{s_0, s_1, s_2, s_3, s_{\text{new}}\}, S_0' = \{s_{\text{new}}\}, R' = \{(s_{\text{new}}, s_0), (s_0, s_1), (s_1, s_2), (s_2, s_3)\} \) and
The modified LTL model in (A) is 
\[ M, s_0 \] 

The modified LTL model in (B) is 
\[ M', (s_0, s_1, s_2, s_3) \]

The modified LTL model in (C) is 
\[ M', (s_0, s_1, s_2, s_3, s_{new}) \]

\[ L' \] assigns all states \( s_0, s_1, s_2, s_3 \) and \( s_{new} \) in \( M' \) with \{a\}, \{b\}, \{c\}, \{d\} and \{new\} respectively.

In (C), \( s_k = s_3 \) and \( s_{k+1} = \emptyset \); the modified LTL model is \( M' = (S', S'_0, R', S') \), where 
\[ S' = \{s_0, s_1, s_2, s_3, s_{new}\}, \quad S'_0 = \{s_0\}, \quad R' = \{(s_0, s_1), (s_1, s_2), (s_2, s_3), (s_3, s_{new})\} \] and \( L' \) assigns all states \( s_0, s_1, s_2, s_3 \) and \( s_{new} \) in \( M' \) with \{a\}, \{b\}, \{c\}, \{d\} and \{new\} respectively.

In Figure A.3, the deletion operation \( delete(M, s_k) \) works on the same LTL model as above. (A), (B) and (C) in this figure are three different simple modifications from the deletion operation with \( s_k = s_2, s_k = s_0, \) and \( s_k = s_3 \) respectively.

The modified LTL model in (A) is \( M' = (S', S'_0, R', S') \), where 
\[ S' = \{s_0, s_1, s_3\}, \quad S'_0 = \{s_0\}, \quad R' = \{(s_0, s_1), (s_1, s_3)\} \] and \( L' \) assigns all states \( s_0, s_1, s_3 \) of \( M' \) with \{a\}, \{b\} and \{d\} respectively.

The modified LTL model in (B) is \( M' = (S', S'_0, R', S') \), where 
\[ S' = \{s_1, s_2, s_3\}, \quad S'_0 = \{s_1\}, \quad R' = \{(s_1, s_2), (s_2, s_3)\} \] and \( L' \) assigns all states \( s_1, s_2, s_3 \) of \( M' \) with \{b\}, \{c\}, \{d\} respectively.

The modified LTL model in (C) is \( M' = (S', S'_0, R', S') \), where 
\[ S' = \{s_0, s_1, s_2\}, \quad S'_0 = \{s_0\}, \quad R' = \{(s_0, s_1), (s_1, s_2)\} \] and \( L' \) assigns all states \( s_0, s_1, s_2 \) of \( M' \) with \{a\}, \{b\}, and \{c\} respectively.

**Definition A.5 (Update)** Given an LTL Kripke model \( M \) and an LTL formula \( \phi \). An update of \( M \) with \( \phi \), denoted as \( Update(M, \phi) \), is a new LTL Kripke model \( M' \), such that

1. there exists a sequence of simple modifications \( (\delta_0, \delta_1, \delta_2, \cdots, \delta_n) \) for

\[ (M_0, M_1, M_2, \cdots, M_{n-1}, M_n) \] respectively, where \( M_0 = M \). The simple mod-

---

**Figure A.2:** The illustration of the addition operation.

**Figure A.3:** The illustration of the deletion operation.
The Principles of Minimal Change

In order to eliminate possible undesirable update results from the modifications in the previous section, we need to provide justified semantic criterion on LTL model updating. Basically, we prefer to select only those new models that have the least change compared with the original models. In belief revision and update research, various minimal change principles have been developed based on classical propositional logic such as described in [87]. It has been observed that these principles are usually not directly applicable for updating on modal logic based systems such as those described in [2], which works on the S5 modal logic Kripke model. In this section, we propose a new minimal change definition for the LTL Kripke model update, based on both logic structure comparison and state differences of LTL Kripke models.

Given a set $X$, $|X|$ denotes the cardinality of $X$. Given an integer, $N$, which could be either positive or negative. $||N||$ denotes its absolute value. e.g., $||-3|| =$
\[ ||3|| = 3. \] Given any two sets \( X \) and \( Y \), the symmetric difference between \( X \) and \( Y \) is denoted as \( \text{Diff}(X, Y) = (X - Y) \cup (Y - X) \).

**Definition A.6 (Ordering on Relations)**

Given three LTL models \( M = (S, S_0, R, L) \), \( M_1 = (S_1, S_{10}, R_1, L_1) \) and \( M_2 = (S_2, S_{20}, R_2, L_2) \), we say relation \( R_1 \) in \( M_1 \) is as close to relation \( R \) in \( M \) as relation \( R_2 \) in \( M_2 \), denoted as \( R_1 \leq_R R_2 \), if

1. \( \forall (s_i, s_{i+1}) \in (R - R_1), (s_i, s_{i+1}) \in (R - R_2) \), and
2. \( ||(|R| - |R_1|)|| \leq ||(|R| - |R_2|)|| \).

Furthermore, \( R_1 =_R R_2 \) iff \( R_1 \leq_R R_2 \) and \( R_2 \leq_R R_1 \); \( R_1 <_R R_2 \) if \( R_1 \leq_R R_2 \) and \( R_1 \neq_R R_2 \).

In Definition A.6, condition 1 means that any element of \( R \) that is not retained in \( R_1 \) is also not retained in \( R_2 \); condition 2 indicates that the different number of elements between \( R \) and \( R_1 \) is less than or at most the same as those between \( R \) and \( R_2 \).

To illustrate how this definition works, let us consider Figure A.4. It is easy to see that \( R - R_1 = \{(s_2, s_3), (s_3, s_4)\} \) and \( R - R_2 = \{(S_1, S_2), (S_2, S_3), (S_3, S_4)\} \). Thus \( R - R_1 \subset R - R_2 \), and condition 1 in Definition A.6 is satisfied. Also we have \( ||(|R| - |R_1|)|| = ||4 - 3|| = 1 \), and \( ||(|R| - |R_2|)|| = ||4 - 2|| = 2 \). That is \( ||(|R| - |R_1|)|| < ||(|R| - |R_2|)|| \), so condition 2 is also satisfied. Therefore, we have \( R_1 \leq_R R_2 \). In other words, in terms of relations, model \( M_1 \) is closer to \( M \) than model \( M_2 \).

**Definition A.7 (Ordering on Models)**

Giving \( M, M_1, \) and \( M_2 \), we say \( M_1 \) is as close to \( M \) as \( M_2 \), which is denoted as \( M_1 \leq_M M_2 \), if the following conditions hold:

1. \( R_1 \leq_R R_2 \), or
2. if \( R_1 =_R R_2 \), then, for \( \forall (s_i, s_{i+1}) \in R, \exists (s_i', s_{i+1}') \in R_1 \), such that, \( \forall (s''_i, s''_{i+1}) \in R_2, \text{Diff}(s_i, s'_i) \subseteq \text{Diff}(s_i, s''_i) \) implies \( \text{Diff}(s_{i+1}, s'_{i+1}) \subseteq \text{Diff}(s_i, s''_{i+1}) \).

We denote \( M_1 \leq_M M_2 \) iff \( M_1 \leq_M M_2 \) and \( M_2 \not\leq_M M_1 \).
§A.4  The Principles of Minimal Change

Figure A.4: The illustration of Definition A.6.

Boolean positive;
int t = 1, n = 0;
for(\(n! = -2, t! = 3; n = -t + +\))
{if(\(n \geq 0\)) positive := 1;
else positive := 0;}

Figure A.5: The illustration of minimal changed update for example 4 and Definition A.7.

Boolean positive;
int t = 1, n = 0;
for(\(n! = 2, t! = 3; n + + t + +\))
{if(\(n \geq 0\)) positive := 1;
else positive := 0;}

Definition A.8 (Admissible Update) Given an LTL Kripke model \(M\) and an LTL formula \(\phi\), an update of \(M\) with \(\phi\), denoted as Update\((M, \phi)\), is called admissible, if any resulting model \(M'\), obtained from Update\((M, \phi)\), has the following properties:

1. \(M' \models \phi\);
2. There does not exist a resulting model \(M''\) from other update of \(M\) with \(\phi\), such that \(M'' \models \phi\) and \(M'' <_M M'\).

Example 4 Consider the software code illustrated in Figure A.6. We need the software system \(M\) to satisfy the property: \(M \models G(\text{positive})\), which means “\(\text{positive} = True\)” when all states of path \(\pi\) in \(M\) contain “\(n=\text{positive number}\)”. We denote \(\phi = G(\text{positive})\).

Now, we have model \(M = (S, S_0, R, L)\) in Figure A.5 to represent the code in Figure A.6, and \(M \not\models \phi\). We give up most other obvious none minimal changed updated models and pick up two updated models \(M_1 \models \phi\) and \(M_2 \models \phi\) in the figure for comparison. To result in both new models, the operation \(\text{sub}(M, S_{\text{new}}, s_k)\), i.e., the simple modification should be applied twice in each. Now we shall pick out a minimal changed model between \(M_1\) and \(M_2\).
Under Definition A.6, we have $R_1 = R_2$. We need to apply Definition A.7 to
decide the desirable update. Under condition 2 of this definition, $Diff(s_1, s'_1) = \{n_1 = -1, n'_1 = 1\}$ and $Diff(s_1, s''_1) = \{n_1 = -1, n'_1 = 1, t = 2, t = 3\}$ resulting in $Diff(s_1, s'_1) \subset Diff(s_1, s''_1)$. Similarly, it is easy to see $Diff(s_2, s'_2) \subset Diff(s_2, s''_2)$.

Thus, from the above comparison, $M_1 <_M M_2$, we have final admissible minimal
changed model $M_1 |\models \phi$, which leads to the final updated software code in Figure A.7.

\[\Box\]

A.5 Characterization

In this section, we will characterize the update of the LTL Kripke model with $F \phi$
and $\phi \cup \psi$, which are two of the most common and frequently used path formulae
in LTL model checking. For simplicity, we consider those LTL Kripke models that
only contain a single initial state. However, our results can easily be extended to
the general case. The characterizations significantly simplify the underlying update
process for these specific formulas.

Theorem A.1 Given $M = (S, \{s_0,\}, R, L)$ with a path $\pi = [s_0, \cdots, s_k, s_k, s_{k+1}, \cdots]$, where $s_{k+1}$ is not the final state, and $M \not\models F \phi$. \text{Update}(M, F \phi) = M'$, where $M' \models F \phi$ and $M' = (S', \{s_0\}, R', L')$, is admissible if the update generates one of
the following resulting models by applying the simple modifications:

1. Addition operation $\text{add}(M, s_{\text{new}}, s_k, s_{k+1})$. The updated model $M'$ has a path $\pi' = [s_0, \cdots, s_k, s_{\text{new}}, s_{k+1}, \cdots]$, such that,

   - $S' = S \cup \{s_{\text{new}}\}$,
   - $R' = R - \{(s_k, s_{k+1})\} \cup \{(s_k, s_{\text{new}}), (s_{\text{new}}, s_{k+1})\}$, and
   - $L' : S' \to 2^{AP}$, where $\forall s \in S'$, if $s \in S$, then $L'(s) = L(s)$,
   - else, $s = s_{\text{new}}, L'(s_{\text{new}}) := \tau(s_{\text{new}})$, where $\tau$ are a set of true variables assigned in $s_{\text{new}}$;
   - $Diff(S_{k+1}, s_{\text{new}})$ is minimal and $s_{\text{new}} \models \phi$.

2. Substitution operation $\text{sub}(M, s_{\text{new}}, s_k)$. The updated model $M'$ has a path $\pi' = [s_0, \cdots, s_{k-1}, s_{\text{new}}, s_{k+1}, \cdots]$, such that,
- $S' = S - \{s_k\} \cup \{s_{\text{new}}\}$,
  $R' = R - \{(s_{k-1}, s_k), (s_k, s_{k+1})\} \cup \{(s_{k-1}, s_{\text{new}}), (s_{\text{new}}, s_{k+1})\}$, and
  $L': S' \rightarrow 2^{AP}$, where $\forall s \in S'$, if $s \in S$, then $L'(s) = L(s)$,
  else, $s = s_{\text{new}}$,
  $L'(s_{\text{new}}) := \tau(s_{\text{new}})$, where $\tau$ are a set of true variables assigned in $s_{\text{new}}$;

- $\text{Diff}(S_k, s_{\text{new}})$ is minimal and $s_{\text{new}} \models \phi$.

**Proof.** Suppose $M_1 = (S_1, \{s_0\}, R_1, L_1)$. To show $M'$ is a minimal resulting model from the original model $M = (S, \{s_0\}, R, L)$, we need to show that for any other updated model $M''$, that $M' \models F \phi$ and $M'' \not\prec_M M'$.

The updated admissible models in case 1 and 2 are not compatible according to Definitions A.6 and A.7. Thus, updated models from case 1 and 2 are equally minimally changed models.

For case 1, $M'$ is an updated model from $M$ with an addition operation $\text{add}(M, s_{\text{new}}, s_k, s_{k+1})$. It is easy to observe that the type of LTL Kripke models for $M''$ are those updated models obtained by performing a sequence of addition operations; e.g., $\text{add}(M, s_{\text{new}}, s_k, s_{k+1})$, $\text{add}(M, s'_{\text{new}}, s_{\text{new}}, s_{k+1})$, · · · , or addition and substitution operations, e.g., $\text{add}(M, s_{\text{new}}, s_{k-1}, s_k)$, $\text{sub}(M, s'_{\text{new}}, s_{k+1})$, · · · on the original model $M$. From Definitions A.6 and A.7, it is easy to prove that for any $M''$ updated by the sequences, we have $M' \prec_M M''$.

Case 2 has a similar proof principle as that for case 1. We do not repeat here. $\square$

**Theorem A.2** Given $M = (S, \{s_0\}, R, L)$ with a path $\pi = [s_0, \cdots, s_k, \cdots, s_j, \cdots]$, where $M \not\models \phi U \psi$. An update $\text{Update}(M, \phi U \psi) = M'$, where $M' \models \phi U \psi$, is admissible by applying the simple modification:

1. if $s_0 \not\models \phi$, substitution $\text{sub}(M, s'_0, s_0)$ is applied to $s_0$, where $s'_0 \models \phi$ and $\text{Diff}(s'_0, s_0)$ is minimal, such that
   - if $s_0 \models \phi$, then $s'_0 = s_0$,
   - else
     - $S = S - \{s_0\} \cup \{s'_0\}$,
     - $R = R - \{(s_0, s_1)\} \cup \{(s'_0, s_1)\}$ and
     - $L(s'_0) := \tau(s'_0)$, where $\tau$ are a set of true variables assigned in $s'_0$, else
2. if $\forall s_k < s_j$, $s_k \models \phi \lor \neg \phi$, $\exists s_j \models \psi$, $s_i$ is the first state from $s_0$ such that $s_i \not\models \phi$, then,

(a) if $Diff(s_i, s^*_k) \leq Diff(s_i, s'_k)$, where $s^*_k \models \psi$ and $s'_k \models \phi$, then, substitution $sub(M, s^*_k, s_i)$ is applied to $s_i$. The resulting model is $M' = (S', \{s'_0\}, R', L')$ with $\pi' = [s'_0, \ldots, s'_k, \ldots, s_j, \ldots]$ such that,

$S' = S - \{s_i \mid i < j, s_i \in S, \not\in S'\} \cup \{s^*_k \mid s'_k \not\in S, \in S'\}$,

$R' = R - \{(s_{i-1}, s_i) \mid s_i, s_{i+1} \mid i \leq (j - 1), s_{i-1}, s_{i+1} \in S, S'\}$

$\cup \{\pi'(s'_k) \mid s'_k \not\in S, \in S'\}$, and

$L' : S' \to 2^{AP}$, where $\forall s \in S'$, if $s \in S$, then $L'(s) = L(s)$, else, $s = s'_k$.

$L'(s'_k) := \tau(s'_k)$, where $\tau$ are a set of true variables assigned in $s'_k$, else

(b) substitution $sub(M, s'_k, s_k)$ to each false $s_k$ where $s_k \not\models \phi$, and $Diff(s_k, s'_k)$ is minimal. The resulting model is $M' = (S', \{s'_0\}, R', L')$ with $\pi' = [s'_0, \ldots, s'_k, \ldots, s_j, \ldots]$ (Only one false state $s_k$ is proposed on $\pi$ for convenience of description) such that,

$S' = S - \{s_k \mid k < j, s_k \in S, \not\in S'\} \cup \{s'_k \mid s'_k \not\in S, \in S'\}$,

$R' = R - \{(s_{k-1}, s_k) \mid s_k, s_{k+1} \mid k \leq (j - 1), s_{k-1}, s_{k+1} \in S, S'\}$

$\cup \{\pi'(s'_k) \mid s'_k \not\in S, \in S'\}$, and

$L' : S' \to 2^{AP}$, where $\forall s \in S'$, if $s \in S$, then $L'(s) = L(s)$, else, $s = s'_k$.

$L'(s'_k) := \tau(s'_k)$, where $\tau$ are a set of true variables assigned in $s'_k$.

Theorem A.2 can be proved in a similar way to Theorem A.1.
Extracting Kripke Models from NuSMV

Our model updater works with a model directly defined in source code. The model definition consists of two major components. These are the individual states of the model containing the corresponding variable values, and the transfer relationships from each state to any subsequent states. This information can be extracted from an SMV implementation after loading and processing a model.

For this description, the NuSMV compiler is used. This compiler is readily available from the internet (both source and executable versions) which allows modification to enhance this data extraction process. This work was performed in the Unix environment, working with an editor (vi) and employing 'X' source capture techniques on the NuSMV compiler output.

First, NuSMV is started up in interactive mode with the model to be captured. For example...

> NuSMV -int afs0.smv

*** This is NuSMV 2.3.0 (compiled on Sat Aug  6 03:06:32 UTC 2005)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

Next, the model is parsed and organised by NuSMV to identify its states and transitions...

NuSMV > go
The model is rationalised by NuSMV into a set of states dictated by the number of variables, and their possible values. The afs0.smv model contains 4 variables each with 2 values resulting in $2 \times 2 \times 2 \times 2 = 16$ possible states. Some of these variable combinations will never be realised in practice and NuSMV eliminates these by providing access to a subset of "reachable" states. NuSMV has also identified the initial states, and state transitions according to the loaded model definition.

The command to display all possible states is `print_fair_states`. The `-v` option verbosely prints each state with the values of each variable. The output for the afs0.smv model from the standard NuSMV is as follows...

```
NuSMV > print_fair_states -v

system diameter: 4 fair states: 16 ($2^4$) out of 16 ($2^4$)

------- State  1 -------
  Client.out = 0
  Client.belief = valid
  Server.out = 0
  Server.belief = none

------- State  2 -------
  Client.out = 0
  Client.belief = nofile
  Server.out = 0
  Server.belief = none

------- State  3 -------
  Client.out = 0
  Client.belief = valid
  Server.out = val
  Server.belief = none

------- State  4 -------
  Client.out = 0
  Client.belief = nofile
  Server.out = val
```
Server.belief = none

------- State 5 -------
Client.out = 0
Client.belief = valid
Server.out = 0
Server.belief = valid

------- State 6 -------
Client.out = 0
Client.belief = nofile
Server.out = 0
Server.belief = valid

------- State 7 -------
Client.out = 0
Client.belief = valid
Server.out = val
Server.belief = valid

------- State 8 -------
Client.out = 0
Client.belief = nofile
Server.out = val
Server.belief = valid

------- State 9 -------
Client.out = fetch
Client.belief = valid
Server.out = 0
Server.belief = none

------- State 10 -------
Client.out = fetch
Client.belief = nofile
Server.out = 0
Server.belief = none
------- State 11 -------
Client.out = fetch
Client.belief = valid
Server.out = val
Server.belief = none
------- State 12 -------
Client.out = fetch
Client.belief = nofile
Server.out = val
Server.belief = none
------- State 13 -------
Client.out = fetch
Client.belief = valid
Server.out = 0
Server.belief = valid
------- State 14 -------
Client.out = fetch
Client.belief = nofile
Server.out = 0
Server.belief = valid
------- State 15 -------
Client.out = fetch
Client.belief = valid
Server.out = val
Server.belief = valid
------- State 16 -------
Client.out = fetch
Client.belief = nofile
Server.out = val
Server.belief = valid
NuSMV identifies the states in display order. This is an important characteristic of the model for later state to state transition relationships and is adopted as the identifier for individual states for the model incorporated into our model updater. The characteristic unique to each state is the variable values, and for our purposes the standard NuSMV verbose presentation is rather clumsy. A modification was implemented in the NuSMV code to streamline the presentation of the variable values by creating a unique single line characteristic for each state. This aided the identification of individual states using a standard unix text editor for text matching.

The modifications were implemented in the BddEnc.c file in the enc/bdd subdirectory of the NuSMV source tree.

An example of the modified NuSMV output follows...

```
NuSMV > print_fair_states -v

# system diameter: 4 fair states: 16 (2^4) out of 16 (2^4)
------ State 1 ------
0, valid, 0, none
------ State 2 ------
0, nofile, 0, none
------ State 3 ------
0, valid, val, none
------ State 4 ------
0, nofile, val, none
------ State 5 ------
0, valid, 0, valid
------ State 6 ------
0, nofile, 0, valid
------ State 7 ------
0, valid, val, valid
------ State 8 ------
0, nofile, val, valid
```
As previously described, not all possible states represent valid variable value combinations. Thus, NuSMV provides another command to display the subset of states with valid variable values. An example follows...

NuSMV > print_reachable_states -v

system diameter: 4 reachable states: 6 (2^2.58496) out of 16 (2^4)

-------- State 1 ------
0, nofile, val, valid

-------- State 2 ------
fetch, nofile, val, valid

-------- State 3 ------
0, nofile, 0, none

-------- State 4 ------
0, nofile, 0, none
It is interesting to note that NuSMV has identified these states in display order with no reference to the full state set identities. This demonstrates the significance of the variable value display string as a unique characteristic for individual states which can be used for state identification. For example to find the identity of State 1 in the total state set, the variable line "0, nofile, val, valid" can be used as a search string within a text editor applied to the total state set output to find the corresponding full set state identity. For this example, the corresponding full set state identity is State 8.

Having established a means of state identification, attention is now focussed on retrieving the state transition relationships from NuSMV for the loaded model. NuSMV supports commands for establishing a trace from an initial state to subsequent valid later states as a series of simulation steps. First an initial state is chosen from the possible initial states. The NuSMV command for this is ”pick_initial_state”. This command implements a ”−i” option to specify interactive mode, as well as a ”−a” option to specify full display of each possible state. The following example demonstrates the choice of an initial state. It should be noted that for the afs0.smv model, only one possible initial state is valid...

```
NuSMV > pick_initial_state -i -a
```

```
*************** AVAILABLE STATES **************

================= State =================
0) -------------------------
0, nofile, 0, none

```

fetch, nofile, 0, none
------- State 5 -------
0, valid, 0, valid
------- State 6 -------
fetch, valid, 0, valid
------------------------
There's only one available state. Press Return to Proceed.

Chosen state is: 0

Thus, the only possible initial state must be chosen. It should be noted that the chosen state's variable value string is "0, nofile, 0, none". This identifies the initial state as State 2 in the full state set and State 3 in the reachable state set.

The other NuSMV command used for determining state to state transitions is "simulate". This command also implements the "-i" and "-a" options as described above for the "pick_initial_state" command. The "simulate" command requires a number as the last parameter to nominate how many steps or state transitions to perform. For this activity, a step count of 1 is used. The command offers a choice of states for the next state transition. An example follows...

NuSMV > simulate -i -a 1

****** Simulation Starting From State1.1 ******

*************** AVAILABLE STATES ***************

================= State =================
0) -------------------------
  0, nofile, 0, none

================= State =================
1) -------------------------
  fetch, nofile, 0, none

Choose a state from the above (0-1): 0
Chosen state is: 0

It should be noted that the "0" choice is the same state as the initial state. For the above example, the "0" state is chosen. The next example below demonstrates the alternative choice...

NuSMV > simulate -i -a 1

********** Simulation Starting From State 1.2 **********

*************** AVAILABLE STATES *************

=============== State ===============
0) -------------------------
  0, nofile, 0, none

=============== State ===============
1) -------------------------
  fetch, nofile, 0, none

Choose a state from the above (0-1): 1

Chosen state is: 1

In this case the chosen next state has the variable combination "fetch, nofile, 0, none", which is equivalent to State 10 in the full state set and State 4 in the reachable state set.

The process is continued using the same command as follows...

NuSMV > simulate -i -a 1
********** Simulation Starting From State 1.3 **********

*************** AVAILABLE STATES **************

================= State =================
0) -------------------------
0, nofile, val, valid

================= State =================
1) -------------------------
fetch, nofile, val, valid

Choose a state from the above (0-1): 0

Chosen state is: 0

This step by step interrogation is continued to determine all possible state transitions from the initial states. The trace is terminated when either an initial state is revisited or a previously visited state is again arrived at. The most common style of termination is the same current state representing the only possible next choice.

Another command which can assist in the trace assessment is the "show_traces" command. This command has a \(-v\) option to request verbose output. An example follows...

NuSMV > show_traces -v

<!-- ################### Trace number: 1 ################### -->
Trace Description: Simulation Trace

Trace Type: Simulation
For the afs0.smv model, the resulting state transitions using the full state set identities is $2 \rightarrow 2$ or $10, 10 \rightarrow 8$ or $16, 8 \rightarrow 13$ or $5, 16 \rightarrow 13$ or $5, 13 \rightarrow 5,$ and $5 \rightarrow 5,$ which terminates the traces.

The models analysed by this technique are theoretical representations of the Andrew File System as described in chapter 6. The afs0.smv model is relatively simple, with 4 variables each with 2 possible values resulting in 16 potential states, with only 6 of the possible states reachable. The afs1.smv model is more complex having 5 variables with increased possible values resulting in 216 possible states with 26 of them reachable. For this model, the subset of reachable states was used as the basis of state identification. There were 4 initial states and a significantly more complex set of state to state transitions. Fortunately, patterns of similarity began to emerge from the analysis, and ultimately it was realised that two of the initial states launched paths shared by the other two initial states. Also, it became obvious that all traces terminated at one of two possible apparent end states. These observations significantly assisted in finalising the analysis for this model.
ReadMe for the CTL Model Updater Prototype

C.1 Files

The software package of the CTL model updater prototype contains files for library functions, model checking, model update, parsing and input models. The zip tar files are available in the attached CD disk of this dissertation or at the following URL:

www.scm.uws.edu.au/~yding/CTLMU/CTLMUprototype1.tar.gz

where the screen printout only contains the original model and the updated model, or

www.scm.uws.edu.au/~yding/CTLMU/CTLMUprototype2.tar.gz

where the screen printout contains the original model, the intermediate data and the updated model.

The gz tar file should be decomposed in Linux or Unix with the following commands:

>gunzip *.tar.gz
>tar -xvf *.tar

After this package has been decomposed, we can see more detailed files as follows. “makefile” which is for directing the build in Unix/Linux is not explained in the following.
The main file

The main file is main.c which is for calling the major functions of model checking, model update, printout functions to print models. This file also performs switching the package to different models and calling test functions.

Library Functions

Files for library functions are states.c and its header file states.h. All structures related to definitions for models, paths and states are contained in states.h. states.c consists of functions for initialization such as setting states and links (transitions), operations for changing a model such as adding a state, adding or removing incoming and outgoing links of a state, changing values of variables in a state, printout functions for models and some model checking functions.

Parsing Functions

Files for parsing are parser.c and its header file parser.h, and String_Parsers.c and its header file String_Parsers.h. The atom and calc_pair structures are contained in parser.h. parser.c consists of parsing and normalizing functions. They are library functions for parsing. Printout functions for structures of parsed strings are also contained. String_Parsers.c is for parsing and normalizing the specification property strings used in the microwave oven, AFS0 and AFS1 models.

Model Checking Functions

Files for model checking are modelchecks.c and its header file modelchecks.h. This file contains functions for model checking for CTL semantics according to CTL semantics such as EG or AG to identify true or false paths or states. This file also identifies values of variables in a state.

Model Update Functions

Files for model update are primitives.c and its header file primitives.h, modelupdates.c and its header file modelupdates.h and CTLModelUpdates.c and its header
file CTLModelUpdates.h. Recursive calls are used in these files. primitives.c contains functions for performing operations PU1 to PU5. modelupdates.c contains the update for AG and update for the case “$\phi$ is $p$” or “$\phi$ is $\neg p$”. CTLModelUpdates.c is the combination of parsing and model update for the three models: the microwave oven, AFS0 and AFS1. The reachable state algorithm is implemented in this file. Interfaces have been marked for future usage to perform other model update functions.

Model Definition Files

Files for defining three models are model.c for the microwave oven model, afs0.c for the AFS0 model and afs1.c for the AFS1 model. Library functions are used in these files. In these files, states are initialized by adding new states to an empty model; adding variables in states; setting values for variables and setting incoming and outgoing links for each state. The model definition file calls library functions to finish their task.

C.2 Building and Running the Prototype

The first command:

>make

Next, running the prototype to see the updated models by text printout on the screen by using the following commands:

>./model

for the microwave oven model,

>./afs0

for the AFS0 model, and

>./model

for the AFS1 model.

Some Unix user profiles may not need “./” in front of the model name. The screen text printout contains the following information:

a. the original model including states, transitions and paths;
b. structures of parsed specification property strings;
c. intermediate data from model checking and update. For example, true or false states, paths display;
d. intermediate data from the reachable state algorithm;
e. updated model printout including states, transitions and paths.

The intermediate data printout can be switched on or off by commenting the related code in some files.

a. switch on or off the printout of the parsed structure for the specification properties: uncomment or comment test_parser() in main.c;
b. switch on or off the printout of paths: uncomment or comment print_paths() in the function print_state_defn() in the file states.c;
c. switch on or off the printout for parsing and normalizing results: uncomment or comment “#define DEBUG 1” in parser.c, and “#define PARSER 3” in String_Parsers.c;
d. switch on or off the printout for intermediate data from model checking and update results: uncomment or comment “#define MODELCHECK 3” in modelchecks.c, and “#define MUDEBUG 2” in file CTLModelUpdates.c.

To print the screen display step by step, the following command can be used:

```
>./afs1 | more
```

To remove the build products from the directory, the makefile has a clean command.
Appendix D

Publications related to this Dissertation

   The above paper contains the major contents of this dissertation and was extracted from some conference publications.

   The above paper contains the major contents of Chapter 7 and some contents of Chapter 6 of this dissertation.

   The above paper contains some contents of Chapter 6 of this dissertation.

   The above paper contains the major contents of Chapter 3 and some contents of Chapter 6 of this dissertation.


179
The above paper contains the major contents of Chapter 5 and some contents of Chapter 6 of this dissertation.

The above paper contains the major contents of Chapter 2 and some contents of Chapter 4 of this dissertation.

The above paper contains the update part of chapter 4 of this dissertation but does not include minimal change rules in the algorithms. It is a foundation for algorithms containing update with minimal change rules in Chapter 4.

The above paper contains the major contents of Appendix A of this dissertation.
Bibliography


International Conference on Computer Aided Verification (CAV03), pp. 206-209. Springer-Verlag, LNCS 2725.


http://www.cs.cmu.edu/~modelcheck/smv.html


[67] NuSMV. http://nusmv.irst.itc.it/bin/download_src2-v2.cgi


