2013

Dynamic abstraction model checking

Wanwu Wang
Iowa State University

Dynamic abstraction model checking

by

Wanwu Wang

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:

Ting Zhang, Major Professor

Wensheng Zhang

Lu Ruan

Iowa State University

Ames, Iowa

2013

DEDICATION

My deepest gratitude goes first and foremost to father, Dongming Wang, and mother, Meizhen He, for their constant encouragement and loving considerations.

Last, my thanks go to my beloved husband, Ying Lai, for his great confidence in me. Also I thank him for his support and love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I take this opportunity to express my thanks to those who helped me with various aspects of conducting research and writing of this thesis.

First and foremost, Thank Dr. Zhang for his guidance, patience, and support throughout this research and writing of this thesis. His insights and words of encouragement often inspired me and renewed my hopes for completing my graduate education.

I thank my committee members for their efforts and contributions to this work: Dr. Wensheng Zhang and Dr. Lu Ruan.

Additionally I thank my labmates, Chuan Jiang and Xiang Huang, who discussed with me a lot of my research.

ABSTRACT

Model checking has attracted considerable attention since this technique is an automatic technique for verifying finite state concurrent systems. It is a formal method to verify if a software system or hardware system meets its properties. Nowadays, model checkers have become indispensable tools in hardware and software design and implementation, since they can reduce human efforts. Time and feasibility of the model checking process depends up on the size and complexity of the formal system model. However, the state space explosion problem still remains a major hurdle, as the number of global states can be enormous. There are many methods to improve the speed of model checkers and abstraction technology is one of them. Abstraction amounts to removing or simplifying details, as well as removing entire components of the concrete model irrelevant to the specifications. In practice, abstraction-based methods have been essential to verify designs of different fields of industrial complexity. Manual abstraction is ad hoc and error-prone; hence, automatic abstraction strategies are desirable for verifying actual hardware and software design.

This thesis presents a new approach to check the model using two abstractions—Universal Abstraction and Existential Abstraction. These new techniques can check both the Existential fragment of Computational Tree Logic (ECTL) and the Universal fragment of Computational Tree Logic (ACTL) specifications. I developed a Model Checker, called LOTUS, building upon these new techniques with a traditional fixed point algorithm on Linux. Experimental results confirmed the feasibility and validity of this new dynamic model checking technique. The input grammar is designed and implemented using Bison and YACC on Linux. The process for this new technique follows. First, automatically construct two abstraction models according to the specifications defined in the input file by the user. Second, LOTUS verifies whether the abstraction model satisfies the specifications and outputs the conclusion. Lotus can produce the final output, if the conclusion is credible; otherwise, refine the two abstraction models

according to the counterexamples or witnesses produced by the abstraction model. This process is repeated until the abstraction model is equivalent to the concrete model or the authentic conclusion can be obtained. In this thesis, I aim to provide a complete picture of this dynamic model-checking algorithm, ranging from design details to implementation-related issues and experiments of the Philosopher Dinning Problem.

The main contributions of this approach include three aspects. First, Dynamic Abstraction Algorithm can check both ECTL and ACTL within abstraction methods. Second, a transition abstraction is introduced in this thesis with the purpose is to make the model checker easier to implement. Third, refinement of both abstraction models according to either witness or counterexample is actually modifying the transitions. This method may reduce time and space consumption.

Key Words: Abstraction, Counter-example refinement, Dynamic model checking

CHAPTER 1. OVERVIEW

I provide a brief overview of the mechanism of model checking and how abstraction is used in model checking. Next I describe how model checking is used to verify complex system designs. The overview and background of this research area will be discussed. I also trace the development of abstraction technology applied in model checking.

1.1 Introduction

Model checking is becoming more and more popular as different realistic problems of hardware and software can be solved using modern model checkers. The basic idea of Model Checking is to check whether a given model satisfies certain properties. The basic step in model checking is as follows:

1. Modeling: The first task is to turn the practice problem into a formally defined model. This step is usually completed by the user, but it is difficult to model the problem sufficiently to save space and time. The modeling of a design requires the use of abstraction to eliminate the irrelevant or unimportant details to reduce time consuming.

2. Specifications: The model checking problem is to check whether the model satisfies certain specifications and how to express the specifications is a problem. Nowadays, it is common to use temporal logic to represent the logic. The most popular in model checking is LTL (Linear Temporal Logic) and CTL (Computation Tree Logic).

3. Verification: It is ideal to verify the model automatically but it is difficult to implement this in practice. Usually, it needs manual analysis to determine the verification. If the results are negative, a counterexample could disprove the specifications and help the

designer to modify the model to avoid this counterexample. The positive output is used by the designer directly.

Clark and Emerson used Temporal Logic in the state exploration search problem. Their paper showed non-trivial problems can be solved using this combination according to Clarke (1981). Model checking has many advantages: no proofs, fast, generates counter-examples, etc. The most influential results for model checking is Tarski's Fixpoint Lemma, every monotonic functional on a complete lattice has a fixpoint based on Tarski (1995) and Kleene's First Recursion Theorem from Kleene (1971). The symbolic model checker SMV by McMillan (1993) was introduced based, on ordered binary decision diagrams in Bryant. (1986). After this, more problems can be solved in less time. However, as the real problems always have enormous states, it is difficult to use the Symbolic Model Checker without any state reduction techniques. Based on this idea, many researchers focus on abstraction and other reduction algorithms.

Abstraction uses abstract states domain instead of concrete states domain and checks the properties in the abstract states domain. It probably is the most significant technique for reducing the complexity of model checking. This will generate the output, if the checker can obtain credible results from the abstraction model; otherwise, refine the abstraction model to make the abstraction more precise, according to the counterexample produced by the abstraction model. The reason abstraction can be applied is some models have many transitions and states irrelevant to the specifications among the checking procedure. Recently, some work has already been completed on this topic, such as Bensalem (1992), Clarke (1994), and S.Graf (1997). Refinement of abstraction is also a popular topic in model checking. Some researchers tried to refine the abstraction, based on the counterexample generated by the abstraction model, such as the works from Clark (2001), Clark (2002), and E. M. Clarke and Strichman. (2002).

We note that even though there are many ideas on abstraction, there is still much room for improvement on abstraction, as abstraction could be simpler, more precise, and more intelligent. Much work can be achieved to enlarge the states the model checker can check.

We present a new idea that can check both ECTL and ACTL by the abstraction techniques. This idea also provides a new refinement method (transition refinement) based on the comparison of the counterexample or witness between two abstract models.

1.2 Related work

Beginning with the localization reduction from the work of Kurshan (1994), many researchers start with using the counterexample to refine the abstract models. The localization reduction begun with small subsets of states related with the specifications. All other variables are abstracted with nondeterministic assignments. The model will be refined, if the counterexample is determined spurious. Then, the abstract states will be modified to be more precise. This method either leaves a variable unchanged or replaces it by an assignment. The work by Balarin and Sangiovanni-Vincentelli (1993) proposes similar ideas.Lind-Nielsen and Andersen (1999) develop another technique to refine the model. The main idea from the work is using upper and lower approximations instead of handling CTL.

Many papers (Lee and Somenzi (1996), Pardo (1997), Pardo and Hachtel (1998)) stated techniques for CTL modeling checking using abstraction. Govindaraju and Dill (1998) discussed the idea to identify the first spurious state from the abstract counterexample. In their work, they proposed the method that chooses the concrete state randomly from the counterexample and constructs a concrete counterexample by the transition relation.

The above research focuses on the finite state system. However, infinite state systems are more practical. Abstraction techniques are also applied in this field. S.Graf (1997) stated the method, called predicate abstraction, which abstracts an infinite state system to a finite state system, and then checked the model using the same idea of finite state systems. Then, a number of researchers worked on this field and proposed ideas on model checking on infinite state systems, such as Bensalem and Owre (1998), and Das and Park (1999).

Model checking is used in many fields of industry. Abstraction techniques have been used to verify the industrial hardware systems by Fura (1993), Graf (1994), Ho and Kam (1998). Date independent systems can be verified by model checking by Wolper and Lovinfosse (1989). Verifying pipeline systems has been discussed by Berezin and Zhu (1998), Burch and Dill (1994), Jones (1998a). McMillan introduced a method of data abstraction, assume-guarantee reasoning, and theorem proving techniques using the work by Jones (1998b).

1.3 Background

The background of model checking and abstraction will be introduced in this section. Examples will be provided to help understanding the fundamental knowledge.

1.3.1 Fundamentals of model checking

The problem of model checking can be described as the following from Edmund.M.C (2008).

Let $\mathcal{M}$ be a Kripke structure (i.e., state-transition graph). Let f be a formula of temporal logic (i.e., the specification). Find all states s of $\mathcal{M}$, such that $\mathcal{M}, s \models f$.

Hence, the meaning of model checking is to determine whether the given model satisfies certain specifications. Clarke stated the advantages and disadvantages from the Edmund.M.C (2008). The advantages are:

1. No proofs! The model checking problem only outputs the results, which show whether the model satisfies the specifications. Hence, no proof is needed to determine the conclusion. This makes it easier to design and implement the model checker.

2. Fast. Since the model checker only cares about the final result, it will generate the output as soon as it finishes. The model checker's speed is faster than rigorous methods using the proof method.

3. Diagnostic counterexamples. The model checker can produce a counterexample showing why the model does not satisfy certain specifications. The user can modify the model according to the counterexample to make the model satisfy the specifications.

4. No problem with partial specifications. Model checking does not need to completely specify the program. Then, model checking can be used during the design of a complex system.

5. Temporal Logics can be easily expressed. It is easy to represent the specification by the temporal logic using the model checker, which is easy for users to utilize.

However, prove the model checker is not the same as other proof methods, it also has some disadvantages. for example the user cannot better understand the output, since no proof

is generated. Users need to represent the specifications by themselves, but it is difficult, as different specifications may cause different efficiency, even though the meanings are same. The biggest problem is the state explosion problem because the model checker will checks all states in the worse case scenario and may cause this problem when the number of states is huge.

1.3.2 Modeling systems

A formal description of the system behavior must be defined. In this thesis, *Kripke structure* is used to represent the state transition graph, as a description of the model to be checked. A Kripke structure consists of a set of states, a set of transitions, and a label function. Let AP be a set of atomic propositions. A Kripke structure, M over AP, is a tuple $M = \{S, S_0, R, L\}$ where

1. S is a finite set of states
2. $S_0 \subseteq S$ is the initial states.
3. $R \subseteq S \times S$ is the state transition and must be total.
4. $L : S \to 2^{AP}$ is label function, which labels each state the set of atomic propositions true for that state.

An example with three states of Kripke structure graph can be seen in Figure 1.1. In this example, there are three states labeled by the atom propositions. The transitions among these states are represented by the graph, where an edge between state a to state b means there is a transition from a to b.

1.3.3 Temporal logic

1.3.3.1 CTL

Computation Tree Logic (CTL) is a restricted subset of CTL*. CTL* formulas are composed of path quantifiers and temporal operators. **A** (for all computation paths) and **E** (for some computation path) are two quantifiers. There are five basic operators:

1. **X**: requires a property hold in the next state of the path.

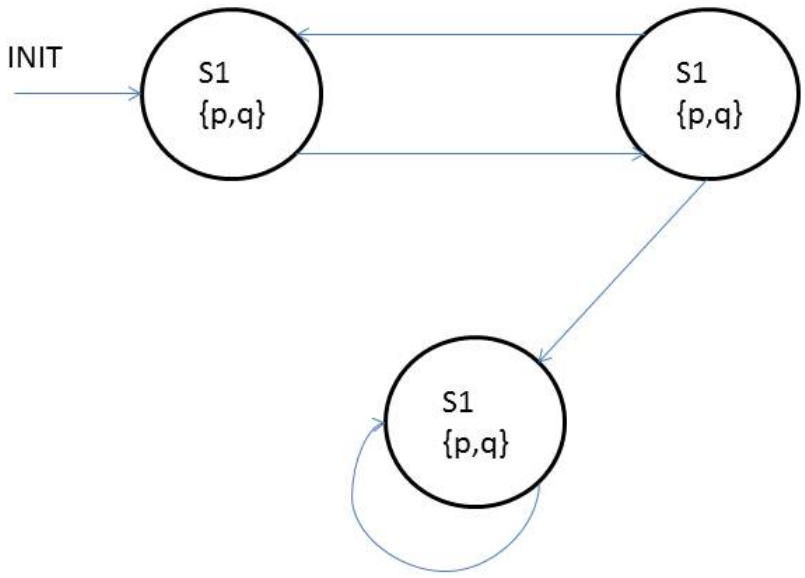


Figure 1.1: Kripke structure graph

2. **F**: the property will hold at some state on the path.

3. **G**: the property holds at every state on the path.

4. **U**: It holds if there is a state on the path where the second property holds, and at every preceding state on the path, the first property holds.

5. **R**: The second property holds along the path up to and including the first state where the first property holds.

There are ten basic CTL operators:

1. AX and EX

2. AF and EF

3. AG and EG

4. AU and EU

5. AR and ER

An example with four pictures to explain the CTL can be seen in Figure 1.2.

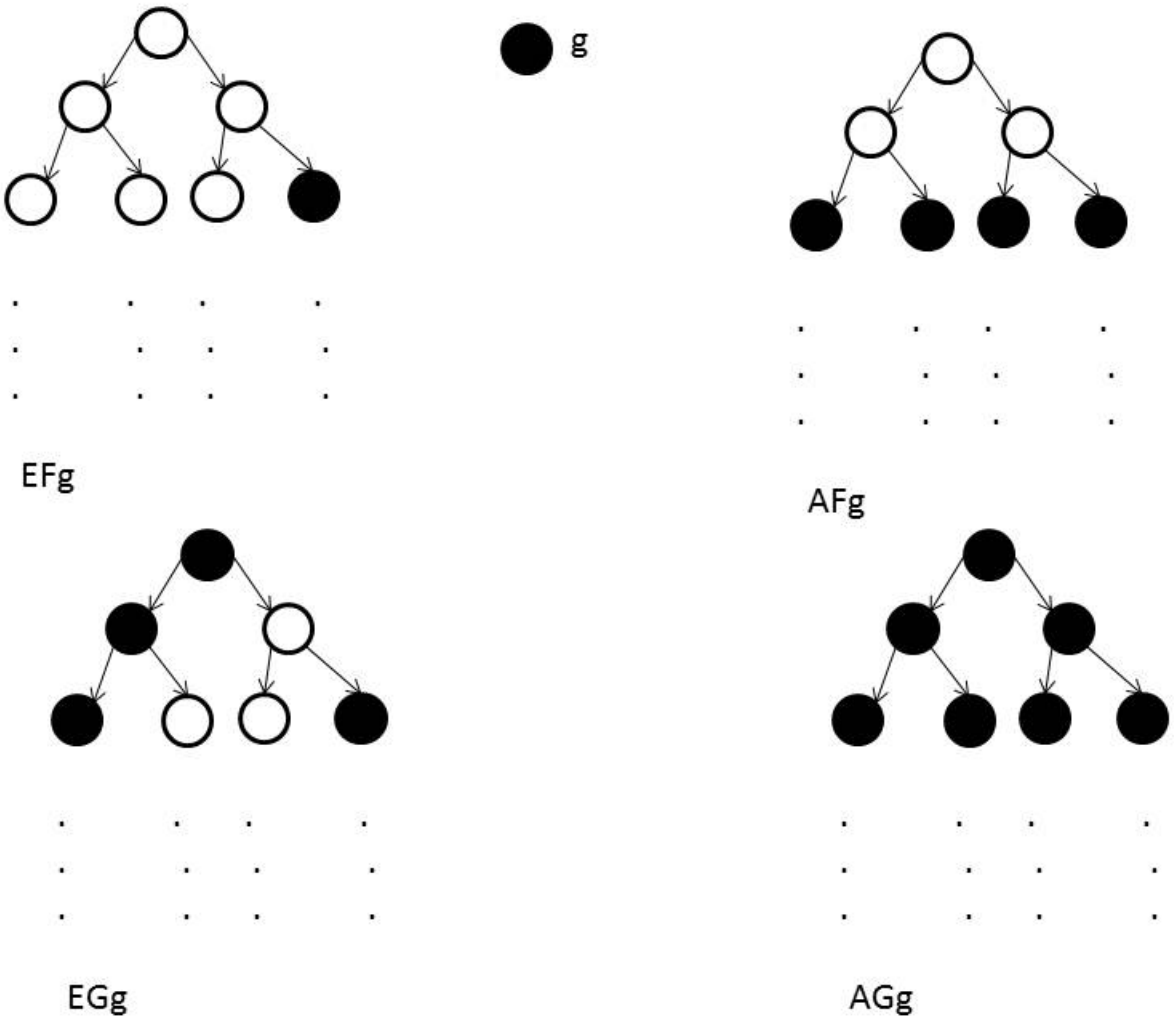


Figure 1.2: Basic CTL operators.

The black nodes represent the states satisfying the corresponding property.

1.3.3.2 LTL

Linear Temporal Logic has the path formula like this:

- If $p \in AP$, then p is a path formula.
- If f and g are path formulas, then $\overline{f}$, $f \vee g$, $f \wedge g$, $\mathbf{X}f$,$\mathbf{F}f$,$\mathbf{G}f$,$f\mathbf{U}g$,$f\mathbf{R}g$.

CTL and LTL are both a subset of CTL*. But CTL and LTL have different expressive powers. For example: $F(G\,p)$ can be expressed by LTL, but cannot defined using CTL. No LTL formula can represent $AG(p \rightarrow (EX\,q \wedge EX\,\overline{q}))$, which can be defined in CTL.

1.3.4 BDD

Ordered binary decision diagrams (BDD) are a popular representation for boolean formulas, based on Randal (1986). Binary decision diagram is a directed acyclic graph. BDD is a rooted tree and each node has two children: $low(v)$ and $high(v)$. Every binary decision diagram with root, v, determines a boolean function $f_v(x_1, ..., x_n)$ in the following way:

- If v is a terminal vertex:
  - If $value(v) = 1$, then $f_v(x_1, ..., x_n) = 1$.
  - If $value(v) = 0$, then $f_v(x_1, ..., x_n) = 0$.
- If v is a nonterminal vertex with $var(v) = x_i$, then f_v is the function $f_v(x_1, ..., x_n) = (\overline{x_i} \wedge f_{low(v)}(x_1, ..., x_n)) \vee (x_i \wedge f_{high(v)}(x_1, ..., x_n))$

BDD is a popular tool to represent the Kripke graph and a number of model checkers use BDD to represent the Kripke structures. Here is an example to illustrate the process to represent states and transitions using BDD. Assume we use two variable (a,b) to represent a state. Then there are four states can be represented :ab,$a\overline{b}$,$\overline{a}b$ and $\overline{a}\overline{b}$. We can represent transition from state aa to state $\overline{a}\overline{b}$ by the conjunction $(a \wedge b \wedge \overline{a} \wedge \overline{b})$.

There are three ways to reduce the BDD:

1. remove the duplicate terminals: Eliminate all but one terminal vertex and redirected the other arcs.

2. remove duplicate nonterminals: If two nonterminals have the same value, same low and same high, then eliminate one of them and redirect the other edges.

3. remove redundant tests: If the nonterminal node has the property, $low(v) = high(v)$, then eliminate this node and redirect all incoming arcs to $low(v)$.

The example in Figure 1.3 compares the OBDD and reduces OBDD after applied in section 3.

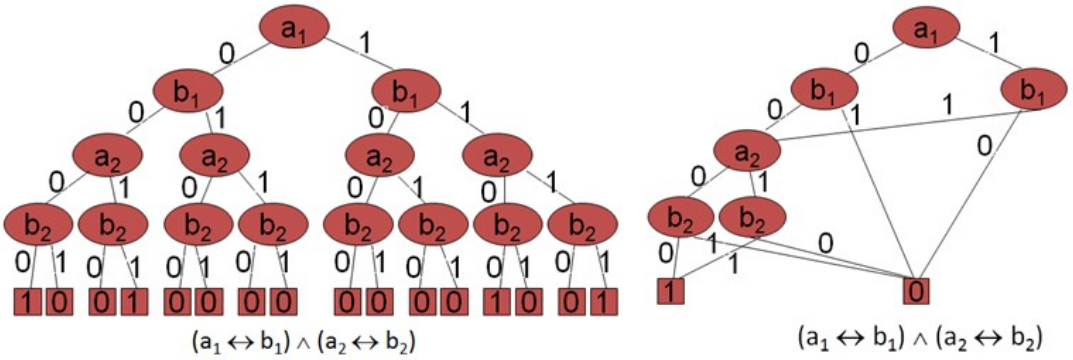


Figure 1.3: OBDD and reduced OBDD

Details will be discussed in the chapter of introduction of Lotus.

1.3.5 Symbolic model checking

The basic algorithm for model checking is fixpoint theory. There are two kinds of fixpoints: least fixpoints and greatest fixpoints. CTL formulas can be represented by these two fixpoints. Each of the basic CTL operators may be stated as a least or greatest fixpoint, according to the following:

1. $AF\, f_1 = \mu Z.f_1 \vee AX\, Z$

2. $EF\, f_1 = \mu Z.f_1 \vee EX\, Z$

3. $AG\, f_1 = \nu Z.f_1 \wedge AX\, Z$

4. $EG\, f_1 = \nu Z.f_1 \wedge EX\, Z$

5. $A[f_1 \, U \, f_2] = \mu Z.f_2 \vee (f_1 \wedge AX \, Z)$

6. $E[f_1 \, U \, f_2] = \mu Z.f_2 \vee (f_1 \wedge EX \, Z)$

7. $A[f_1 \, R \, f_2] = \nu Z.f_2 \wedge (f_1 \vee AX \, Z)$

8. $E[f_1 \, R \, f_2] = \nu Z.f_2 \wedge (f_1 \vee EX \, Z)$

The the fundamentals applied in the model checking field. The symbolic model checking starts with the function $Check$, which returns an BDD representing exactly the state of the system, which satisfies the formula. Hence, the CLT procedures can be represented as:

1. $Check(EX \, f) = CheckEX(Check(f))$

2. $Check(E[f \, U \, g) = CheckEU(Check(f), Check(g))$

3. $Check(EG \, f) = CheckEG(Check(f))$

1.3.6 Abstraction

Since the number of states to verify the problem become larger, the state explosion problem can be the biggest challenge for model checking. Abstraction is a way to solve this by reducing irrelevant states in the concrete model. A number of researchers worked on this field (Adiya and Ashish (2012), Matt and Vinod (2012), Georges and Christoph (2011), Luca and Armando (2010)) trying to abstract the model automatically and verify the model by the abstraction model. There are two main basic abstraction methods. (1) the cone of influence reduction and (2) data abstraction in this field.

The cone of influence reduction intends to obtain the set of states related with the specification checking procedure. To make it more precise, the cone of influence C of V' is the minimal set of variables, such that

- $V' \subseteq C$
- if for some $v_l \in C$, its f_l depends on v_j, then $v_j \in C$.

This abstraction method will not cause a spurious counterexample, since it only abstracts the states which do not have dependencies on the specifications. Let f be a CTL* formula with atomic propositions in C, then $M \models f \Leftrightarrow \widehat{M} \models f$.

The other abstraction method is called data abstraction. The key point is to construction a function mapping concrete state into an abstract state to reduce the complexity of the model checking. In other words, building this abstraction function is the most important part in the abstraction process. This function can be produced automatically or manually. The example by Edmund (1999) shows the abstraction function meaning. We define the abstraction, h, and the mapping is $h(red) = stop$, $h(yellow) = stop$ and $h(green) = go$. The example can be seen in Figure 1.4 With the abstraction function, the states space can be reduced by integrating the

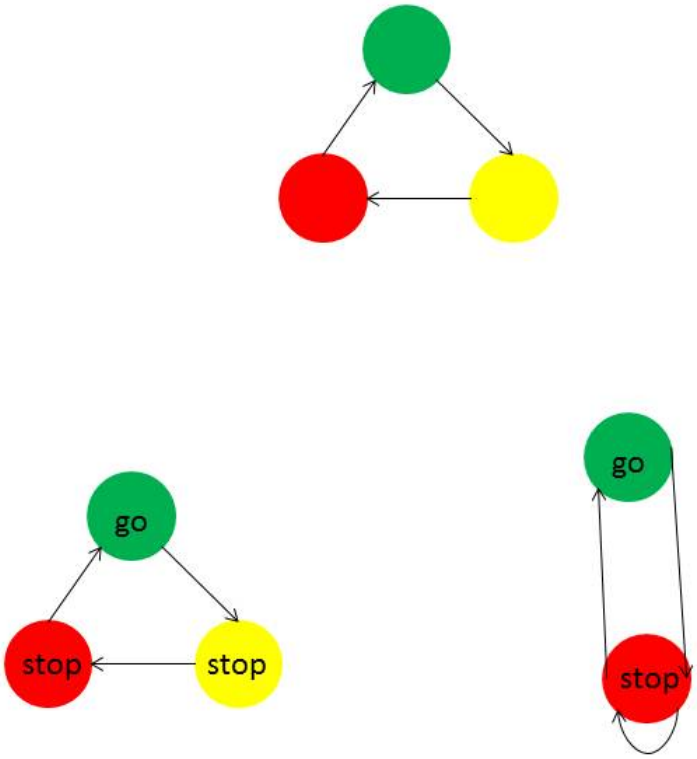


Figure 1.4: Abstract traffic light

concrete states. Hence, the abstraction model is much smaller and simpler than the concrete model. Due to this, abstract model may produce a spurious counterexample, which needs checked before obtaining the final conclusion.

1.3.7 Counter-example guided abstraction refinement

Counter-example guided abstraction refinement first proposed by the work of Clark (2002). Then, other researchers such as Thomas and Sriram (2001), Thomas (2004), Edmund (2003), Edmund (2004) continued in this field. The main idea is using abstraction to reduce the state space and check the abstraction model, instead of using the concrete model. According to the theorem of ACTL, the model checker can output the positive result directly. However, for the negative conclusion, after verifying whether the counterexample is spurious or not, the model will automatically produce the result or refine the model. The refinement is to modify the abstraction, function which means separate certain abstract states, into different abstract states and check again.

To identify the spurious path counterexample is to try to find the concrete counterexample corresponding to the abstract counterexample. $\widehat{T}$ is a counterexample path produced by the abstract model and $h(s) = \widehat{s}$, that is, $h^{-1}(\widehat{s}) = \{s|h(s) = \widehat{s}\}$. These three statements are equivalent:

1. The path $\widehat{T}$ corresponds to a concrete counterexample.

2. The set of concrete paths $h^{-1}(\widehat{T})$ is nonempty.

3. For all $a \leq i \leq n$, $S_i \neq \emptyset$.

If the counterexample corresponds to a concrete counterexample, then the negative result occurs. Otherwise, the model must be refined. Clarke provided two ways to modify the model, based on the type of the abstract counterexample —SplitPath and SplitLoop.

The first is SplitPath. This algorithm starts from the initial path and try to find the corresponding concrete path. If the set is empty, then the counterexample is spurious; otherwise, the model does not satisfy the specification. The states are defined as two types: bad states

and dead-end states. Bad states can cause the counterexample to be spurious and dead-end states have no connection to the previous abstract states, but have a connection to the next abstract state. An example is shown in Figure 1.5

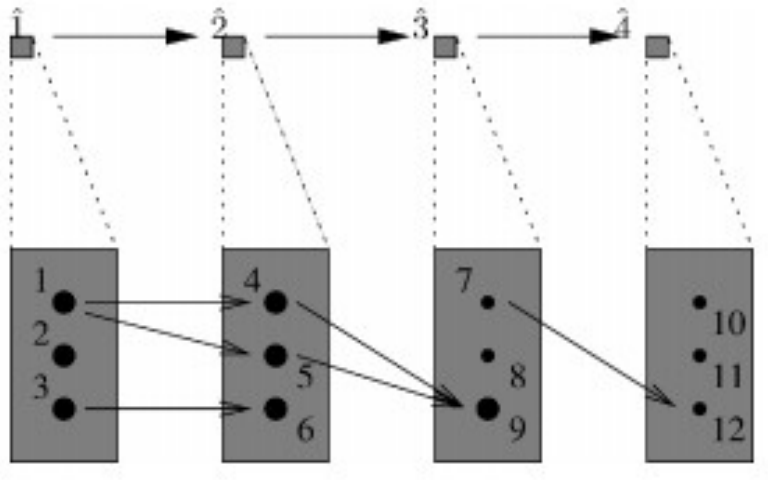


Figure 1.5: Path abstract counterexample

The second is Splitloop. In this case, the minimum length is calculated by the formula, $min = min_{i+1 \leq j \leq h} |h^{-1}(\widehat{s_j})|$. $\widehat{T}_{unwind}$ denotes the path $\langle \widehat{s_1}, ..., \widehat{s_i} \rangle \langle \widehat{s_{i+1}}, ..., \widehat{s_n} \rangle^{min+1}$. After transfering the loop to path, it will be the same procedure as SplitPath. With this method, SplitLoop will call the method Splitpath according to the following:

- $\widehat{T}$ corresponds to a concrete counterexample.
- $h^{-1}(\widehat{T}_{unwind})$ is not empty.

The example is in shown Figure 1.6

Depending on the result of whether the counterexample is spurious or not, the model will be refined, if the counterexample is spurious. The key to refinement is to separate the bad states and dead-end states from the abstract states. Clarke proved the problem of finding the coarsest refinement is NP hard and provided a way to modify the model using the algorithm in Algorithm 1

Hence, there exists a unique refinement algorithm, which can be computed in polynomial time. The algorithm assumes a composite abstraction function as $(d_1, ..., d_m)$. Given a set

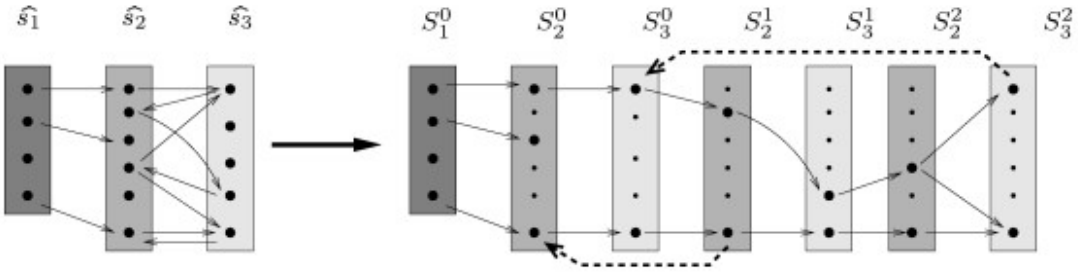


Figure 1.6: Loop abstract counterexample

Algorithm 1 Algorithm PolyRefine

```
for j=1 to n do
    ≡'_j=≡_j
    for every a, b ∈ E_j do
        if proj(S_D, j, a) ≠ proj(S_D, j, b) then
            ≡'_j=≡_j \{(a, b)}
        end if
    end for
end for
```

$X \subseteq X$, an index $j \in [1, m]$, and $a \in D_j$, the projection function is defined as :

$proj(X, j, a) = \{(d_1, ...d_{j-1}, d_{j+1}, ...d_m) | (d_1, ...d_{j-1}, a, d_{j+1}, ...d_m) \in X\}$. After refinement, apply the modified model to check the specifications again, until the checker outputs the credible results.

E. M. Clarke and Strichman. (2002) stated the abstraction method, using ILP and Decision tree. The key is to divide the bad states and dead-end states using ILP or Decision tree. The ILP method, attempts to find the minimum set of solutions, which avoid the spurious counterexample output by the abstraction. The ILP function is shown in (1.1):

$$\begin{aligned} & Min \sum_{i=1}^{|\mathcal{I}|} v_i \\ subject\,to : (\forall s \in S_{D_I})(\forall t \in S_{B_i}) & \sum_{1 \le i \le |\mathcal{I}|, s(v_i) \ne t(v_i)} v_i \ge 1 \end{aligned} \tag{1.1}$$

$v_i = 1$ if and only if v_i is in the separating set. Then, represent each constraint as a pair of states (s_i, t_j), meaning at least one of the variables that separates the two states should be selected.

Abstraction with Decision tree makes use of machine learning ideas to build Decision tree and obtain concrete results. Learning with Decision tree is one of the most popular algorithms in the machine learning area. The Decision tree algorithm inputs a set of examples and generates a Decision tree that classifies them. Data are classified by starting at the root node and testing the attribute specified by the node. After determining the direction, the algorithm moves down the tree branch corresponding to the value of the attribute. They shown the results of using this algorithm, and in some cases it is better than the original.

They format the problem of separating S_{D_I} from S_{B_I} as follows:

1. The attributes correspond to the invisible variables.
2. The classifications are $+1$ and -1 corresponding to S_{D_I} and S_{B_I}.
3. The examples are S_{D_I} labeled $+1$ and S_{B_I} labeled -1.

The algorithm to build the Decision tree $DecTree(Examples, Attributes)$ is as follows:

1. Create a Root node
2. If all examples are classified the same, return Root.
3. Let $A = BestAttribute(Examples, Attributes)$. Label Root with attribute A.
4. For $i \in \{0, 1\}$, let $Examples_i$ be the subset of $Examples_i$ having value i for A.
5. For $i \in \{0, 1\}$, add an i branch to the Root pointing to THE subtree generated by $DecTree(Examples, Attributes - \{A\})$.
6. Return Root.

CHAPTER 2. DYNAMIC MODEL CHECKING

Chapter 1 describes the abstraction methods applied to the ACTL specifications. For abstraction to be used more widely in model checking, I define a new kind of abstraction technique, Dynamic Abstraction Algorithm. In this chapter, I will discuss this new method in detail and provide a complete picture of this Dynamic Abstraction Algorithm, which can be applied in both ACTL and ECTL specifications.

2.1 Algorithm

The main point of this new algorithm is to use existential abstraction to check the ACTL specifications and universal abstraction to check the ECTL specifications. Using refinement to modify the abstraction model, if the counterexample or witness is spurious.

2.1.1 Two abstraction

I provide a brief description of two abstraction techniques in this section: Universal Abstraction and Existential Abstraction. Let AP be an atomic proposition, S is the set of all possible states, S_0 is the set of initial states, R is the transition relation, and L is the label function. Define $M = (AP, S, S_0, R, L)$ be a concrete Kripke structure. Let $\widehat{S}$ be a set of abstract states and $\gamma : \hat{S} \to 2^S$ be a concretization function.

The existential abstraction is defined as a Kripke structure $\hat{M} = (AP, \hat{S}, \hat{S_0}, \hat{R}, \hat{L})$ satisfying

1. $\hat{S_0}(\hat{s})$, if and only if $\exists s(s \in \gamma(\hat{s}) \wedge S_0(s))$.

2. $\hat{R}(\hat{s_1}, \hat{s_2})$, if and only if $\exists s_1 \exists s_2(s_1 \in \gamma(\hat{s_1}) \wedge s_2 \in \gamma(\hat{s_2}) \wedge R(s_1, s_2))$.

3. $\hat{L}(\hat{s}) = \bigcap_{s \in \gamma(\hat{s})} L(s)$.

The universal abstraction is defined as a Kripke structure $\tilde{M} = (AP, \tilde{S}, \tilde{S_0}, \tilde{R}, \tilde{L})$ satisfying

1. $\tilde{S_0}(\tilde{s})$, if and only if $\exists s(s \in \gamma(\tilde{s}) \wedge S_0(s))$.
2. $\tilde{R}(\tilde{s_1}, \tilde{s_2})$ if and only if $\forall s_1 \exists s_2(s_1 \in \gamma(\tilde{s_1}) \wedge s_2 \in \gamma(\tilde{s_2}) \wedge R(s_1, s_2))$.
3. $\tilde{L}(\tilde{s}) = \bigcap_{s \in \gamma(\tilde{s})} L(s)$.

Intuitively, in existential abstraction, there exists a transition from an abstract state if at least one corresponding concrete state has the transition. However in universal abstraction, there is a transition from an abstract state, if all corresponding concrete states have the transition.

Figure 2.1 is an example to show the differences between these two abstractions. The original model is shown in Figure 2.1a. In the universal abstraction Figure 2.1b, a transition goes from an abstract state, A, to another abstract state B, if and only if every concrete state in A has a transition going to a concrete state in B. In the, existential abstraction Figure 2.1c, a transition goes from A to B, if and only if there exists a transition from a concrete state in A to a concrete state in B.

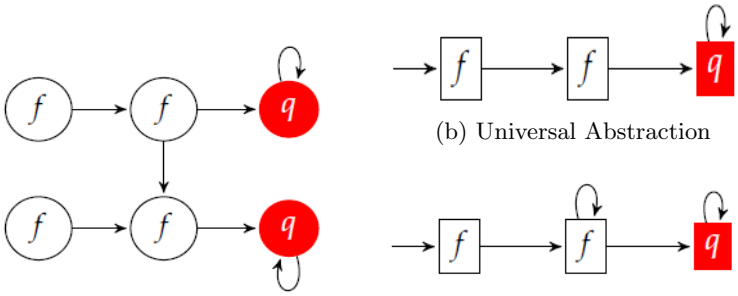


Figure 2.1: Abstraction types.

We say a model M satisfies a property φ, $M \models \varphi$, if all initial states of M satisfy φ. The existential fragment of Computation Tree Logic ($ECTL$) means the quantifier of the specification is existential and the universal fragment of Computation Tree Logic ($ACTL$) means the quantifier of the specification is universal. Existential abstraction is regarded as

an over-approximation and universal abstraction is regarded as an under-approximation. Both abstractions have preservation properties. Theorem 2.1 claims these properties.

Theorem 2.1 *Let $\hat{M}$ is the existential abstraction of M, then for ACTL* specification φ, $\hat{M} \models \varphi \Rightarrow M \models \varphi$. Let $\tilde{M}$ is the universal abstraction of M, then for ECTL* specification φ, $\tilde{M} \models \varphi \Rightarrow M \models \varphi$.*

2.1.2 Abstraction for the concrete model

We design two abstraction models, based on abstracting the transitions of the concrete model. First I divide the set of variables into two sets, according to the specifications in the input file: the set of *visible* variables and the set of *invisible* variables. The rule is all variables appearing in the specification are defined as visible variables and others are invisible. The reason to define visible variables, based on the specification, is these variables directly influence the checking procedure. We abstract the concrete states into abstract states according to the value of visible variables, such that an abstract state represents all concrete states with the same value of visible variables.

For the existential abstraction, we should make a transition from an abstract state, s_1, to another abstraction state ,s_2, if there exists some corresponding concrete state of s_1 with this transition. This can be implemented by the method making all concrete states in s_1 have transitions to all the concrete states in s_2. So, the refinement step will delete the additional transitions, since the existential abstractions have more behavior than the concrete model.

For the universal abstraction, we should make a transition from an abstract state, s_1, to another abstraction state, s_2, if all the corresponding concrete states of s_1 have this transition. This can be implemented by the method of deleting transitions, if not all the concrete states have the transition. During the refining process, the universal abstraction will add certain transitions, as the universal abstraction deletes some behaviors from the concrete model.

If we intend to check whether a model satisfies the property, φ, the model checker will first calculate a set of states, S, that satisfied the property, φ. Then, if the initial states set is a subset of satisfied states ($I \subseteq S$), we can conclude that model satisfies φ. Considering the initial states only is used in the final decision, we redefine the abstract initial states to be the

same as the concrete initial states in the two abstraction models. This will not destroy the preservation property for the two abstractions since the concrete initial states are only a subset of the abstract initial states in the two abstraction models.

2.1.3 Algorithm of dynamic model checking

Algorithm 2 is the main function for dynamic model checking. It first calculates the reachable model, which means all states in the RM are reachable from the initial states. Then, it constructs the existential abstract model and calls the function UniversalCheck, if the property is ACTL or constructs the universal abstract model and calls the function ExistentialCheck, if the property is ECTL.

Algorithm 2 AbstractCheck

Input: Original Model M and Property φ.

1: $RM = reachable(M)$
2: **if** φ is ACTL **then**
3: EM=existential_abstraction(RM).
4: UniversialCheck(EM, φ)
5: **else**
6: AM=universal_abstraction (RM)
7: ExistentialCheck(AM, φ)
8: **end if**

There are three values returned by any check function: (1) SAT means all the initial states satisfies φ,(2) $NSAT$ ($NEGATE\_SAT$) means all the initial states do not satisfy φ, (3) $PSAT$ ($PARTIAL\_SAT$) means some of the initial states satisfying φ, but others are not.

If the property is ACTL, then Algorithm 3 is called. It first checks the property φ, in the existential abstract model EM. If EM satisfies φ, then due to the preservation, theorem, the concrete model satisfies φ. If EM equals the concrete model, then the result is UNSAT. If φ only contains an AX or AG property, then the counterexample of φ is a path containing only EX or EF. According to Theorem 2.5 ,the concrete model does not satisfy the property, if the existential abstract model does not satisfy the property, because the counterexample cannot be spurious. If we cannot obtain the results at this time, then I build the universal abstract model, AM, to check the negation of the property, φ. If the result for AM is not $NEGATE\_SAT$,

then there must exists at least one initial state that satisfies $\neg\varphi$. According to the fact the negation of ACTL is ECTL and the preservation theorem of universal abstraction, the concrete model does not satisfy φ because there exists at least one initial state that satisfies $\neg\varphi$. So, check $\neg\varphi$ by AM is not NSAT, this means there exists some initial states satisfying $\neg\varphi$. Then, the original model also has some initial states satisfying $\neg\varphi$. The result is UNSAT, as not all initial states satisfy φ. cx is the counterexample output by EM when checking φ. FNE (First Nonexisting Edge) is a function to return an edge in the counterexample, which does not exist in the universal abstraction. Refine is a function to refine the EM and AM, according to the edge returned by FNE. Algorithm 4 is to explain the function refine. After separate s_1 and s_2, the transitions related with s_1 and s_2 in EM and AM should be modified accordingly. Then, the new EM is used to check until a final result is obtained.

Algorithm 3 UniversalCheck

Input: Existential Abstract Model EM and ACTL Property p.

```
1: if (Check(EM, φ) = SAT) then
2:   return SAT
3:   if EM is equivalent to RM or (the counterexample of φ is a path) then
4:     return UNSAT
5:     AM = UniversalAbstraction(RM)
6:     if (Check(AM, ¬φ)! = NSAT) then
7:       return UNSAT
8:       cx = CounterExample(EM, φ)
9:       flag=FNE(cx,AM)
10:      EM'=Refine(flag);
11:      UniversalCheck(EM', φ)
12:    end if
13:  end if
14: end if
```

If the property is ECTL, then Algorithm 5 is called. It first checks for φ in the universal abstract model AM and returns SAT, if AM satisfies φ, according to the preservation theorem. If the abstract model equals the concrete model, then the function terminates, resulting in $UNSAT$. If the authentic result cannot be obtained from the above steps, then construct an existential abstract model, EM, and check whether φ is satisfied or not. If $Check(EM, \varphi)$ is not SAT, there exists at least one state that satisfies the negation of φ ($\neg\varphi$). According to the

fact the negation of ECTL is ACTL and the preservation of existential abstraction, it can be concluded there exists at least one concrete initial state that satisfies $\neg\varphi$ in the concrete model. It returns $UNSAT$, because not all concrete initial states satisfy φ. The function $Witness$ is designed to generate a witness wn in EM for the ECTL property, φ. The beginning state of wn is one of the initial states which does not satisfy φ in AM. The Algorithm 4 is called to refine the two abstractions based on the witness. Then the new AM is used to check the property iteratively.

Algorithm 4 Refine

Input: An edge $flag$.

1: m and n are start point and end point of $flag$
2: $s_1 = backward(n) \wedge concrete(m)$
3: $s_2 = concrete(m) - s_1$
4: EM' =RefineEM(EM)
5: AM' =RefineAM(AM)

Algorithm 5 ExistentialCheck

Input: Universal Abstract Model AM and ECTL Property φ.

1: **if** $Check(AM, \varphi) = SAT$ **then**
2: **return** SAT
3: **if** AM is equivalent to RM **then**
4: **return** UNSAT
5: **if** $Check(EM, \varphi)! = SAT$ **then**
6: **return** UNSAT
7: $wn = Witness(EM, \varphi)$
8: flag=FNE(wn,AM)
9: AM'=Refine(flag)
10: ExistentialCheck(AM',p)
11: **end if**
12: **end if**
13: **end if**

Example 2.2 *Consider the transitions in Figure 2.2 and for simplicity we only show the transitions related. The concrete transitions are in Figure(a) and Figure(c). Before refinement, the existential abstract model EM and the universal abstract model AM are shown in Figure(b). There are three abstract states:* $a = \{1, 2, 3\}$, $b = \{4, 5, 6\}$ *and* $c = \{7, 8, 9\}$. *The edge* $b \to c$ *in the path* $\pi : a \to b \to c$ *is in EM not in AM. Then, using Algorithm 4, LOTUS separates*

the abstract state b into two abstract states, b1 and b2, such that b1 = {4, 5} and b2 = {6}. Then, the new abstractions EM′ and AM′ are obtained according to this change. In these new abstraction, π does not exist only in EM.

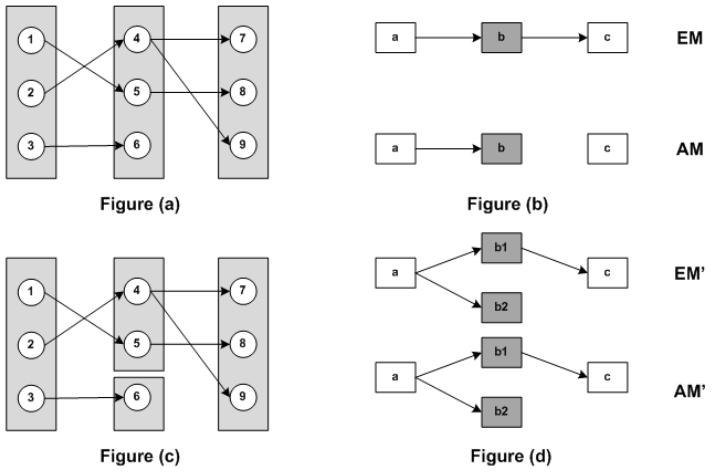


Figure 2.2: Example of refinement

2.2 Correctness of the algorithm

The Dynamic Model Checking algorithm can check the fragment of both ECTL and ACTL properties, where the counterexample is linear. This section intends to prove the correctness of the algorithms.

Theorem 2.3 *If the counterexample output by the existential abstraction is a spurious path then there are two kinds of states in the concrete model by Clark (2002):*

- *Dead-end state: Reachable, but there are no outgoing transitions to the next state in the counterexample.*
- *Bad state: Not reachable, but outgoing transitions cause the spurious counterexample. The spurious counterexamples is caused by the bad states.*

Theorem 2.4 *If all states in the model are reachable, then the spurious path counterexample does not exist in existential abstraction.*

Proof From Theorem 2.3, we know that *Bad states* are not reachable. Since we have removed the transitions outgoing from the bad states, then the *Bad states* cannot exist. This will result in no such abstract state with both bad states and dead-end states. Because all states are reachable, the path counterexample cannot be spurious in existential abstraction.

Theorem 2.5 *If the property, φ, contains only AG or AX, then the counterexample of φ cannot be spurious.*

Proof If the property, φ, only contains AG or AX, then the counterexample must contains only EF or EX. Hence, the counterexample must be a path (not a loop) as EF (EX) means there exists a path, such that the property is satisfied in the ith state (there exists a path such that the property is satisfied in the next state). According to Theorem 2.4, the path counterexample cannot be spurious, as all states are reachable. Then, if the property only contains AG or AX, the counterexample cannot be spurious. Then the algorithm can obtain the result UNSAT, if the existential abstraction does not satisfy the property containing only AG or AX.

Theorem 2.6 *If there exists an initial state in existential abstract model, EM, does not satisfy the ECTL, φ, then this initial state satisfies $\neg\varphi$.*

Proof The existential abstraction is an over-approximation abstraction, which means the existential abstraction has more behaviors than the concrete model. We claim every state in the concrete model has outgoing transitions, so all abstraction states in the Existential abstract model have outgoing edges. If every state in the model has outgoing transitions, then the negation of ECTL property, φ, is ACTL property, $\neg\varphi$. Hence, if one initial state does not satisfy the ECTL property, φ, then this state must satisfy the ACTL property, $\neg\varphi$.

Theorem 2.7 *FNE must return an edge when it is called in UniversalCheck and ExistentialCheck*

Proof If the theorem is not true then all the edges of the counterexample(witness) *cx* output by EM are in AM. If φ is $ACTL$ and the counterexample of $\neg\varphi$ exists in the universal abstraction

(AM), then $Check(AM, \neg\varphi)$ is not $NEGATE\_SAT$, and the final result is original model does not satisfy φ, since at least one initial state satisfies $\neg\varphi$. If φ is $ECTL$, wn is a witness path satisfying φ in EM, but not in AM, because the beginning state is one of the initial states that does not satisfy φ in AM. Hence, before the functions *UniversalCheck* and *ExistentialCheck*, call the function *FNE*, it will obtain the results if cx (wn) is in both AM and EM. Therefore, if calling FNE, it must return an edge of the cx (wn), which is in EM, but not in AM.

Theorem 2.8 *Algorithm 3 and Algorithm 5 will terminate.*

Proof After every refinement step, the existential abstract model and universal abstract model will be more precise than before. And, the two abstractions will be closer to the concrete model. When the abstract model equals the concrete model, the algorithm will terminate with result $UNSAT$. Hence, both algorithms terminate finally.

CHAPTER 3. LOTUS: A DYNAMIC MODEL CHECKER

I designed and developed a model checker, implementing the ideas of dynamic abstraction algorithm with C++. The new model checker is LOTUS which can check both ACTL and ECTL by the abstraction technique. The implementation details are discussed in this chapter.

3.1 Introduction

LOTUS is constructed with C++ on Linux to implement the model checker with dynamic abstraction algorithm. The main parts are paser, encoding, model checking, witness and counterexample generation, and refinement. This abstracts the concrete model, based on the input file and attempt to output the conclusion, if the the result is positive. If the result is negative, LOTUS will refine the abstraction model, based on the counterexample or witness, if the path is spurious; otherwise, output the negative result directly.

3.2 Input file

LOTUS needs the input file to describe the model and specifications. This process is achieved by the user, following the grammar of LOTUS. The example is an input file to describe the model with four states in Figure 3.1 .

The example is to describe the transitions among four states. Each state has two variables. The initial state is the state which satisfies *boday1=Hungry* and *state1=Free*. Fairness means the conditions must be satisfied each time. The specification is to check whether there exists a path, which in the future, will satisfy the conditions that *boday1=Full* and *state1=Busy*. The CTLSPEC is the token to show the specification is a CTL specification.

```
DEFINE
    Body : Full, Hungry;
    State : Free, Busy;

MODULE
    VAR
        Body body1;
        State state1;

    INIT
        body1=Hungry, state1=Free;

    TRANS
        body1=Full, state1=Free => state1=Busy;

        body1=Full => body1=Hungry;

        state1=Busy => state1=Free;

        body1=Hungry, state1=Free => body1=Full;

    FAIRNESS
        state1=Busy

    SPEC
        CTLSPEC EF(body1=Full & state1=Busy)
```

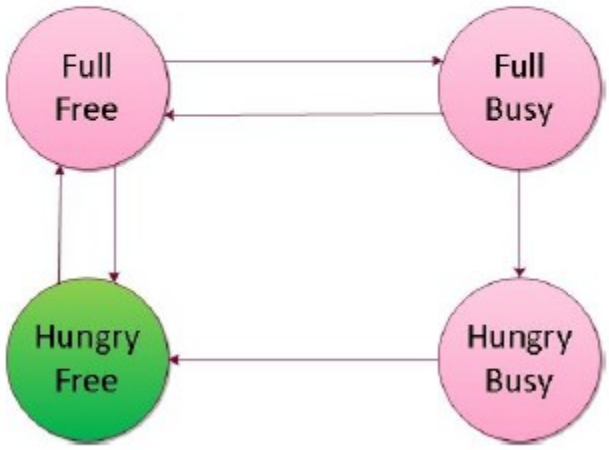


Figure 3.1: Input file of LOTUS

LOTUS has its own grammar, which means all the input file needs to follow is the Lotus Grammar. An error messages will be produced, if the input file does not have legal grammar.

3.3 Grammar for LOTUS

3.3.0.1 DEFINE

This keyword is used to define the type. There are two types allowed in Lotus.

1. Enumeration type: This type is used to define an enumeration type. The elements in this type can be characters or numbers. Example: *state: begin,continue,end;*

2. Range type: This type is used to define a range type used for numbers. The variable for this type can choose any value in the range. Example: *week: 1...7*;

3.3.0.2 MODULE

This keyword is used to indicate the beginning of a description of Module.

3.3.0.3 VAR

This keyword is used to define variables using the custom type. The variables are stored in the symbol table of LOTUS and can be used in the definition of transition. The example is *state s; week w;* (*type variable_name*).

3.3.0.4 INIT

This keyword is used to indicate the description of the model. A model can have more than one initial state. The definition should list all initial states. Each state is a list of variables with values. An example is *s=begin,week=1; s=end,w=7;*(First sentence is a state and the second sentence is another state. This initial set of states has two states).

3.3.0.5 TRANS

This keyword is used to define the transition of the model. The transition is a list of edges between two states in the Kripke structure of the model. Use => to represent the edge between two states. The example is $s = begin, w = 1 => s = end, w = 2$; If one transition relates only to a set of variables, then the transition can be written only using these variables. The other variables should maintain the same value. The example is $w = 1 => w = 2$; (This equals to $s = begin, w = 1 => s = begin, w = 2; s = continue, w = 1 => s = continue, w = 2; s = end, w = 1 => s = end, w = 2;$).

3.3.0.6 FAIRNESS

Fairness means these conditions should appear infinite times in the satisfied path. The definition of fairness is like :*s=begin*(This means s equals to begin should appeared infinite times in the satisfied path).

3.3.0.7 SPEC

LOTUS version1.0 only accepts CTL(Computation Tree Logic) specifications so all specifications should be written after the keyword *CTLSPEC*. The grammar for CTL specifications

is as follows:

- AG(w=1): For all paths, the condition of w=1 should be satisfied all the time.
- AF(w=1): For all paths,the condition of w=1 should be satisfied in the future.
- AX(w=1): For all paths,the condition of w=1 should be satisfied in the next step.
- A[w=1 U w=7]: For all paths, before w=7, all states should satisfy w=1.
- EG(w=1): There exists a path, such that the condition of w=1 should be satisfied all the time
- EF(w=1): There exists a path, such that the condition of w=1 should be satisfied in the future.
- EX(w=1): There exists a path, such that the condition of w=1 should be satisfied in thenext step.
- E[w=1 U w=7]: There exists a path, such that before w=7 all the states should satisfy w=1.
- EX(EG(w=1)): There exists a successor, such that from this states on, there exists a path that satisfy w=1 all the time.
- !(w=1): The initial states do not satisfy the condition of w=1.
- w=1 & s=begin: The initial states do not satisfy the condition of w=1 and s=begin.

3.3.0.8 Comment

LOTUS uses // to represent a comment.

3.3.0.9 Example

DEFINE

type1 : start, end;

type2 : 0...4;

MODULE

VAR

type1 system;

type2 feelings;

INIT

system=start,feelings=0;

system=end,feelings=0;

TRANS

system=start => system=end;

feelings=0 => feelings=1;

feelings=1 => feelings=2;

feelings=2 => feelings=3;

feelings=3 => feelings=4;

feelings=4 => feelings=3;

//FAIRNESS

SPEC

CTLSPEC E[feelings=0 U !(system=end)];

CTLSPEC AG(system=start);

CTLSPEC AX(system=start);

CTLSPEC AF(feelings=4)

3.4 Parse

Parsing in LOTUS is based on Flex and Bison in Linux. Each node is a basic structure in parsing. *flex* and *Bison* are used to analyze the lexical and syntactic parts, according of the input file. They build the parsed tree to represent the model and specifications used in the future. The parsed tree is another representation of the model.

The parsed tree consists of a basic node, in Figure 3.2. Each tree node has two children, left and right. line_num is the number of lines in the input file. If the node is the leaf, then it will have a name for this node. This parsed tree is used to represent all the information of the model: initial states, transitions and specifications.

```
NodeType _type;

union node_p{
      struct{
            NodePtr left;
            NodePtr right;
            int line_num;
      } content;
      char name[sizeof(content)/sizeof(char)];
} _property;
```

Figure 3.2: Definition of a node structure

3.5 Encode

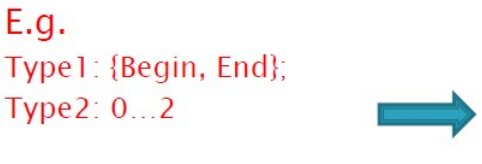
As discussed previously, most model checkers use BDD to encode the model to add speed. There are also many BDD packages or library for use. LOTUS is based on the Buddy 2.4 by Lind-Nielsen (2004). This library provides all BDD operations in both C and C++. After encoding the parsed tree of the model into BDD, LOTUS uses BDD directly to calculate the transitions and checks the process.

The Encode step is to encode each state with a binary value. The first step calculates the number of boolean variables used in BDD. The second step encodes each self defined type. Then, encoding the states, encodes each self defined variable. By the same way, LOTUS encodes the transitions and specifications.

To encode the transitions, representing previous states and the next states in one BDD formula is needed. Hence,the formula needs twice the variables to represent states and transitions. One part is to represent the current state and the next part is to represent the next state. If the BDD is only used to represent a single state, then only use the current field to represent this state and set 0 of the next state representation field. For the purpose of speeding up the calculation, LOTUS represents the previous state using boolean variables in odd locations and the next states with boolean variable in even locations. The Preimage Computation calculates the previous state by computing $\exists y.[T(x,y) \wedge Z(y)]$. Image Computation calculates the next

state, according to the transitions by computing $\exists x.[T(x,y) \wedge Z(x)]$. Here $T(x,y)$ represents the transitions and $Z(x)$ signifies the current states and $X(y)$ signifies the next state.

Here is an example to explain the encoding in Figure 3.3. The user defines two new self-defined types. One is called *Type1* with two values. One boolean variable can be used to encode this, where 0 represents *Begin* and 1 represents *End*. The other self-defined type is *Type2*, having the enumerative type with 3 value from 0 to 2. Then, two boolean variables need to encode this, where 00 represents *0*, 01 represents *1*, and 10 represents *2*. After encoding the model, LOTUS uses this encoded BDD to check whether the model satisfies certain specifications.



Type Value	Encoded Value
Begin	0
End	1
0	00
1	01
2	10

Figure 3.3: Encode example

3.6 Abstraction

LOTUS implements the two abstractions which are existential abstraction and universal abstraction. When the input file has been encoded as BDD, LOTUS will construct abstraction models according to the specification. Before abstraction, LOTUS first calculates the set of reachable states from the initial states. The abstraction model is used to check whether the specification is satisfied or not. These two abstractions are constructed before the checking procedure.

If the specification is ACTL then LOTUS calls the existential abstraction function to abstract the model. The existential abstraction means the abstraction model making a transition from an abstract state, if at least one corresponding concrete state has the transition. Here is an example to show the Existential Abstraction in Figure 3.4 from Sinha (2005).

If the specification is ECTL, then call the universal abstraction function to abstract the

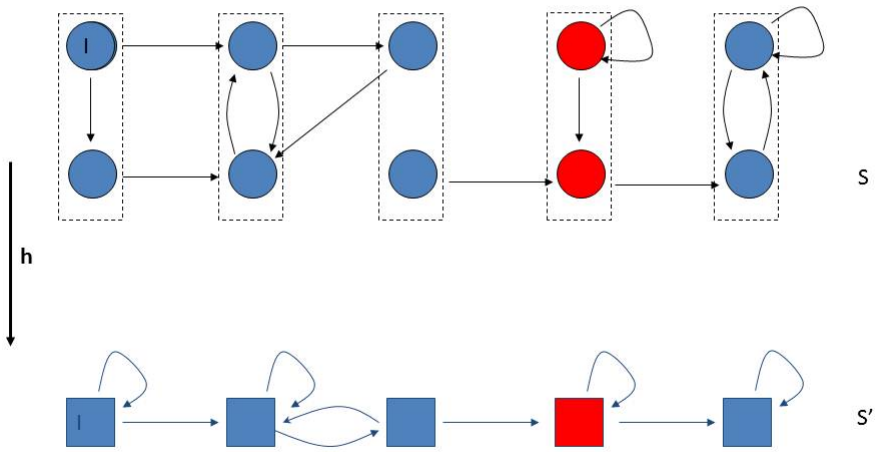


Figure 3.4: Existential abstraction

model. The universal abstraction means in the abstraction model making a transition from an abstract state, if all corresponding concrete states have the transition. Here is an example to explain the Universal Abstraction in Figure 3.5 from Sinha (2005)

3.7 Model checking

LOTUS will use both existential abstraction and universal abstraction to check the model according to the algorithm in Chapter 2. The model checking algorithm in LOTUS is according to the following:

1. All initial states should satisfy the specification in the state.
2. $EX(f) = backward(f)$.
3. $EG(f) = \nu Z[f \wedge EX\ Z]$.
4. $E[f\ U\ g] = \mu Z[g \vee (f \wedge EX\ Z)]$.
5. $AX(f) = \neg EX \neg f$.

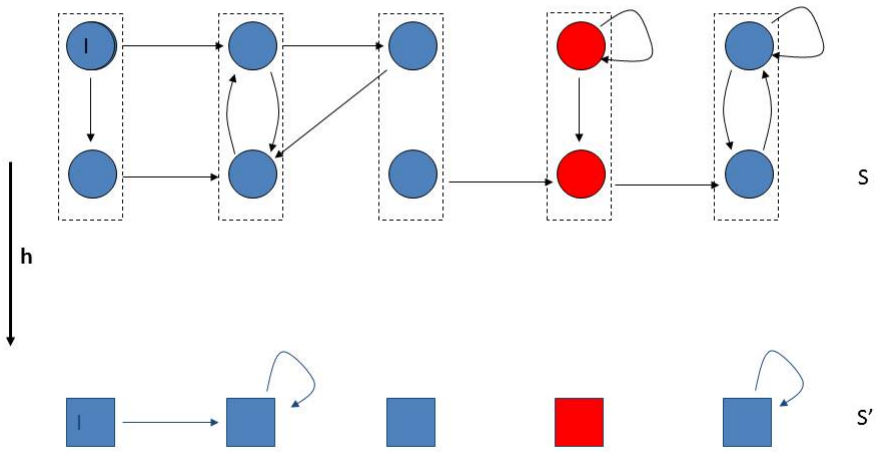


Figure 3.5: Universal abstraction

6. $EF(f) = E[true\, U\, f]$.

7. $AF(f) = \neg EG \neg f$.

8. $AG(f) = \neg EF \neg f$.

9. $A[f\, U\, g] = \neg E[\neg g\, U\, \neg f \wedge \neg g] \wedge \neg EG \neg g$.

The basic functions are as follows. Other functions obtaining the set of states satisfying certain properties will call these basic functions to calculate the corresponding set of states.

1. *satsify* checks whether a state satisfying certain property;

2. *negate* intends to return set of states, which does not satisfy the certain property;

3. *backward* calculate the set of previous states, according to the transition;

4. *forward* calculate the set of next state, according to the transition;

5. *EX* calculate the EX property by the function *backward*;

6. *EG* is used to get the set of states satisfying EG property by the greatest fixpoint algorithm;

7. *EU* is a function obtaining the set of states satisfying the EU property by the least fixpoint algorithm.

The least fixpoint function and the greatest fixpoint function are according to Algorithm 6 and Algorithm 7. LOTUS uses these two basic fixpoint functions to implement all the models checking functions.

Algorithm 6 Leastest Fixpoint Algorithm

Input: τ: Predicate Transformer: Predicate.

1: $Q = False$
2: $Q' = \tau(Q)$
3: **while** $(Q \neq Q')$ **do**
4: $\quad Q = Q'$
5: $\quad Q' = \tau(Q')$
6: **end while**
7: return (Q)

Algorithm 7 Greatest Fixpoint Algorithm

Input: τ: Predicate Transformer: Predicate.

1: $Q = True$
2: $Q' = \tau(Q)$
3: **while** $(Q \neq Q')$ **do**
4: $\quad Q = Q'$
5: $\quad Q' = \tau(Q')$
6: **end while**
7: return (Q)

3.8 Counterexample or witness generation

To refine the model, LOTUS must generate a counterexample or witness, if the property cannot be obtained in the first place. The data structure for generating a counterexample or witness is **stack** and **vector**. Start from the initial set of states and store the set of states forward in the stack. When the target states are found, then a vector is used to construct the counterexample or witness. The process starts from the target set of states and then obtains

the previous states from the stack and push them in the vector. The counterexample or witness is generated from this vector. As counterexample or witness is only one path (including loop), there is a function in the BDD library that can generate one path from the vector of sets of states.

Here is an example to elaborate the procedure of generating the counterexample in Figure 3.6. The CTL specification checks the property, $EF(g)$. Beginning with the initial states and pushing the set of states is the next states of the top of the stack. In the example, to check whether there is a path to the states satisfying the property g, it begins from the initial states. At each step, it calculates the next state and pushes it to the stack. Until LOTUS finds the state satisfying the g, the procedure stops. To construct the path, call the function in the BDD library which can choose one state from a set of states. Hence, start from the state satisfying the property, g, and calculate the previous states. Then, push the state to the vector.

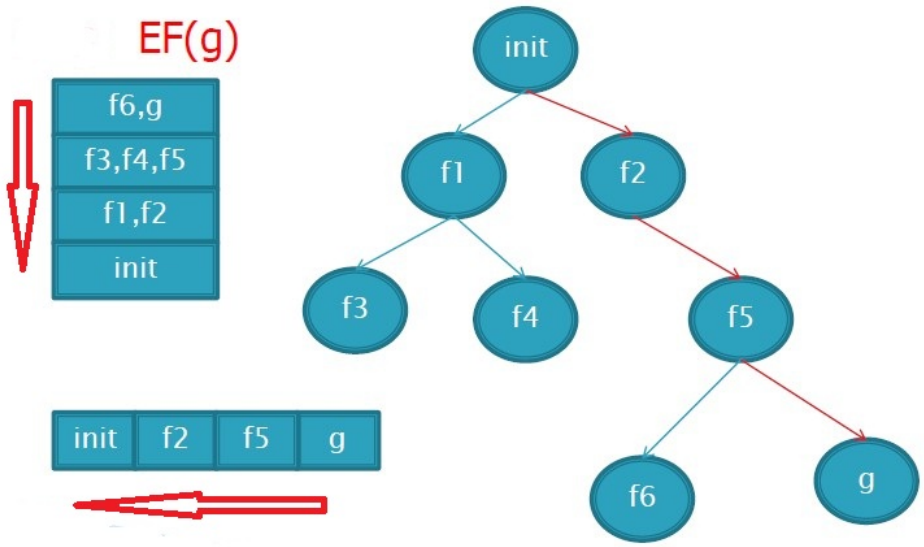


Figure 3.6: Counterexample generation

3.9 Refinement

It is ideal if the results can be generated directly from the abstraction model. However, if the counterexample is generated, LOTUS must modify the model and recheck the refined model. Since LOTUS uses two abstraction (existential abstraction and universal abstraction),

the refinement must be completed on both abstractions. The key idea is to find the abstract states, including bad states and dead-end states, and then separate the abstract states into two new abstract states to remove the counterexample.

The main steps for the refinement in LOTUS are: (1) Obtain the abstract state which causes the counterexample; (2) Separate the abstract states into two abstract states to make the model more precise to check the specifications;(3) and modify the existential abstraction and universal abstraction according to the refined transitions.

To determine the transition for the reason to refine the abstraction model, LOTUS traverses the counterexample from the beginning to the end attempting to locate the transition is in the existential abstraction but not in the universal abstraction. After finding the transition, LOTUS will refine both the existential abstraction and the universal abstraction, since the existential abstraction has more transitions than the original model, but the universal abstraction has less transitions than the original model. Separating the bad states and dead-end states into two abstract states is the key to refinement. Refine the model related with these transitions, which means deleting extra transitions in the existential abstraction and adding transitions in the universal abstraction. LOTUS will check the model and refine the model until the final results are obtained.

CHAPTER 4. RESULTS

In this chapter, I select two different problems to demonstrate the power of this new approach. One is Philosopher Dining Problem and the other is Philosopher Dining With Token. These two cases will be discussed in details in the following section. The experiment results are presented also.

4.1 Experiment design

4.1.1 Philosopher dinning problem

The original Philosopher Dining Problem can be seen in Figure 4.1. The description of this original problem is as follows. Five philosophers sit around a table with a bowl of spaghetti and fork. They do not speak. The rule for eating the spaghetti is each philosopher must alternately think and eat. A philosopher can only eat, when he has both left and right forks. But, a deadlock can occur because each fork can be held by only one philosopher and the philosopher needs to wait for the other fork. After he finishes eating, he needs to put down both forks so they become available to others. This stipulates a philosopher can grab the fork from the right side or the left side, as it becomes available, but cannot start start eating before both are available at the same time.

In the experiment, a different number of philosophers are tested. Each fork can be picked by the left person or right person. The person can eat only if he has both a left fork and a right fork. Each person has different states: want (to eat), eating, thinking, left (having left fork), right (having right fork). Each fork has different states: left (belongs to the left person), right (belongs to the right person), none (laying on the table). The CTL specification is a deadlock situation will never happen. In fact, a deadlock will occur in this schedule, since there is one

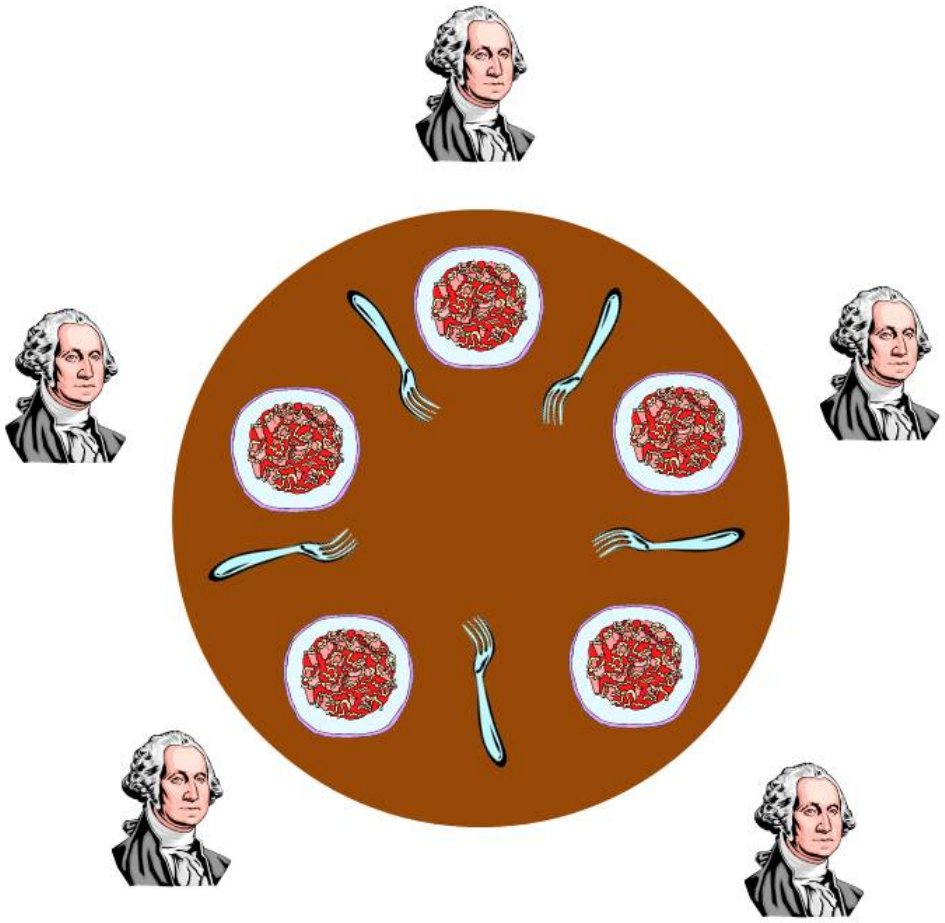


Figure 4.1: Philosopher dining problem

possibility that all the people hold only one fork and wait for the other fork. Hence, the check specification is $EG(!deadlock)$ and the checker should return UNSAT.

4.1.2 Philosopher dining with token

There are many methods to avoid a deadlock in the Philosopher Dinning Problem. In this experiment, I choose the method using Token. The key point is to use a serializing token to solve this problem. The token will rotate from the first person and proceed to the next person. If one philosopher wants to eat he must wait for the token and transfer the token to the next person after he finishes eating. In this case, a deadlock situation cannot occur since only one person can have the token and all other people must eat need to wait. Even though this may

waste time, this method can solve both the starvation problem and the deadlock problem. Because the token will be delivered in sequence, everyone wanting to eat will definitely have the token and eat with the token. The method is explained in Figure 4.2 from wikipedia.

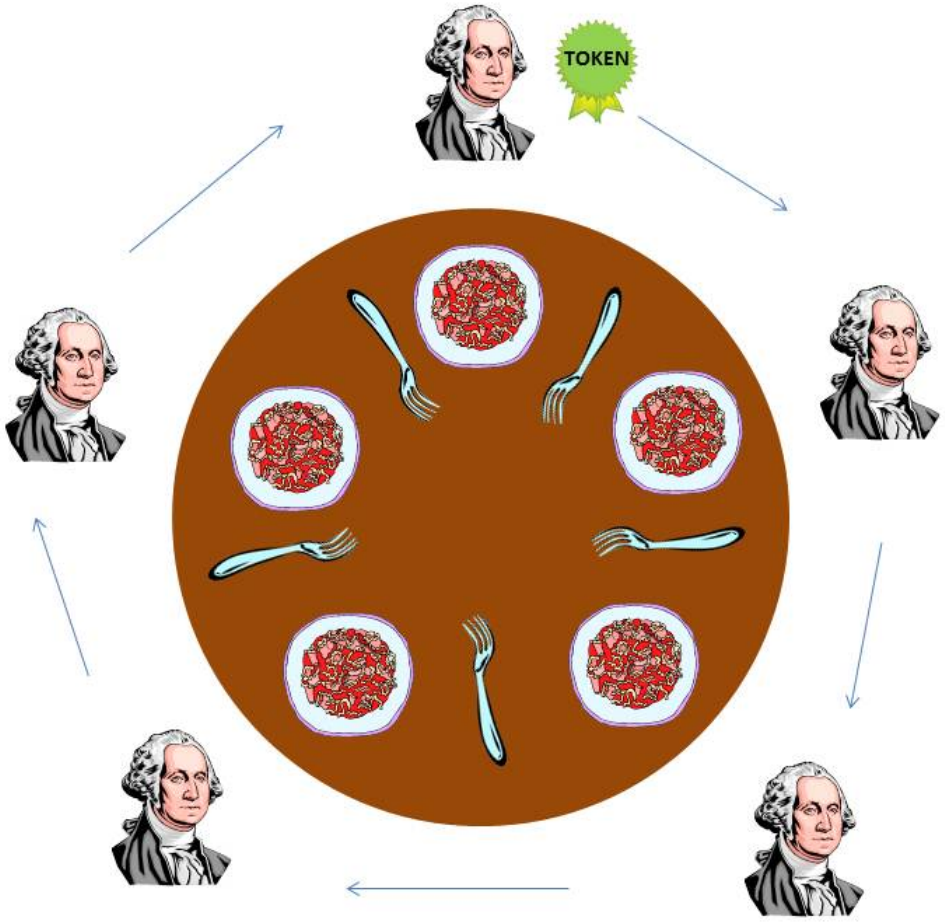


Figure 4.2: Philosopher dining problem with token

In the test cases, the number of token states equals the number of people around the table. The token belongs to the first person in the initial state. At each transition, the token moves on to the next person. Everyone can eat when the token is available in his place. The CTL specification in this problem also checks whether everyone can eat, if he wants.

4.2 Experiment result

I implemented Dynamic Model Checking with *LOTUS*. All the experiments were performed on a desktop machine running on Intel i7-2600K Quad-Core Processor 3.4 Ghz, 8 MB Cache, 8GB RAM and Ubuntu 11.10.

Philosophy Dinning Problem (PDP) is a popular problem in the verification field. First I completed the experiment on the original philosophy problem without any strategy. Thus, starving, i.e.,deadlock, is possible. The performance of an original model checking algorithms with a dynamic model checking algorithm is shown in Table 4.1. The objective is to show how the performance among different philosophers in the problem.

Table 4.1: Performance of PDP

PDP	Original Model Checking		Dynamic Model Checking	
	time(s)	BDD node	time	BDD node
PDP100	1.6121	4975002	1.3	5382594
PDP120	34.5182	9249676	2.57216	9769425
PDP140	83.6252	15202716	14.896	15852483
PDP160	146.481	22808116	123.904	23570117
PDP180	248.32	32257876	206.309	33114327
PDP200	22.1054	43811459	14.8689	44729118
PDP220	34.2141	62310007	20.9533	62850471
PDP240	43.8907	75394982	25.737	62850471
PDP260	54.5671	89989005	31.0499	75611318
PDP280	81.8931	25172720	45.6669	89879724
PDP300	100.754	25172720	51.1672	25071445
PDP320	118.555	74198613	58.7837	50266764
PDP340	770.888	28034760	81.9	72343634
PDP360	179.067	49341494	85.1733	19841137
PDP380	209.365	87626219	95.374	83963969

A comparison of the time consuming and BDD node number between original and abstraction are shown in Figures 4.3 and Figure 4.4

There are many efficient strategies to prevent a deadlock of Philosophy Dining Problem. One is using a token, which means the person having the token can have the dinner and then pass the token clockwise. We test this using the ACTL AF property, which means each person can have the dinner in the future. For some problems, many iterations of refinement

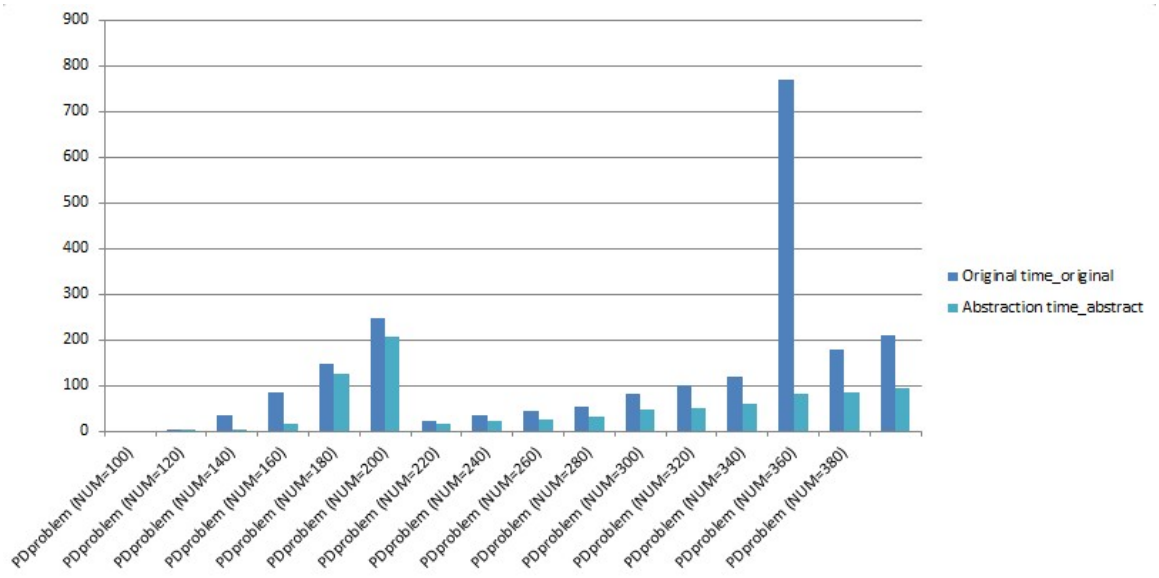


Figure 4.3: PDP time comparison

will cost too much time. Hence, we use a threshold to indicate if the number of iterations is more than the threshold, then using the concrete reachable model (the concrete model after reachability calculation). As we have already paid the price for calculating reachability, the checking procedure on the reachable model is also much faster than checking using original concrete model. In the experiment, we set the threshold to 15. The performance of the original model checking algorithm with dynamic model checking algorithm of the Philosophy Dinning Problem With Token (PDT) is shown in Table 4.2. In this table, threshold is used as a max refinement time, which means if LOTUS can output the result with the number of refinements less than the threshold it directly outputs the results or it checks the original model after computing reachability. The comparison is between original model checking and dynamic

4.2.1 Analysis

In this chapter, two cases with a Philosopher Dining Problem are compared between dynamic abstraction algorithm and original algorithm. These two cases tested both ACTL and ECTL properties. From the experiment results, these conclusions can be obtained as follows:

- If the state space is not huge, the differences between original algorithm and dynamic abstraction algorithm are not obvious. Because even abstraction may reduce the checking

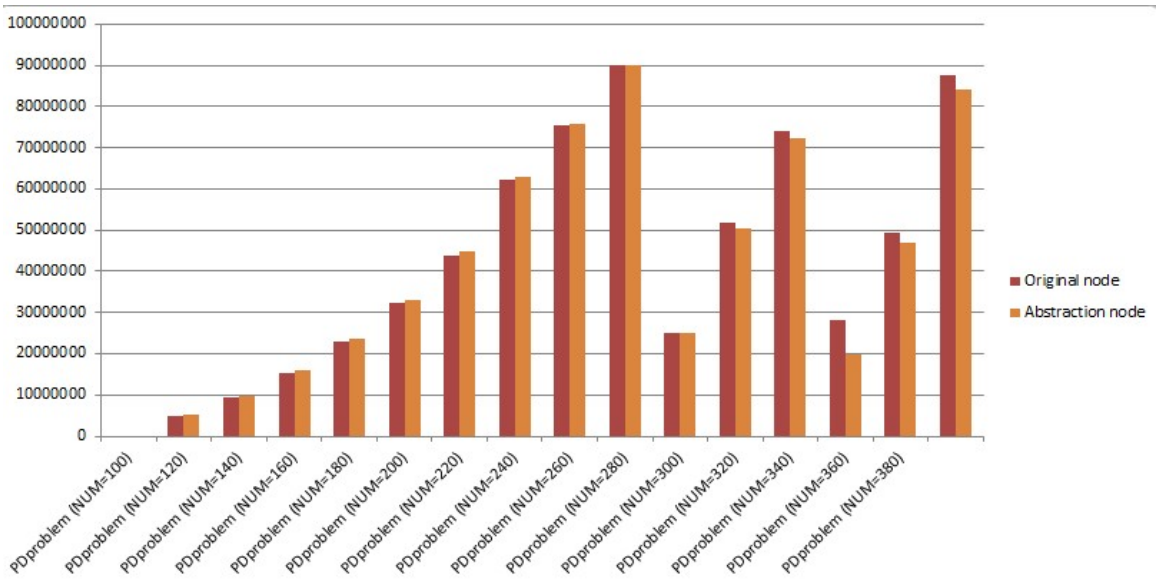


Figure 4.4: PDP BDD node comparison

time consuming, calculating both existential abstraction and universal abstraction needs extra time and space.

- If the state space is larger, then the dynamic abstraction algorithm is better than the original algorithm. The abstraction method can reduce the time consuming checking to speed process. If the results can be obtained after less refinement, then the effect is obvious from the comparison.

- From the Philosopher Dining Problem with the Token, we can conclude the more abstraction refinement, the more time consuming, because the more refinement, the more precise the abstraction model. In this case, I designed a threshold, which is the max number of the times for refinement. If LOTUS can compute the final results then it will output. Otherwise after the threshold, it will calculate using the original model after reachability computing.

- Dynamic abstraction algorithm is not always better than the original algorithm because it costs more time to refine after certain times. The more precise the abstraction model, the less effective. In this thesis, reachability is calculated in the beginning and is better in these two cases. However, if the irrelevant state is less, calculating reachability may

Table 4.2: Performance of PDTOKEN

PDTOKEN	Original Model Checking		Dynamic Model Checking threshold=15	
	time(s)	BDD node	time	BDD node
PDTOKEN25	2.27614	162340	2.59216	863996
PDTOKEN35	21.8854	486884	7.68848	1734693
PDTOKEN45	129.024	891182	15.781	1259861
PDTOKEN55	1411.11	2805999	27.4617	1938920
PDTOKEN65	NO	NO	42.3666	2838231
PDTOKEN75	NO	NO	59.4971	3872419
PDTOKEN85	NO	NO	76.6608	5047334
PDTOKEN95	NO	NO	107.995	6452952
PDTOKEN115	NO	NO	145.385	10007719
PDTOKEN125	NO	NO	211.237	13981761
PDTOKEN135	NO	NO	287.894	18462855

not be the best choice.

CHAPTER 5. SUMMARY

5.1 Conclusions

In this thesis, I provided a new abstraction strategy called dynamic abstraction algorithm, which abstracts the model with both existential abstraction and universal abstraction. This algorithm can checked both ECTL and ACTL by using the abstraction method. Dynamic abstraction algorithm modifies the original abstraction method with these points:

1. The key point for the dynamic abstraction algorithm it abstracts the model according to specifications from the input file. It first builds one abstraction to check and then outputs the results, if the abstraction model is sufficient to output. Otherwise, both existential abstraction and universal abstraction are used to refine the abstraction to recheck the abstraction model with more precision. Hence, it dynamically decides which abstraction is built and when to refine the model.

2. Refinement in most abstraction methods is to split some abstraction states into several states to make the abstraction model more precise. In this thesis, I provided another abstraction technology, called transition abstraction. When the transition causing the spurious counterexample or witness is found, LOTUS will modify all transitions in both existential abstraction and universal abstraction to split the bad states and dead-end states. This strategy is designed to reduce the burden of code and increase continuity.

3. In the experiment, LOTUS finishes the reachability computation. Since the test cases in the thesis had too many irrelevant states the specification checking, computing reachability had a great effect on this. However, in some cases, most states are related with the specifications or the state space is huge. Therefore obtaining reachability is not a good

choice before checking.

4. I elaborated the process of generating the counterexample or witness in this thesis. The point here is the same mechanism generates the counterexample and witness. The generation process is to select one path (loop) used to refine. Hence, it does not cost extra time to calculate the witness, since it is the same procedure as the counterexample calculation. Since in the encoding part may encode some useless state, which does not exist in the original model into the abstraction model, to make the counterexample exist in the abstraction model, I modified the counterexample generation to ensure every counterexample or witness generated exists in the abstraction model, even though it may be spurious.

5.2 Discussion

I focus primarily on the abstraction strategy on both ECTL and ACTL specifications. In this thesis, the dynamic abstraction algorithm is presented with details and a software LOTUS is also developed to implement this idea.

The dynamic abstraction algorithm attempts solve the original abstraction, which can only check ACTL with the universal abstraction. However, constructing the universal abstraction is also time consuming, if the state space is large. In this thesis, this algorithm utilize more efficiency on the experiment. In future work, improving the speed of building the universal abstraction is a main research field. This algorithm uses transition abstraction, which looks good to implement and easier to refinement than the original algorithm. For refinement, I used the idea to separate the bad states and dead-end states by modifying the transitions. Hence, the abstraction in this algorithm is abstracting the transitions and not the states. In future work, a comparison between the transition abstraction and the state abstraction needs completion.

LOTUS is a software which implements the dynamic abstraction algorithm by using C++ on Linux. In this study, the experiment on this software is shown in Chapter 4. LOTUS uses the BDD package Buddy, which has the C++ interface for the BDD library. As most operations

related with BDD have been done directly by Buddy, it may be faster to modify the package in the future to increase speed for these operations, such as obtain one counterexample among the counterexample vector.

In assumption, dynamic abstraction algorithm is presented with software called LOTUS in this research. I also discussed LOTUS with experimented results. Abstraction is an efficient method to increase speed and save space. However, refinement is a problem, if the abstraction is not efficient. In the future, attempts to construct a more precise abstraction, based on specification, is the popular topic. In LOTUS, abstraction is based on all the predicates presented in the specification. Also, it may useful to abstract the model, based on the predicates both presented in the specifications and related with the CTL property. Thus this algorithm provides a new idea to use both existential abstraction and universal abstraction to solve the model checking problem within abstraction technology.

BIBLIOGRAPHY

Adiya, Z., S. S. and Ashish, T. (2012). Timed relational abstractions for sampled data control systems. *CAV*.

Balarin, F. and Sangiovanni-Vincentelli, A. (1993). An iterative approach to language containment. *CAV*.

Bensalem, S., B. A. L. C. S. J. (1992). Property preserving simulations. *CAV*.

Bensalem, S., L. Y. and Owre, S. (1998). Computing abstractions of infinite state systems compositionally and automatically. *CAV*.

Berezin, S., B. A. C. E. and Zhu, Y. (1998). Computing abstractions of infinite state systems compositionally and automatically. *CAV*.

Bryant., R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, pages 677–691.

Burch, J. and Dill, D. (1994). Automatic verification of pipelined microprocessor control. *CAV*.

Clark, E., G. O. J. S. L. Y. V. H. (2002). Counterexample guided abstraction refinement. *CAV*.

Clark, E., L. Y. J. S. V. H. (2001). Counterexamples in model checking. tech. rep. *Carngie Mellon University. CMU-CS-01-106.*

Clarke, E.M., E. E. (1981). Design and synthesis of synchronization skeletons using branching time temporal logic. *Kozen, D.(ed.) Logic of Programs.*

Clarke, E.M., G. O. L. D. (1994). Model checking and abstraction. *ACM transactions on Programming Languages and Systems*, pages 1512–1542.

Das, S., D. D. and Park, S. (1999). Experience with predicate abstraction. *CAV*.

E. M. Clarke, A. Gupta, J. K. and Strichman., O. (2002). Sat based abstraction refinement using ilp and machine learning techniques. *CAV*.

Edmund, M. C. (2003). Sat-based counterexample guided abstraction refinement in model checking. *Automated Deduction C CADE-19*.

Edmund, M. C. (2004). Sat-based counterexample-guided abstraction refinement. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*.

Edmund, M.C., O. G. D. A. (1999). *Model Checking*. MIT press.

Edmund.M.C (2008). The birth of model checking. *Springer-Verlag Berlin, Heidelberg*.

Fura, D., W. P. a. S. A. (1993). Abstraction techniques for modeling real-world interface chips. *International Workshop on Higher Order Logic Theorem Proving and its Applications*.

Georges, M., F. P. and Christoph, S. (2011). Fully symbolic model checking for timed automata. *CAV*.

Govindaraju, S. and Dill, D. (1998). Verification by approximate forward and backward reachability. *Proceedings of the international Conference of Computer-Aided Design*.

Graf, S. (1994). Verification of distributed cached memory by using abstractions. *CAV*.

Ho, P.-H, I. A. and Kam, T. (1998). Formal verification of pipelione control using controlled token nets and abstract interpretation. *Proceedings of the International Conference of Computer-Aided Design*.

Jones, R.B., S. J. a. D. D. (1998a). Reducing manual abstraction in formal verification of out-of-order execution. *Form.Meth. Comput.-Aided Des*.

Jones, R.B., S. J. a. D. D. (1998b). Reducing manual abstraction in formal verification of out-of-order execution. *Form.Meth. Comput.-Aided Des*.

Kleene, S. (1971). Introduction to metamathematics. *Wolters-Noordhoff, Groningen,*.

Kurshan, R. (1994). Computer-aided verification of coordinating processes. *Princeton University Press.*

Lee, W., P. A. J. J. G. and Somenzi, F. (1996). Tearing based abstraction for ctl model checking. *Proceedings of the International Conference of Computer-Aided Design.*

Lind-Nielsen, J. (2004). Buddy- a binary decision diagram package. *http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy/index.html.*

Lind-Nielsen, J. and Andersen, H. (1999). stepwise ctl model checking of state/event systems. *CAV.*

Luca, P. and Armando, T. (2010). An abstraction-refinement approach to verification of artificial neural networks. *CAV.*

Matt, F., R. J. S. J. T. R. P. P. H. S. and Vinod, Y. (2012). Efficient runtime policy enforcement using counterexample-guided abstraction refinement. *CAV.*

McMillan, K. (1993). Symbolic model checking: An approach to the state explosion problem. *Kluwer Academic Publishers.*

Pardo, A. (1997). Automatic abstraction techniques for formal verification of digital systems. *Ph.D. dissertation, Dept. of Computer Science, University of Colorado at Boulder, Boulder Colo.*

Pardo, A. and Hachtel, G. (1998). Incremental ctl model checking using bdd subsetting. *Design Automation Conference.*

Randal, E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE transactions on Computers.*

S.Graf, H. S. (1997). Construction of abstract state graphs with pvs. *CAV.*

Sinha, N. (2005). Abstraction in model checking. *www.cs.cmu.edu/ emc/15817-s05/cegar.ppt.*

Tarski, A. (1995). A latice-theoretical fixpoint theorem and its applications. *Pacific J.Math.*, pages 285–309.

Thomas, B. and Sriram, K. (2001). Automatically validating temporal safety properties of interfaces. *SPIN '01 Proceedings of the 8th international SPIN workshop on Model checking of software.*

Thomas, B., R. J. R. M. a. K. L. (2004). Abstractions from proofs. *POPL.*

Wolper, P. and Lovinfosse, V. (1989). Verifying properties of large sets of processes with network invariants. *Proceedings of the 1989 Internatinoal Workshop on Automatic Verification Methods for Finite State Systems.*