Formal Method & Model Checking

2301591 Individual Study

Supakit Sroynam

PREFACE

This report has been written to fulfill the requirements of the 2301591 Individual study course. The report is ultimately based on the formal methods' presentation slide from Roberto Sebastiani, DISI, Università di Trento, Italy and Handbook of model checking. This course is privately lectured by Associate Professor Krung Sinapiromsaran.

I would like to thank my supervisor, Associate Professor Krung Sinapiromsaran, for the excellent guidance and support during the process.

Supakit Sroynam
20 April 2025

CONTENTS

Preface **2**

Contents **3**

List of Figures **5**

1 Introduction **6**

- 1 Motivation . . . 6
- 2 Motivating Examples . . . 7
  - 2.1 Therac-25 . . . 7
  - 2.2 Ariane flight V88 . . . 7
  - 2.3 Pentium FDIV bug . . . 7
  - 2.4 The 1990 AT&T blackout . . . 8
- 3 Problems with traditional methods . . . 8
- 4 Formal Verification . . . 9
  - 4.1 Early Foundations of Formal Verification . . . 10
  - 4.2 Axiomatic Verification of Sequential Programs . . . 10
  - 4.3 Rise of Formal Methods in Modern Systems . . . 10
- 5 Benefits and Limitations of Formal Verification . . . 11
- 6 Formal Verification Methods . . . 12
  - 6.1 Theorem proving . . . 13
  - 6.2 Model checking . . . 14

2 Transition Systems **16**

- 1 Kripke models . . . 16
- 2 Composing Kripke Models . . . 17
  - 2.1 Asynchronous Composition . . . 17
  - 2.2 Synchronous Composition . . . 19
- 3 Properties of transition systems . . . 21

3 Temporal Logics **22**

- 1 Equivalence and Equi-satisfiability . . . 22
- 2 Negative Normal Form (NNF) . . . 23
  - 2.1 Reduction to NNF . . . 23
  - 2.2 Properties of the Reduction . . . 24
- 3 Conjunctive Normal Form (CNF) . . . 24
  - 3.1 Reduction to CNF . . . 25
- 4 Computation Tree vs Paths . . . 25
- 5 Linear Temporal Logic (LTL) . . . 26

5.1 Syntax 27
5.2 Semantics 27
5.3 LTL Model Checking 27
6 Computation Tree Logic (CTL) 28
6.1 Syntax 28
6.2 Semantics 29
6.3 CTL Model Checking 30
7 LTL vs. CTL 31
8 Computation Tree Logic (CTL*) 31
8.1 Syntax 32
8.2 Semantics 32
8.3 Extension with Nested Path Quantifiers 33

4 Model Checking 35
1 CTL Model Checking 35
1.1 Model checking algorithm 36
1.2 Example 36
2 Safety Property 40
3 Liveness Property 40
4 Fairness Property 40
5 Ordered Binary Decision Diagrams (OBDDs) 41
5.1 Properties of OBDDs 42
5.2 Example 42

5 SAT-Based Model Checking 47
1 Techniques for SAT 47
1.1 Resolution 47
1.2 DPLL 50
1.3 CDCL 52
2 Bounded Model Checking 55
2.1 Iterative Search for Increasing Length k 55
3 K-induction 55
3.1 Base Case Check 56
3.2 Inductive Step Check 56

Bibliography 57

LIST OF FIGURES

1.1 Standard Development Process. 8
1.2 Software life-cycle and error introduction, detection, and repair costs. 9
1.3 Model Checking Process. 14

2.1 Examples of Kripke models. 17
2.2 A Kripke model of two-process Mutual Exclusion. 17
2.3 An example of Asynchronous Composition. 18
2.4 An examples of Asynchronous Product. 19
2.5 An examples of Synchronous Composition. 19
2.6 An examples of Synchronous Product. 20

3.1 A Kripke model of basic done process. 26
3.2 Computational paths and a computational tree. 26
3.3 Intuitions Behind LTL Semantics. 28
3.4 Intuitions Behind CTL Semantics. 30
3.5 Venn Diagram of CTL and LTL 31
3.6 Venn Diagram of CTL, LTL, and CTL* 34

4.1 An example of CTL model checking. 35
4.2 The first step of model checking. 37
4.3 The second step of model checking. 38
4.4 The 1st step of OBDDs construction. 43
4.5 The 2nd step of OBDDs construction. 43
4.6 The 3rd step of OBDDs construction. 44
4.7 The 4th step of OBDDs construction. 44
4.8 The 5th step of OBDDs construction 45
4.9 The 6th step of OBDDs construction 45
4.10 The 7th step of OBDDs construction. 46
4.11 The 8th step of OBDDs construction. 46

CHAPTER 1

INTRODUCTION

1 Motivation

Formal verification and model checking are techniques used in computer science and engineering to prove the correctness of systems.

Problems in industrial system development tend to emerge as size and complexity increase. That creates problems like testing cost explosion and quality dropping.

Definition 1.1: Desired properties of systems

- **Availability:** A system must be working and able to provide its service.
- **Reliability:** A system must correctly provide its functionality, as expected by users.
- **Safety:** A system must do nothing which is very undesire, like harming people.
- **Security:** A system must resist intruders.

The four core properties—availability, reliability, safety, and security—serve as the foundation for assessing and designing critical systems. These properties determine how well system functions and the consequences of failure, with each property emphasizing a different aspect of the system's operation.

Definition 1.2: Types of critical systems

- **Safety critical:** A system whose failure can cause life losses or serious environmental damage. (Ex. Train & plane controllers, Nuclear power plants controller)
- **Mission critical:** A system whose failure can cause the failure of the goal of an important mission. (Ex. Spacecraft controllers)
- **Financial critical:** A system whose failure can cause the loss of huge amounts of money. (Ex. Bank management software, operating systems)

2 Motivating Examples

2.1 Therac-25

The Therac-25 was a computer-controlled radiation therapy machine developed by Atomic Energy of Canada Limited (AECL) in 1982, following the Therac-6 and Therac-20 models.

Between 1985 and 1987, the Therac-25 was involved in at least six accidents, where some patients were exposed to radiation doses far exceeding safe levels.

Due to concurrent programming errors (known as race conditions), the machine occasionally delivered radiation doses hundreds of times higher than intended, causing serious injury or death. These incidents highlighted the significant risks associated with using software to control safety-critical systems.

2.2 Ariane flight V88

Ariane Flight V88 was the unsuccessful maiden flight of the Arianespace Ariane 5 rocket, vehicle no. 501, on June 4, 1996.

The rocket was carrying the Cluster spacecraft, a set of four research satellites from the European Space Agency. The mission failed due to several software design flaws, including dead code meant for the Ariane 4 rocket and insufficient safeguards against integer overflow.

This led to an exception being mishandled, which caused the inertial navigation system, otherwise functioning properly, to shut down. As a result, the rocket veered off its intended path just 37 seconds after launch, began breaking apart under high aerodynamic forces, and ultimately self-destructed through its automated flight termination system.

This failure has become one of the most well-known and costly software errors in history, resulting in a loss of over US$370 million.

2.3 Pentium FDIV bug

The Pentium FDIV bug was a hardware flaw in the floating-point unit (FPU) of early Intel Pentium processors, which caused the processor to return incorrect results when dividing certain high-precision numbers.

This issue arose due to missing values in a lookup table used by the FPU's floating-point division algorithm, leading to small errors in calculations. Although these errors typically occurred infrequently and resulted in minor deviations from the correct output, in specific circumstances, they could occur more often and cause significant discrepancies. The severity of the FDIV bug remains a subject of debate. While it was estimated that only 1 in 9 billion floating-point divisions would produce incorrect results, the flaw, along with Intel's initial response, faced considerable criticism from the technology community.

In December 1994, Intel initiated a recall of the affected processors, marking the first instance of a full recall of a computer chip. In its 1994 annual report, Intel acknowledged a $475 million pre-tax charge to cover the replacement and write-off of these processors.

2.4 The 1990 AT&T blackout

The 1990 AT&T blackout was a significant telecommunications outage that occurred on January 15, 1990, disrupting AT&T's long-distance network across the United States for approximately nine hours.

The failure was caused by a software error during a routine upgrade to AT&T's tandem switches, which are responsible for routing long-distance calls. A flaw in the software resulted in the malfunction of the network's database management system, leading to a cascading failure that overwhelmed the system's ability to process and route calls.

As a result, millions of long-distance calls were either dropped or could not be completed, severely affecting both personal and business communications, including emergency services. Although local telephone services were largely unaffected, the outage highlighted the vulnerabilities of mission-critical systems relying on complex software.

3 Problems with traditional methods

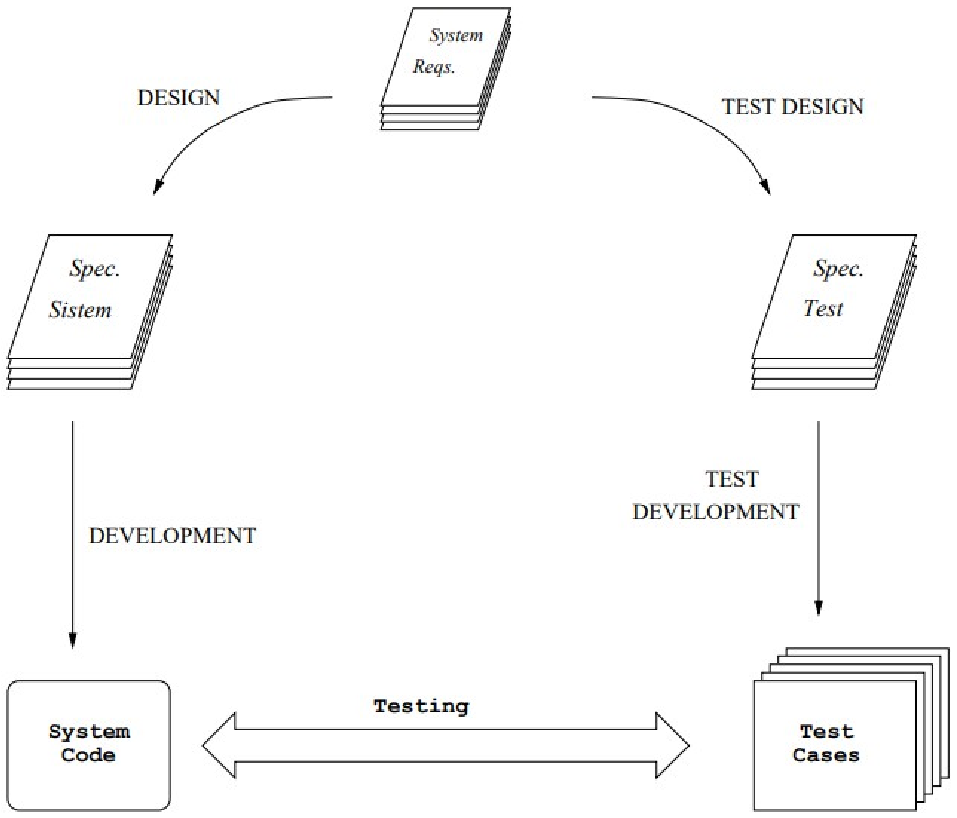

Figure 1.1: Standard Development Process.

Ambiguous Specifications: Inadequate clarity in the requirements, analysis, and design stages can lead to ambiguity in the system's objectives and expectations. This ambiguity can have a profound impact on subsequent phases of development.

Errors in Specifications/Design Refinements: Mistakes in the refinement of specifications and design decisions can propagate throughout the software development life-cycle, often leading to the implementation of incorrect or inefficient solutions.

Limited Test Coverage: Insufficient test coverage can result in the failure to detect issues early in the development process, contributing to defects that are harder to identify and resolve in later stages.

Consequences

- **Expensive Errors in Early Design Phases:** Errors introduced during early design and specification stages can be costly to correct, especially when discovered at later stages of development.
- **Low Software Quality and Maintenance Challenges:** Insufficient attention to detail in early phases leads to low software quality, making future maintenance and enhancements more difficult and error-prone.
- **Infeasibility of Achieving Ultra-High Reliability Requirements:** Due to initial design flaws and inadequate specifications, achieving the desired level of reliability, particularly in complex systems, becomes infeasible.

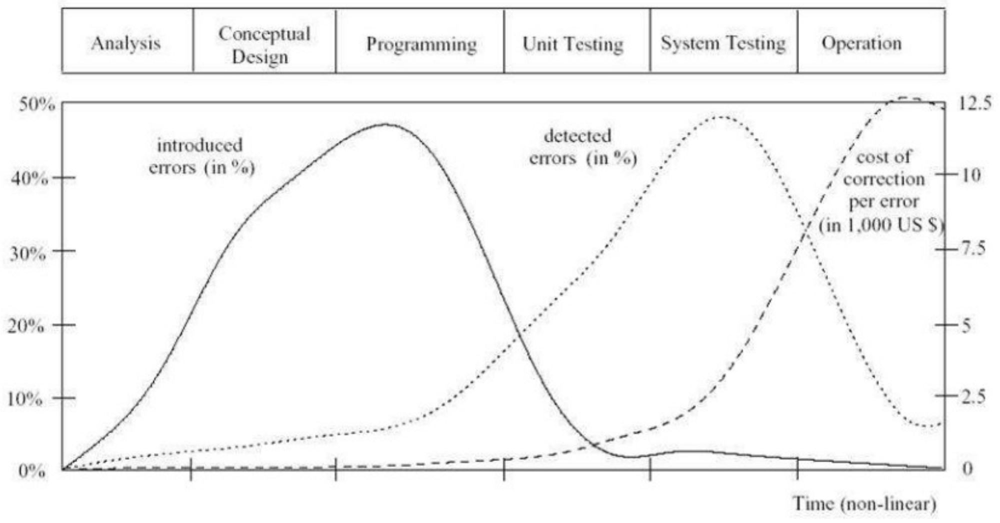

Figure 1.2: Software life-cycle and error introduction, detection, and repair costs.

4 Formal Verification

Formal verification, the use of mathematical techniques to verify the correctness of systems, has a rich history in the field of software engineering. The evolution of formal verification is closely tied to the development of logical assertions, program specifications, and the mathematical foundations of correctness. Below is an overview of the significant milestones in this field.

4.1 Early Foundations of Formal Verification

The conceptual foundations of formal verification can be traced back to the late 1960s, when early pioneers in the field sought to apply logical reasoning to the correctness of computer programs.

- In 1967, *Robert W. Floyd* introduced the concept of *logical assertions* in his seminal work on program verification. Floyd's method, which became known as *Floyd's method*, used assertions placed within the code to formally prove the correctness of a program with respect to its specification.

- Concurrently, *Edsger Dijkstra* made significant contributions to the field, particularly through his development of the *structured programming* paradigm, which emphasized the importance of clarity in program design. Dijkstra's work laid the groundwork for later formal methods by stressing the need for mathematical reasoning in program correctness.

4.2 Axiomatic Verification of Sequential Programs

In 1969, *C.A.R. Hoare* introduced the concept of *axiomatic verification*, which provided a formal basis for reasoning about the correctness of sequential programs. Hoare's method, often referred to as *Hoare Logic*, used a set of formal rules and logical assertions (preconditions and postconditions) to verify that a program correctly satisfies its specification.

- Hoare's axiomatic system was designed to verify the correctness of programs by defining *Hoare triples*, which describe the relationship between the state of the program before and after its execution. This system significantly advanced the formal verification of sequential programs.

- During the 1970s and 1980s, the scope of axiomatic verification was extended to handle *concurrent programs*. The complexity of verifying systems with multiple interacting processes, each executing in parallel, presented new challenges. Nevertheless, axiomatic verification methods were adapted to address these challenges, though manual proofs were often required and the proofs themselves could be labor-intensive.

- Despite the power of axiomatic verification, its widespread acceptance was limited due to the high cost and complexity of constructing manual proofs. Furthermore, while machine assistance was available in some cases, fully automated verification remained elusive.

4.3 Rise of Formal Methods in Modern Systems

In recent decades, formal methods have gained greater traction, particularly in the verification of *reactive* and *concurrent systems*, which are common in modern software and hardware design. Reactive systems, such as embedded systems and control systems, often interact with the environment and require rigorous validation to ensure safety and reliability.

- One of the most significant developments in this area has been the advent of *model checking*, a highly automated, algorithmic technique for verifying the correctness of reactive systems. Model checking involves systematically exploring all possible states of a system to verify that it satisfies its specification, ensuring properties such as safety and liveness.

- The introduction of model checking has revolutionized the field of formal verification, making it feasible to automatically verify complex systems with numerous interacting components. Model checking has found applications in safety-critical systems, such as aerospace and automotive systems, where correctness is paramount.

- Despite the growing adoption of model checking and other formal methods, challenges remain in achieving full verification, especially in systems with large state spaces or complex real-time requirements.

5 Benefits and Limitations of Formal Verification

Benefits of Formal Verification

- **Rigorous Assurance of Correctness:** Formal verification provides mathematical proof that a system satisfies its specification, eliminating the possibility of certain types of bugs and errors that may arise during traditional testing.

- **Increased Reliability:** By proving that a system adheres to its specification, formal verification ensures that the system behaves as intended under all possible conditions, making it highly reliable. This is crucial in systems such as aerospace, automotive, and medical devices, where failure is not an option.

- **Early Detection of Errors:** Formal verification can be applied early in the development process, identifying errors in the design or specification stages before they manifest as defects in the implementation. This reduces the cost of fixing errors later in the development lifecycle.

- **Automated Verification for Complex Systems:** Modern formal verification techniques, such as model checking, allow for the automated verification of large and complex systems. This reduces the reliance on manual proofs and makes the process more scalable.

- **Proving Properties Beyond Testing:** Unlike traditional testing, which can only verify a system's behavior under a limited set of test cases, formal verification guarantees that the system will behave correctly for all possible inputs and states. This provides a higher level of assurance than conventional testing approaches.

Limitations of Formal Verification

- **Complexity of Formal Methods:** Formal verification methods, particularly for large and complex systems, can be difficult to apply and require a high level of expertise. The modeling and specification phases can be time-consuming and require specialized knowledge.
- **State Space Explosion:** Many formal verification techniques, such as model checking, suffer from the *state space explosion* problem, where the number of states grows exponentially as the system's complexity increases. This can make verification of large systems infeasible due to the computational resources required.
- **Difficulty in Modeling:** Accurately modeling the system behavior for formal verification can be challenging, particularly for systems with complex or highly dynamic behavior. The correctness of the verification is heavily dependent on the accuracy and completeness of the model.
- **Limited Applicability to Real-Time and Continuous Systems:** While formal verification is highly effective for discrete systems, its applicability to real-time, continuous, or hybrid systems is more limited. These systems often involve intricate timing and continuous state spaces that are difficult to model and verify formally.
- **High Computational Overhead:** The verification process, particularly for large systems, can be computationally expensive and time-consuming. The resources required to conduct formal verification may outweigh the benefits for smaller or less critical systems.
- **Lack of Support for All Programming Languages:** Formal verification tools often require specific programming languages or formal specification languages, limiting their applicability to all types of software. Adapting existing codebases to work with these tools can be cumbersome.

6 Formal Verification Methods

Once the system is formally specified, it must be validated and verified to ensure that it meets the desired requirements and behaves correctly.

- **Formal Validation:** This is the process of ensuring that the formal description of the system accurately represents the system's intended behavior and conforms to its requirements. Validation typically involves comparing the formal model to the original system requirements to ensure consistency.
- **Formal Verification:** Verification is the exhaustive comparison of the formal description of the system against the formal properties it must satisfy. This process checks

whether the system meets all of its desired specifications and ensures that it will behave correctly under all possible conditions. Verification is often achieved through methods such as model checking or theorem proving.

Two main technologies are used: **Theorem Proving** & **Model Checking**

6.1 Theorem proving

Theorem proving is a formal verification method where the system is modeled as a set of logical formulae, denoted as Γ, and the desired properties are expressed as theorems, denoted as Ψ. This method relies on precise, unambiguous semantics, ensuring that the logical statements are rigorously defined.

Theorem Proving

$$\Gamma \rightarrow \Psi$$

This implies that if the logical formulae Γ (which model the system) hold, then the properties Ψ (which represent the desired outcomes or behaviors) must also hold. In other words, the verification task becomes:

$$\vdash (\Gamma \rightarrow \Psi)$$

Limitations and Disadvantages of Theorem Proving

- **Hard to Mechanize:** Theorem proving is typically an interactive process, requiring human intervention at each step. This makes it difficult to automate and scale to large or complex systems.
- **Difficult Formalization:** The formalization of the system (Γ) can be very challenging and time-consuming. In many cases, capturing the system's behavior precisely with logical formulae is not straightforward.
- **Requires Expertise:** Using a theorem prover effectively requires a high level of expertise in both formal methods and the specific theorem proving tool. This limits its accessibility to those with specialized knowledge.
- **Limited Applicability:** Many verification problems are beyond the capabilities of current theorem provers. As a result, this method is not always feasible for larger, more complex systems.

6.2 Model checking

Model checking verifies whether the system model M satisfies the formal property Ψ, and it does so by examining all possible system states.

Model checking

$$M \models \Psi$$

- If $M \models \Psi$ holds true, then the system satisfies the property Ψ, meaning the property is verified.
- If $M \models \Psi$ does not hold, then a counterexample is generated, representing a scenario (execution trace) that leads to a failure or bug in the system.

The counterexample is a valuable result because it provides an explicit execution path showing where the system deviates from its expected behavior, making it easier to identify and fix the issue.

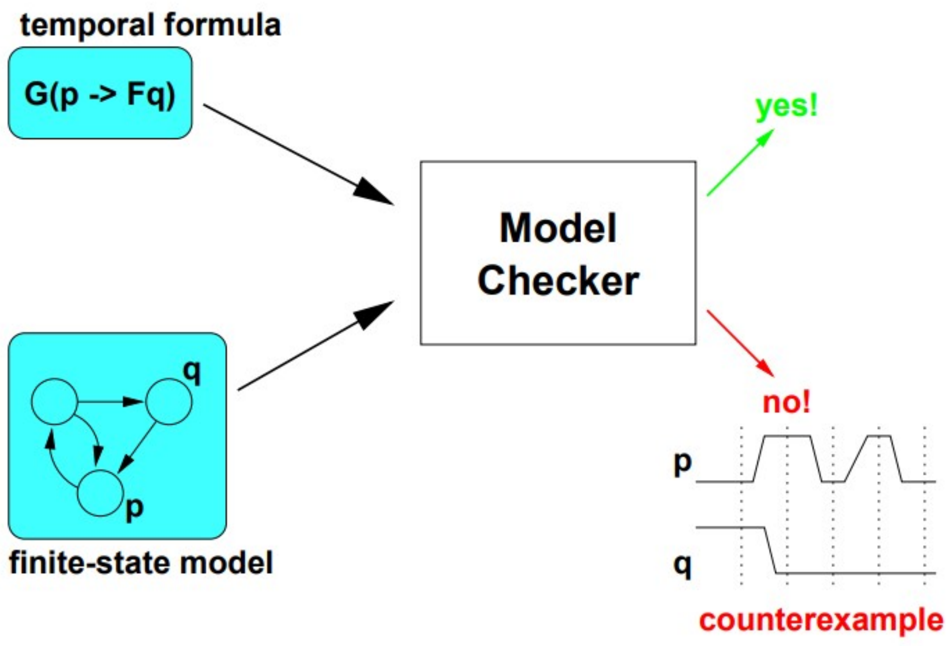

Figure 1.3: Model Checking Process.

Limitations and Disadvantages of Model Checking

- **Limited to Finite State Machines:** Model checking is most effective when the system can be represented as a finite state machine (FSM). For systems with infinite or highly complex state spaces, the approach becomes less applicable.
- **Difficulty with Temporal Logic:** Engineers often find temporal logic formulas challenging to work with due to their abstract nature. While encodings can be provided, the need for expertise in temporal logic remains a barrier to widespread adoption.
- **State Space Explosion:** The state space of a system grows exponentially with the number of interacting components. This phenomenon, known as state space explosion, makes it computationally expensive and often infeasible to check large systems.

CHAPTER 2

TRANSITION SYSTEMS

1 Kripke models

Kripke models are formal tools used to describe **reactive systems**—systems that do not terminate and have infinite behaviors. These systems include a variety of applications such as **communication protocols** and **hardware circuits**, which exhibit ongoing interactions without a fixed endpoint.

A Kripke model represents the dynamic evolution of a modeled system by capturing the various states and transitions that the system can undergo. In such models, a **state** is defined as a combination of several components, including:

Definition 1.1: Kripke model

$$M =< S, I, R, AP, L >$$

- M : a Kripke model.
- AP : a set of atomic propositions(boolean variable).
- S : a finite set of states.
- $I \subset S$: a set of initial states.
- $R \subset S \times S$: a transition relation such that R is left-total
- L : a labeling $L : S \to 2^{AP}$.

Definition 1.2: Path

$$\pi = s_0, s_1, s_2, ... \in S^\omega;\ s_0 \in I \text{ and } (s_i, s_{i+1}) \in R$$

A state s is **reachable** in M if there is a path from the initial states to s

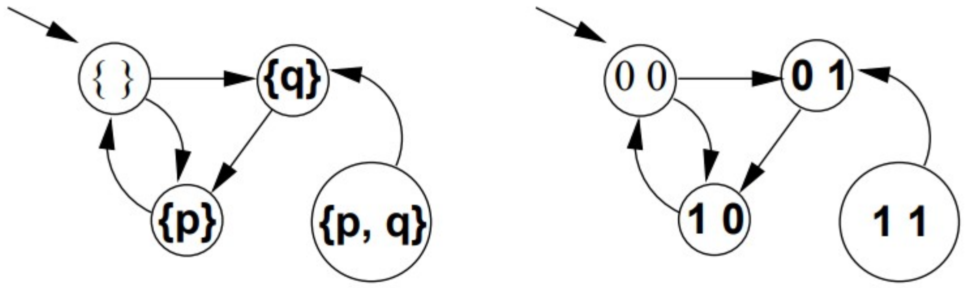

Figure 2.1: Examples of Kripke models.

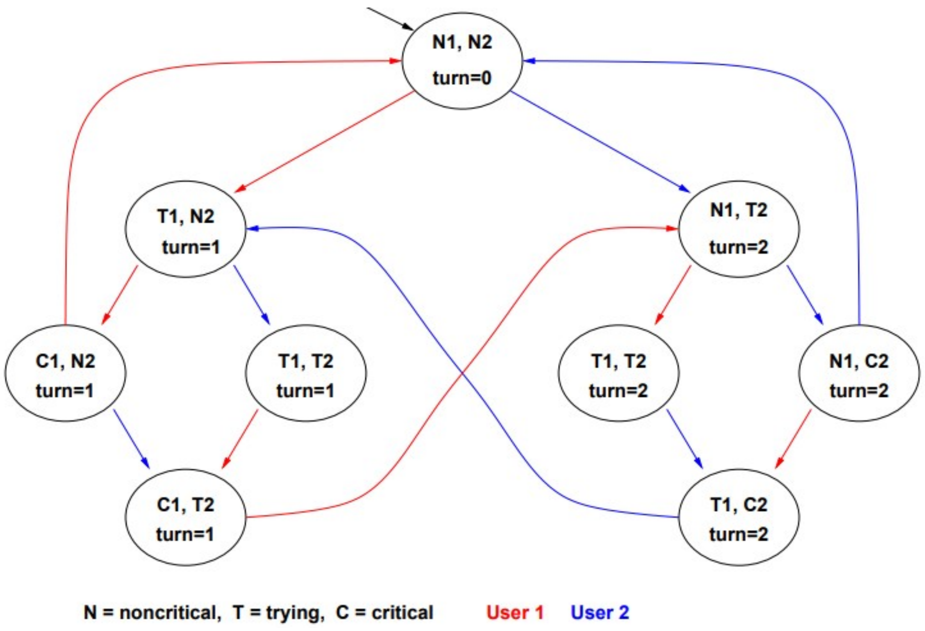

Figure 2.2: A Kripke model of two-process Mutual Exclusion.

From Figure 2.2, a state $(C1, T2)$ is **reachable** because there is a path π from $(N1, N2) \in I$ to $(C1, T2).\pi = (N1, N2), (T1, N2), (T1, T2), (C1, T2)$.

2 Composing Kripke Models

Complex Kripke Models are obtained by composition of smaller ones.

Components can be combined by **asynchronous** composition, and **synchronous** composition.

2.1 Asynchronous Composition

Given 2 models, transitions in both models are independent to each other. At each time instant, one component is selected to perform a transition. For example, a path $\pi = (0, a), (0, b), (1, b), (1, a)$ for asynchronous transition in Figure 2.3.

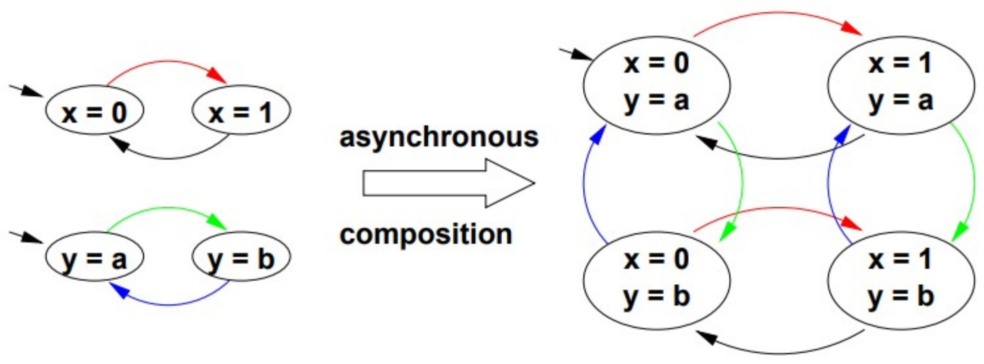

Figure 2.3: An example of Asynchronous Composition.

Definition 2.1: Asynchronous Product of Kripke Models

Let M_1 and M_2 be Kripke models defined as follows:

$$M_1 = \langle S_1, I_1, R_1, AP_1, L_1 \rangle, \quad M_2 = \langle S_2, I_2, R_2, AP_2, L_2 \rangle$$

Then the asynchronous product M of M_1 and M_2, denoted as $M = M_1 \parallel M_2$, is defined as:

$$M = \langle S, I, R, AP, L \rangle$$

where:

- $S \subseteq S_1 \times S_2$ such that $\forall \langle s_1, s_2 \rangle \in S$, $\forall l \in AP_1 \cap AP_2$, we have $l \in L_1(s_1)$ if and only if $l \in L_2(s_2)$,
- $I \subseteq I_1 \times I_2$ such that $I \subseteq S$,
- $R(\langle s_1, s_2 \rangle, \langle t_1, t_2 \rangle)$ if and only if $(R_1(s_1, t_1)$ and $s_2 = t_2)$ **or** $(s_1 = t_1$ and $R_2(s_2, t_2))$
- $AP = AP_1 \cup AP_2$
- $L : S \to 2^{AP}$ such that $L(\langle s_1, s_2 \rangle) = L_1(s_1) \cup L_2(s_2)$.

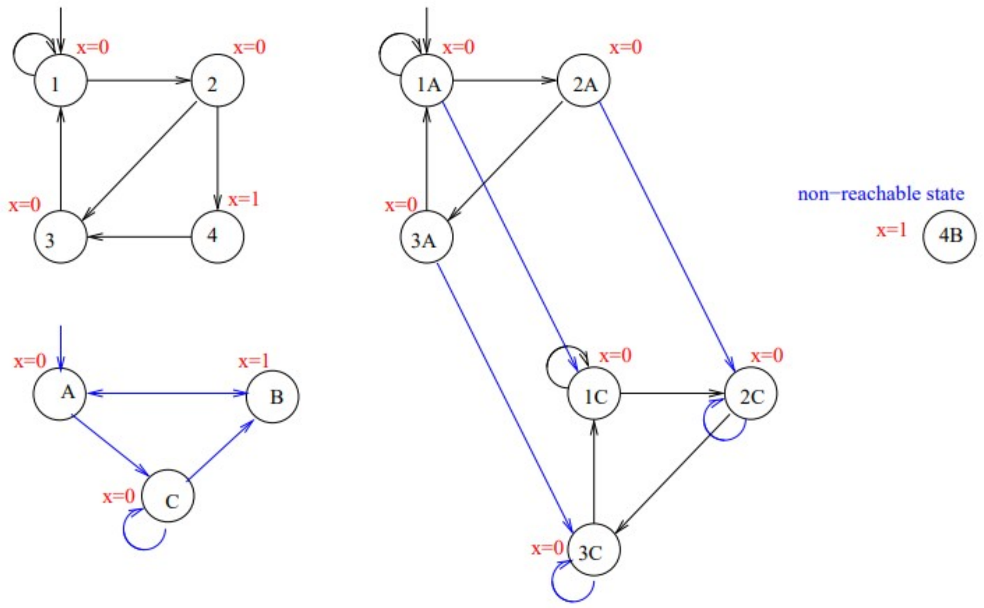

Figure 2.4: An examples of Asynchronous Product.

2.2 Synchronous Composition

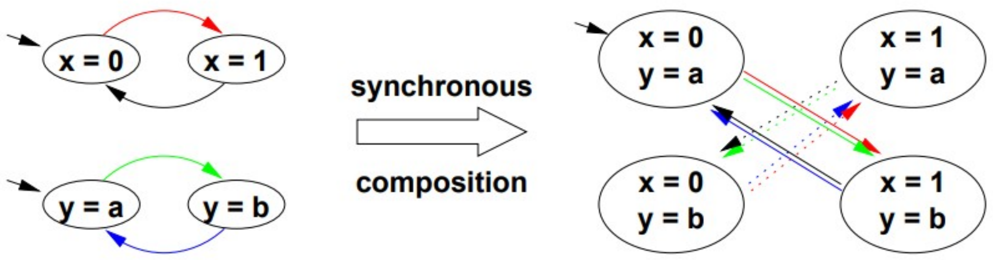

Figure 2.5: An examples of Synchronous Composition.

Given 2 models, transitions in both models are dependent to each other. At each time instant, every component performs a transition. For example, a path $\pi = (0, a), (1, b), (0, a), (1, b)$ for asynchronous transition in Figure 2.5. Moreover, there are no path that reach states $(0, b)$ and $(1, a)$.

Definition 2.2: Synchronous Product of Kripke Models

Let M_1 and M_2 be Kripke models defined as follows:

$$M_1 = \langle S_1, I_1, R_1, AP_1, L_1 \rangle, \quad M_2 = \langle S_2, I_2, R_2, AP_2, L_2 \rangle$$

Then the synchronous product M of M_1 and M_2, denoted as $M = M_1 \times M_2$, is defined as:

$$M = \langle S, I, R, AP, L \rangle$$

where:

- $S \subseteq S_1 \times S_2$ such that for $\forall \langle s_1, s_2 \rangle \in S$, $\forall l \in AP_1 \cap AP_2$, $l \in L_1(s_1)$ if and only if $l \in L_2(s_2)$,
- $I \subseteq I_1 \times I_2$ such that $I \subseteq S$,
- $R(\langle s_1, s_2 \rangle, \langle t_1, t_2 \rangle)$ if and only if $R_1(s_1, t_1)$ **and** $R_2(s_2, t_2)$,
- $AP = AP_1 \cup AP_2$,
- $L : S \rightarrow 2^{AP}$ such that $L(\langle s_1, s_2 \rangle) = L_1(s_1) \cup L_2(s_2)$.

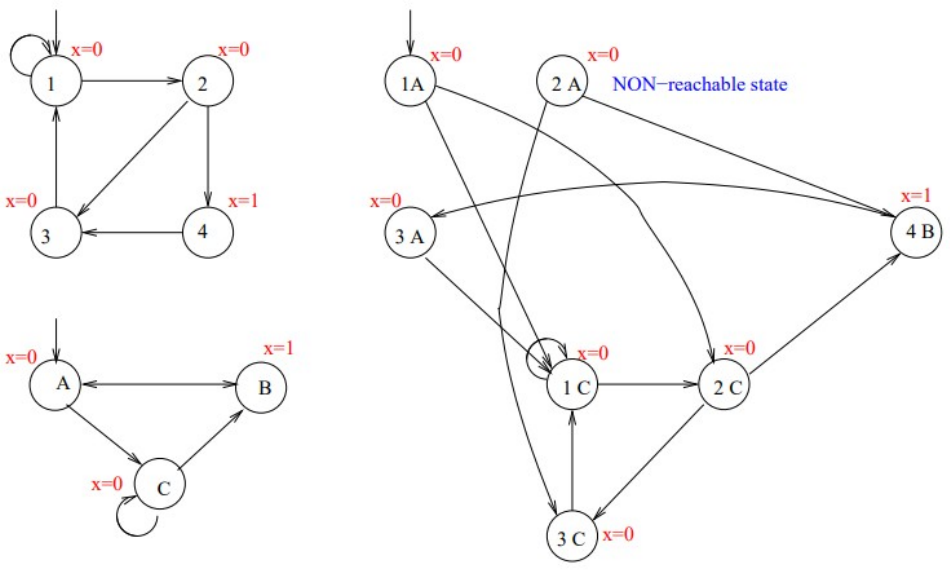

Figure 2.6: An examples of Synchronous Product.

3 Properties of transition systems

Safety properties

Safety properties guarantee that something "bad" will never happen.

Safety property can be refuted by a finite execution trace that demonstrates a violation. This means that if we can find a **finite counterexample** — a finite sequence of states or actions — that leads to an undesirable state, we have disproven (refuted) the safety property.

Example: In a concurrent system, a safety property might assert that two processes should never be in the critical state at the same time. If at any point in time both processes enter the critical state, the property is violated. Like in Figure 2.2, there are no reachable states that are labeled with (C, C).

Liveness properties

Liveness property ensures something "good" eventually happens.

Liveness property can be refuted by infinite behavior because the system might get stuck in an infinite loop without reaching the desired state. This **infinite behavior** violates the liveness property because it ensures that the desired outcome will never occur, even though the system continues running indefinitely.

Example: In a mutex system, two processes, User 1 ,and User 2 share a critical state. The system ensures that only one user can enter the critical state at any given time, and once a process finishes executing in the critical state, the other process can enter. A liveness property ensures that every process requesting access to the critical state will eventually get it. Like in Figure 2.2, every trying state (T in label) must has a path that following by a critical state (C in label).

CHAPTER 3

TEMPORAL LOGICS

1 Equivalence and Equi-satisfiability

Equivalence:

Two formulas φ_1 and φ_2 are equivalent if, for every interpretation μ, the following holds:

$$\mu \models \varphi_1 \iff \mu \models \varphi_2.$$

Equi-satisfiability:

Two formulas φ_1 and φ_2 are equi-satisfiable if there exist interpretations μ_1 and μ_2 such that:

$$\exists \mu_1 (\mu_1 \models \varphi_1) \iff \exists \mu_2 (\mu_2 \models \varphi_2).$$

Equivalent and Equi-satisfiable

$$\varphi_1, \varphi_2 \text{ are equivalent} \Rightarrow \varphi_1, \varphi_2 \text{ are equi-satisfiable}$$

$$\varphi_1, \varphi_2 \text{ are equivalent} \not\Leftarrow \varphi_1, \varphi_2 \text{ are equi-satisfiable}$$

Example : Let the formulas φ_1 and φ_2 be defined as follows:

$$\varphi_1 = \psi_1 \vee \psi_2,$$

$$\varphi_2 = (\psi_1 \vee \neg A_3) \wedge (A_3 \vee \psi_2),$$

where A_3 does not appear in $\psi_1 \vee \psi_2$.

We will show that φ_1 and φ_2 are **equi-satisfiable** but not **equivalent**.

Equi-satisfiability:

There exists a model μ such that:

$$\mu \models (\psi_1 \vee \neg A_3) \wedge (A_3 \vee \psi_2) \implies \mu \models \psi_1 \vee \psi_2.$$

This means that if a model satisfies φ_2, it also satisfies φ_1.

Additionally, there exists a model μ_0 such that:

$$\mu_0 \models \psi_1 \vee \psi_2 \implies \mu_0 \wedge A_3 \models (\psi_1 \vee \neg A_3) \wedge (A_3 \vee \psi_2) \quad \text{or} \quad \mu_0 \wedge \neg A_3 \models (\psi_1 \vee \neg A_3) \wedge (A_3 \vee \psi_2),$$

which shows that φ_1 and φ_2 are equi-satisfiable.

Non-equivalence:

For non-equivalence, we can show that there exists a model μ_0 such that:

$$\mu_0 \not\models \psi_1 \quad \text{and} \quad \mu_0 \models \psi_2 \implies \mu_0 \wedge A_3 \models \psi_1 \vee \psi_2,$$

but

$$\mu_0 \wedge A_3 \not\models (\psi_1 \vee \neg A_3) \wedge (A_3 \vee \psi_2),$$

indicating that φ_1 and φ_2 are not equivalent.

2 Negative Normal Form (NNF)

A formula φ is in Negative Normal Form (NNF) if it is built only using the logical connectives $\wedge$ (AND), $\vee$ (OR), and negation $\neg$, applied recursively to literals. A literal is either a propositional variable or the negation of a propositional variable.

2.1 Reduction to NNF

Any formula φ can be reduced to NNF by performing the following steps:

1. **Substituting Implications and Biconditionals:**

   (a) Replace any implication $\varphi_1 \to \varphi_2$ with its equivalent disjunction:

   $$\varphi_1 \to \varphi_2 \quad \implies \quad \neg\varphi_1 \vee \varphi_2.$$

   (b) Replace any biconditional $\varphi_1 \leftrightarrow \varphi_2$ with its equivalent conjunction of disjunctions:

   $$\varphi_1 \leftrightarrow \varphi_2 \quad \implies \quad (\neg\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \neg\varphi_2).$$

2. **Pushing Down Negations Recursively:**

   (a) Apply De Morgan's law to the negation of a conjunction:

   $$\neg(\varphi_1 \wedge \varphi_2) \quad \implies \quad \neg\varphi_1 \vee \neg\varphi_2.$$

   (b) Apply De Morgan's law to the negation of a disjunction:

   $$\neg(\varphi_1 \vee \varphi_2) \quad \implies \quad \neg\varphi_1 \wedge \neg\varphi_2.$$

   (c) Eliminate double negations:

   $$\neg\neg\varphi_1 \quad \implies \quad \varphi_1.$$

2.2 Properties of the Reduction

- The reduction process is **linear** if a Directed Acyclic Graph (DAG) representation of the formula is used.
- The reduction preserves the **equivalence** of the formulas, meaning that the formula after the reduction is logically equivalent to the original formula.

Example:
Consider the formula:

$$\begin{aligned} & (A_1 \leftrightarrow A_2) \leftrightarrow A_3 \\ \implies \quad & ((A_1 \leftrightarrow A_2) \rightarrow A_3) \wedge ((A_1 \leftrightarrow A_2) \leftarrow A_3) \\ \implies \quad & (((A_1 \leftarrow A_2) \wedge (A_1 \rightarrow A_2)) \rightarrow A_3) \wedge (((A_1 \leftarrow A_2) \wedge (A_1 \rightarrow A_2)) \leftarrow A_3) \\ \implies \quad & (\neg((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee A_3) \wedge (((\neg A_1 \vee A_2) \wedge (A_1 \vee \neg A_2)) \vee \neg A_3) \end{aligned}$$

3 Conjunctive Normal Form (CNF)

A formula is in Conjunctive Normal Form (CNF) if it is a conjunction of clauses, where each clause is a disjunction of literals. A literal is either a propositional variable or its negation.

Definition 3.1: Conjunctive Normal Form (CNF)

Let $\mathcal{L}$ be a set of propositional variables. The CNF formula φ is formally defined as:

$$\varphi = \bigwedge_{i=1}^{n} C_i = \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m_i} L_{ij} \right)$$

where:

- φ is a formula in CNF.
- $\wedge$ represents the logical AND (conjunction).
- $\vee$ represents the logical OR (disjunction).
- $L_{ij} \in \mathcal{L} \cup \{\neg \mathcal{L}\}$ are literals (either a propositional variable or its negation).
- Each C_i is a clause (a disjunction of literals).
- n is the number of clauses.
- m_i is the number of literals in the i-th clause.

3.1 Reduction to CNF

1. **Substituting Implications and Biconditionals:**

   (a) Replace any implication $\varphi_1 \to \varphi_2$ with its equivalent disjunction:

   $$\varphi_1 \to \varphi_2 \quad \Longrightarrow \quad \neg\varphi_1 \vee \varphi_2.$$

   (b) Replace any biconditional $\varphi_1 \leftrightarrow \varphi_2$ with its equivalent conjunction of disjunctions:

   $$\varphi_1 \leftrightarrow \varphi_2 \quad \Longrightarrow \quad (\neg\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \neg\varphi_2).$$

2. **Pushing Down Negations Recursively:**

   (a) Apply De Morgan's law to the negation of a conjunction:

   $$\neg(\varphi_1 \wedge \varphi_2) \quad \Longrightarrow \quad \neg\varphi_1 \vee \neg\varphi_2.$$

   (b) Apply De Morgan's law to the negation of a disjunction:

   $$\neg(\varphi_1 \vee \varphi_2) \quad \Longrightarrow \quad \neg\varphi_1 \wedge \neg\varphi_2.$$

   (c) Eliminate double negations:

   $$\neg\neg\varphi_1 \quad \Longrightarrow \quad \varphi_1.$$

3. **Distribute OR over AND (Distributive Laws):**

   (a) Apply the distributive property:

   $$\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \Longrightarrow (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3).$$

Example:
Consider the formula:

$$(A \to B) \vee (C \wedge D)$$

$$\Longrightarrow \quad \neg A \vee B \vee (C \wedge D)$$

$$\Longrightarrow \quad (\neg A \vee B \vee C) \wedge (\neg A \vee B \vee D)$$

4 Computation Tree vs Paths

Consider the following Kripke structure:
From Figure 3.1, its execution can be seen as an infinite set of **computation paths** or an infinite **computation tree** in Figure 3.2

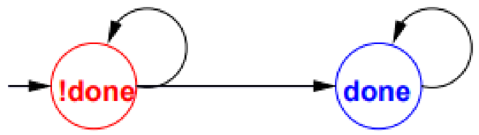

Figure 3.1: A Kripke model of basic done process.

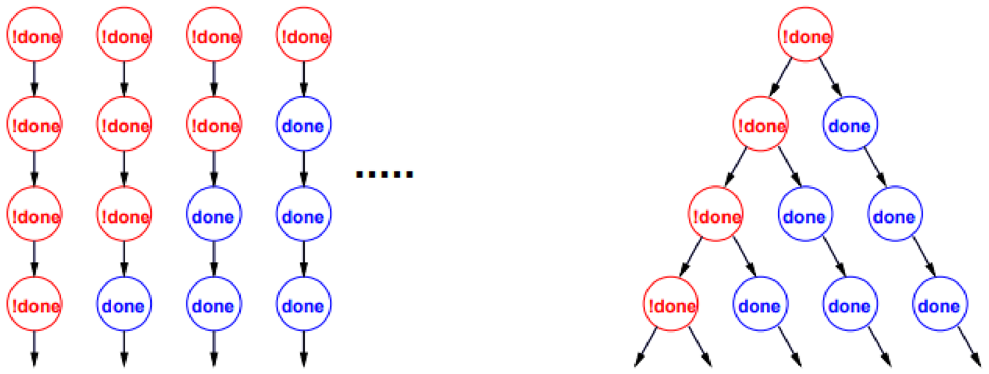

Figure 3.2: Computational paths and a computational tree.

With this execution, temporal logic can be categorized into two types:

1. Linear Temporal Logic (LTL)
    - Interpreted over each path of a Kripke structure.
    - Represents a linear model of time.
    - Utilizes temporal operators to describe properties along a single path.
    - Philosophical interpretation: "Since birth, one's destiny is set."
2. Computation Tree Logic (CTL)
    - Interpreted over the computation tree of a Kripke model.
    - Represents a branching model of time.
    - Utilizes temporal operators in conjunction with path quantifiers to express properties across multiple paths.
    - Philosophical interpretation: "One makes his/her own destiny step-by-step."

5 Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) is a modal logic that is used to describe properties of paths in a Kripke structure, where time is modeled as a linear sequence. The formal definition of LTL can be given as follows:

5.1 Syntax

The syntax of LTL includes the following components:

The syntax of LTL

- **Atomic propositions**: A set of atomic propositions $p_1, p_2, \ldots, p_n$.
- **Logical connectives**: $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), $\rightarrow$ (implication).
- **Temporal operators**:
  - $\mathbf{X}\phi$ (Next): ϕ holds in the next state.
  - $\mathbf{G}\phi$ (Globally): ϕ holds in all future states.
  - $\mathbf{F}\phi$ (Finally): ϕ holds at some future state.
  - $\phi\mathbf{U}\psi$ (Until): ϕ holds until ψ holds, and ψ must eventually hold.

5.2 Semantics

Given a path $\pi = s_0, s_1, s_2, \ldots$ in the Kripke structure, the satisfaction relation $\models$ for an LTL formula φ is defined inductively as follows:

- $s_0 \models p_i$ if and only if $p_i \in L(s_0)$, where p_i is an atomic proposition.
- $s_0 \models \neg\varphi$ if and only if $s_0 \not\models \varphi$.
- $s_0 \models \varphi \vee \psi$ if and only if $s_0 \models \varphi$ or $s_0 \models \psi$.
- $s_0 \models \varphi \wedge \psi$ if and only if $s_0 \models \varphi$ and $s_0 \models \psi$.
- $s_0 \models X\varphi$ if and only if $s_1 \models \varphi$, where s_1 is the successor of s_0.
- $s_0 \models G\varphi$ if and only if $s_0 \models \varphi$ and for all $i \geq 0$, $s_i \models \varphi$.
- $s_0 \models F\varphi$ if and only if there exists an $i \geq 0$ such that $s_i \models \varphi$.
- $s_0 \models \varphi\ U\ \psi$ if and only if there exists an $i \geq 0$ such that $s_i \models \psi$, and for all $j < i$, $s_j \models \varphi$.

5.3 LTL Model Checking

Let M be a Kripke structure and φ be an LTL formula. The LTL model checking problem can be expressed as follows:

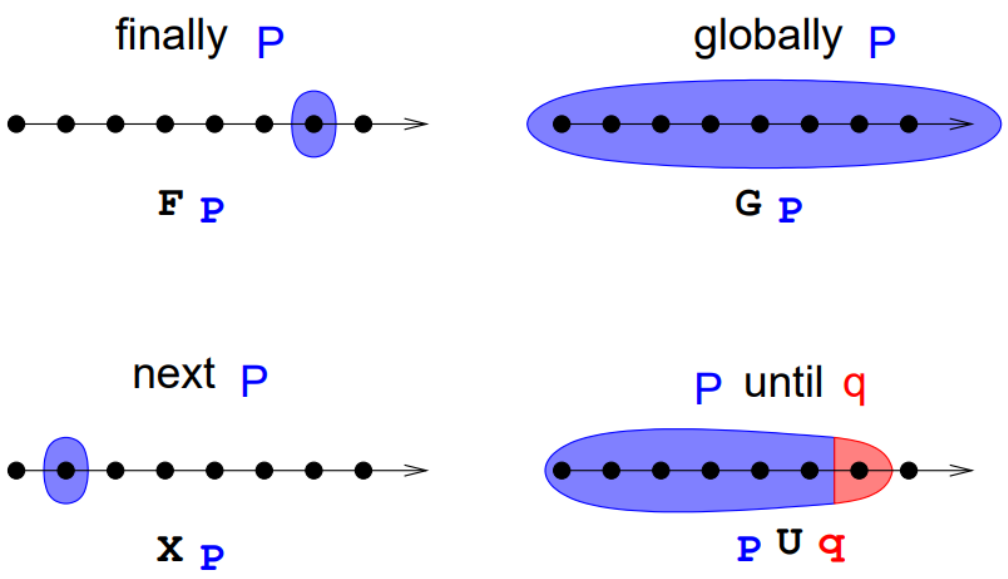

Figure 3.3: Intuitions Behind LTL Semantics.

Definition 5.1: LTL model checking

$$M \models \varphi \quad \Longleftrightarrow \quad \pi \models \varphi \text{ for } \textbf{every path } \pi \text{ in the Kripke structure } M.$$

It is crucial to note that the negation of satisfaction does not imply the satisfaction of the negation:

Important Remark

$$M \not\models \varphi \not\Rightarrow M \models \neg\varphi.$$

Example: Suppose φ is an LTL formula, and two paths π_1 and π_2 exist such that:

$$\pi_1 \models \varphi \quad \text{and} \quad \pi_2 \models \neg\varphi.$$

In this case, $M \not\models \varphi$ does not imply that $M \models \neg\varphi$.

6 Computation Tree Logic (CTL)

Computation Tree Logic (CTL) is a temporal logic used to describe properties of computation trees in Kripke structures, where time is modeled as a branching structure. The formal definition of CTL can be given as follows:

6.1 Syntax

The syntax of CTL includes the following components:

The syntax of CTL

- **Atomic propositions**: A set of atomic propositions $p_1, p_2, \ldots, p_n$.
- **Logical connectives**: $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), $\rightarrow$ (implication).
- **Temporal operators**:
  - $\mathbf{X}\phi$ (Next): ϕ holds in the next state.
  - $\mathbf{G}\phi$ (Globally): ϕ holds in all future states along a specific path.
  - $\mathbf{F}\phi$ (Finally): ϕ holds at some future state along a specific path.
  - $\phi\mathbf{U}\psi$ (Until): ϕ holds until ψ holds, and ψ must eventually hold along a path.
- **Path quantifiers**:
  - $\mathbf{A}\phi$ (For all paths): ϕ holds for all paths starting from the current state.
  - $\mathbf{E}\phi$ (There exists a path): ϕ holds for at least one path starting from the current state.

6.2 Semantics

Given a computation tree $\pi = s_0, s_1, s_2, \ldots$ in the Kripke structure, the satisfaction relation $\models$ for a CTL formula φ is defined inductively as follows:

- $s_0 \models p_i$ if and only if $p_i \in V(s_0)$, where p_i is an atomic proposition.
- $s_0 \models \neg\varphi$ if and only if $s_0 \not\models \varphi$.
- $s_0 \models \varphi \vee \psi$ if and only if $s_0 \models \varphi$ or $s_0 \models \psi$.
- $s_0 \models \varphi \wedge \psi$ if and only if $s_0 \models \varphi$ and $s_0 \models \psi$.
- $s_0 \models X\varphi$ if and only if there exists a successor state s_1 such that $s_1 \models \varphi$.
- $s_0 \models G\varphi$ if and only if for all states s_i, $s_i \models \varphi$.
- $s_0 \models F\varphi$ if and only if there exists a state s_i such that $s_i \models \varphi$.
- $s_0 \models \varphi\ U\ \psi$ if and only if there exists a state s_i such that $s_i \models \psi$, and for all $j < i$, $s_j \models \varphi$.
- $s_0 \models A\varphi$ if and only if for all paths π starting from s_0, $\pi \models \varphi$ at all states along the path.
- $s_0 \models E\varphi$ if and only if there exists a path π starting from s_0 such that $\pi \models \varphi$ at all states along the path.

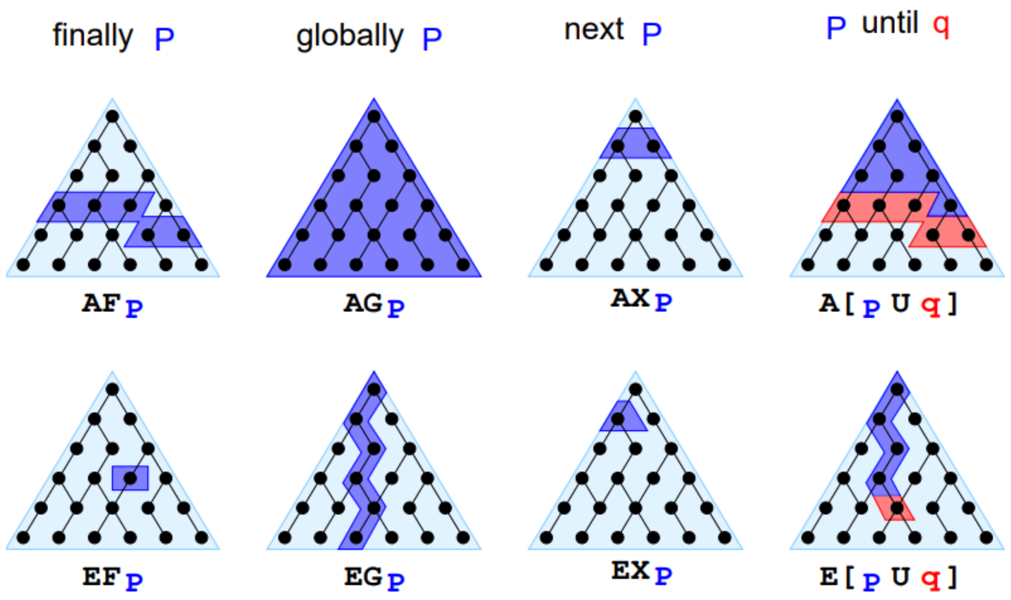

Figure 3.4: Intuitions Behind CTL Semantics.

6.3 CTL Model Checking

Let M be a Kripke structure and φ be a CTL formula. The CTL model checking problem is defined as follows:

Definition 6.1: CTL model checking

$M \models \varphi \quad \iff \quad M, s \models \varphi$ **for every initial state** $s \in I$ of the Kripke structure M.

It is important to note that the negation of satisfaction does not imply the satisfaction of the negation:

Important Remark

$$M \not\models \varphi \not\Rightarrow M \models \neg\varphi.$$

Example: Suppose φ is a universal CTL formula $A\varphi$, and two initial states s_0 and s_1 are such that:

$$M, s_0 \models \varphi \quad \text{and} \quad M, s_1 \not\models \varphi.$$

In this case, $M \not\models \varphi$ does not imply that $M \models \neg\varphi$.
Additionally, if the Kripke structure M has only one initial state s_0, then:

$$M \not\models \varphi \quad \Rightarrow \quad M \models \neg\varphi.$$

7 LTL vs. CTL

Many CTL formulas cannot be expressed in LTL (e.g., those containing existentially quantified subformulas):

$$\text{Example: } \mathbf{AG}(N_1 \to \mathbf{EF}T_1),\ \mathrm{AFAG}\,\varphi$$

In these cases, CTL involves temporal operators such as A (universal quantification) and E (existential quantification), which cannot be directly expressed in LTL, since LTL does not have path quantifiers.

Many LTL formulas cannot be expressed in CTL (e.g., fairness formulas in LTL):

$$\text{Example: } \mathbf{GF}T_1 \to \mathbf{GF}C_1,\ FG\varphi$$

These formulas rely on global conditions that LTL can express, but CTL requires explicit path quantification, which prevents it from directly expressing these types of properties.

However, some formulas can be expressed in both LTL and CTL, typically formulas with operators of nesting depth 1 and/or operators occurring positively:

$$\text{Example: } \mathbf{G}\neg(C_1 \wedge C_2),\ \mathbf{F}C_1,\ \mathbf{G}(T_1 \to \mathbf{F}C_1),\ \mathbf{GF}C_1$$

These are simpler formulas that do not require existential quantification or nested temporal operators, so they can be expressed in both logics.

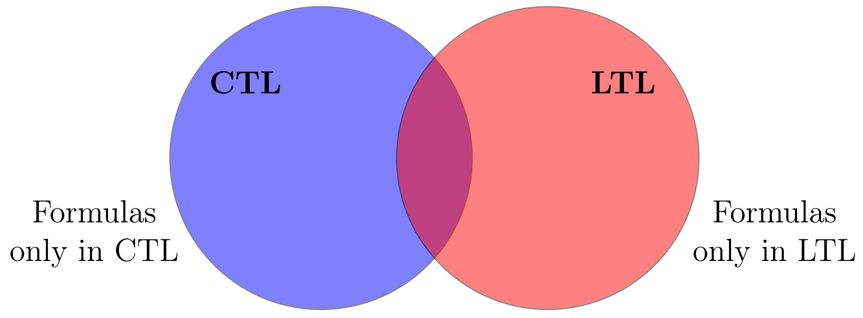

Figure 3.5: Venn Diagram of CTL and LTL

8 Computation Tree Logic (CTL*)

Computation Tree Logic (CTL*) is an extension of Computation Tree Logic (CTL) that allows more expressive temporal operators. Unlike CTL, which only allows path quantifiers (such as A and E) to apply to the entire formula, CTL* allows path quantifiers to be applied to subformulas, making it more flexible in expressing temporal properties. The formal definition of CTL can be given as follows:

8.1 Syntax

The syntax of CTL* includes the following components:

The syntax of CTL*

- **Atomic propositions**: A set of atomic propositions $p_1, p_2, \dots, p_n$.
- **Logical connectives**: $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), $\rightarrow$ (implication).
- **Temporal operators**:
  - $\mathbf{X}\phi$ (Next): ϕ holds in the next state.
  - $\mathbf{G}\phi$ (Globally): ϕ holds in all future states along a specific path.
  - $\mathbf{F}\phi$ (Finally): ϕ holds at some future state along a specific path.
  - $\phi\mathbf{U}\psi$ (Until): ϕ holds until ψ holds, and ψ must eventually hold along a path.
- **Path quantifiers**:
  - $\mathbf{A}\phi$ (For all paths): ϕ holds for all paths starting from the current state.
  - $\mathbf{E}\phi$ (There exists a path): ϕ holds for at least one path starting from the current state.

8.2 Semantics

Given a computation tree $\pi = s_0, s_1, s_2, \dots$ in the Kripke structure, the satisfaction relation $\models$ for a CTL* formula φ is defined inductively as follows:

- $s_0 \models p_i$ if and only if $p_i \in V(s_0)$, where p_i is an atomic proposition.
- $s_0 \models \neg\varphi$ if and only if $s_0 \not\models \varphi$.
- $s_0 \models \varphi \vee \psi$ if and only if $s_0 \models \varphi$ or $s_0 \models \psi$.
- $s_0 \models \varphi \wedge \psi$ if and only if $s_0 \models \varphi$ and $s_0 \models \psi$.
- $s_0 \models X\varphi$ if and only if there exists a successor state s_1 such that $s_1 \models \varphi$.
- $s_0 \models G\varphi$ if and only if for all states s_i, $s_i \models \varphi$.
- $s_0 \models F\varphi$ if and only if there exists a state s_i such that $s_i \models \varphi$.

- $s_0 \models \varphi\ U\ \psi$ if and only if there exists a state s_i such that $s_i \models \psi$, and for all $j < i$, $s_j \models \varphi$.
- $s_0 \models A\varphi$ if and only if for all paths π starting from s_0, $\pi \models \varphi$ at all states along the path.
- $s_0 \models E\varphi$ if and only if there exists a path π starting from s_0 such that $\pi \models \varphi$ at all states along the path.

In CTL*, the path quantifiers A and E can be applied to any part of the formula, not just at the top level. For example, $AF\varphi$ means "for all paths, eventually φ holds," while $EG\varphi$ means "there exists a path where φ holds globally."

8.3 Extension with Nested Path Quantifiers

The key difference between CTL and CTL* is the ability to apply path quantifiers to subformulas. This allows expressions like $AF\phi$, where the path quantifier A is applied only to the subformula $F\phi$, meaning "for all paths, eventually ϕ holds."

CTL* subsumes both CTL and LTL. This means that formulas that are valid in CTL and LTL can also be expressed in CTL*, and CTL* can express more complex temporal properties.

1. CTL formulas are a subset of CTL* formulas:

$$\varphi \in \text{CTL} \implies \varphi \in \text{CTL*}$$

Example: $\mathbf{AG}(N_1 \rightarrow \mathbf{EF}T_1)$ is valid in CTL and can also be expressed in CTL*.

2. LTL formulas can be expressed in CTL*:

$$\varphi \in \text{LTL} \implies A\varphi \in \text{CTL*}$$

Example: $\mathbf{A}(\mathbf{GF}T_1 \rightarrow \mathbf{GF}C_1)$ is a valid LTL formula, and $A(GFT1 \rightarrow GFC1)$ is equivalent in CTL*.

3. CTL and LTL together form a proper subset of CTL*:

$$\text{LTL} \cup \text{CTL} \subset \text{CTL*}$$

Example: $\mathbf{E}(\mathbf{GF}p \rightarrow \mathbf{GF}q)$ is a formula that cannot be expressed in LTL or CTL individually, but can be expressed in CTL*.

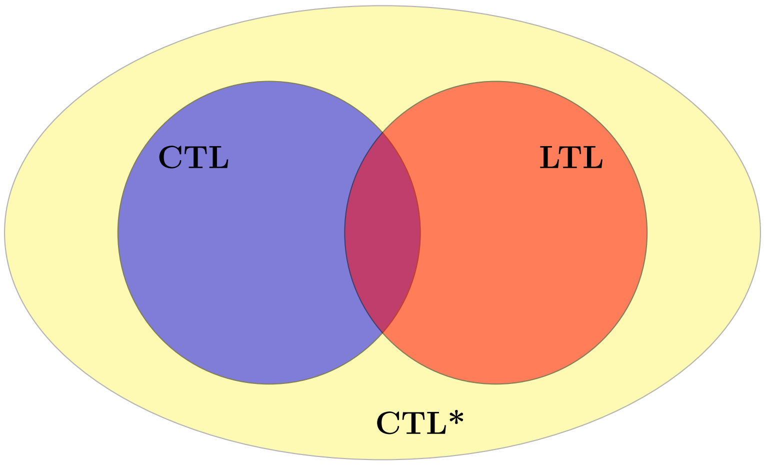

Figure 3.6: Venn Diagram of CTL, LTL, and CTL*

CHAPTER 4

MODEL CHECKING

1 CTL Model Checking

Computational Tree Logic (CTL) Model Checking is a formal verification technique in which the system under consideration is modeled as a finite state machine M. The model checking process involves verifying whether $M \models \varphi$, where φ is a specification expressed in CTL.

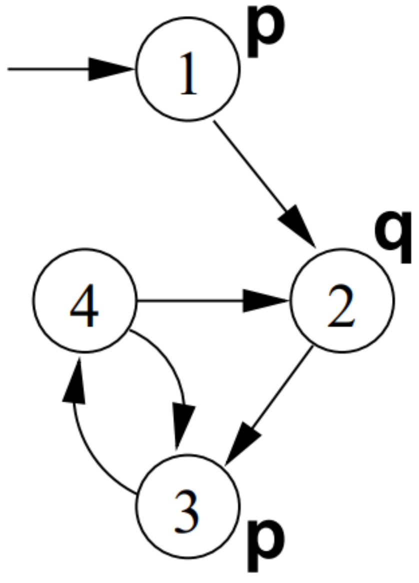

Figure 4.1: An example of CTL model checking.

$$\varphi = \mathrm{AG}(p \rightarrow \mathrm{AF}q)$$

This formula states that on all computation paths (A), globally (G), whenever the proposition p holds, it is guaranteed that at some future point (F), the proposition q will eventually hold.

The model checking algorithm verifies whether this property holds in the system by checking if:

$$M \models \varphi$$

That is, it checks whether in all initial states of the model M, all possible executions satisfy the formula φ.

1.1 Model checking algorithm

The model checking algorithm proceeds in two macro-steps:

1. **Construct the set of states where the formula holds:**

$$[\varphi] := \{s \in S : M, s \models \varphi\}$$

   where $[\varphi]$ is called the *denotation* of φ, representing the set of states in which the formula φ is satisfied.

2. **Compare the set of initial states with the set where the formula holds:**

$$I \subseteq [\varphi]$$

   where I denotes the set of initial states of the model M. If $I \subseteq [\varphi]$, then the formula φ is satisfied in all initial states of M; otherwise, it is not.

1.2 Example

To compute the denotation $[\varphi]$ for a given formula φ in Figure 4.1, the algorithm proceeds *bottom-up* on the structure of the formula. This means we compute the denotation for each subformula starting from the atomic propositions and building up to the complete formula. For the formula $\varphi = \mathrm{AG}(p \rightarrow \mathrm{AF}q)$, we proceed as follows:

1. **Atomic Proposition:**

$$[q] := \{s \in S : M, s \models q\}$$

   This is the set of states where the atomic proposition q holds.

2. **Next, for AFq:**

$$[\mathrm{AF}q] := \{s \in S : \exists s' \in \mathcal{F}(s), M, s' \models q\}$$

   This is the set of states from which there exists a path that leads to a state where q holds.

3. **For the atomic proposition p:**

$$[p] := \{s \in S : M, s \models p\}$$

   This is the set of states where the atomic proposition p holds.

4. **For the implication $p \to$ AFq:**

$$[p \to \mathrm{AF}q] := \{s \in S : M, s \models p \Rightarrow \exists s' \in \mathcal{F}(s), M, s' \models q\}$$

This is the set of states where if p holds, then there is a future state where q holds.

5. **For the globally quantified formula AG$(p \to$ AF$q)$:**

$$[\mathrm{AG}(p \to \mathrm{AF}q)] := \{s \in S : \forall s' \in \mathcal{F}(s), M, s' \models p \to \mathrm{AF}q\}$$

This is the set of states where, on all future paths, the formula $p \to \mathrm{AF}q$ holds.

Finally, the denotation of the entire formula $\varphi = \mathrm{AG}(p \to \mathrm{AF}q)$ is given by:

$$[\varphi] := [\mathrm{AG}(p \to \mathrm{AF}q)]$$

The model checking algorithm then verifies whether $I \subseteq [\varphi]$, where I is the set of initial states.

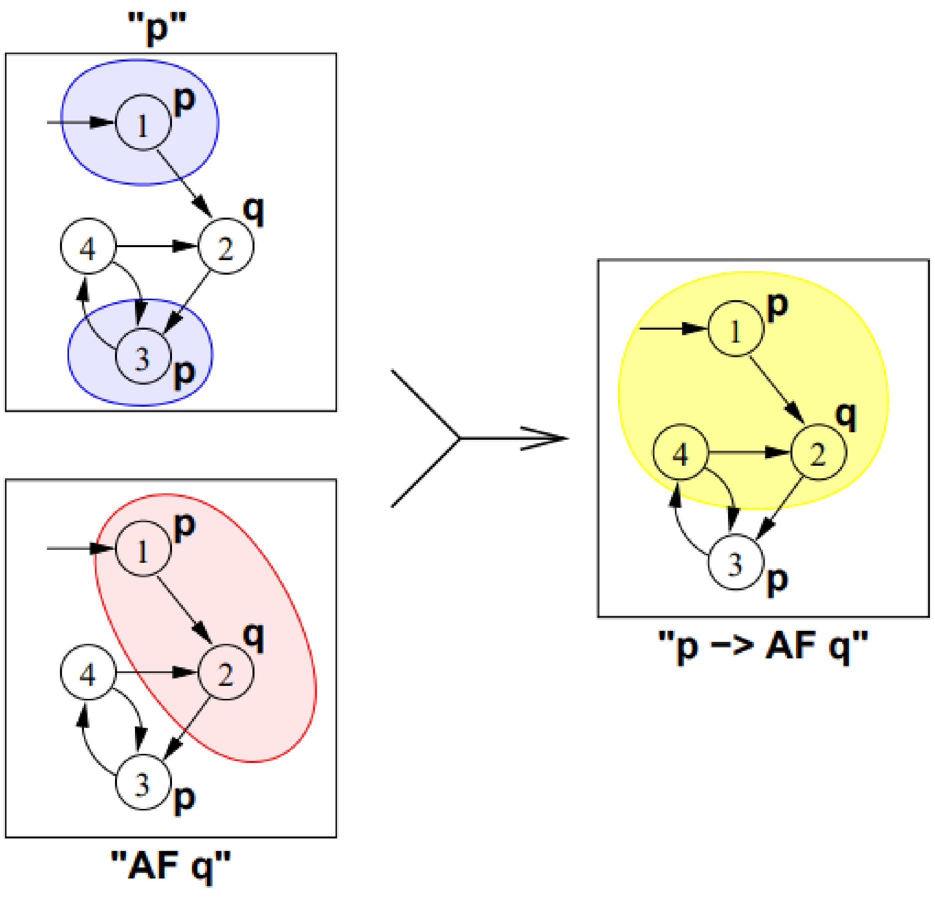

Figure 4.2: The first step of model checking.

Iteration:

1. $[\mathrm{AF}q]^{(1)} = [q]$

$$[\mathrm{AF}q]^{(1)} = [q] = \{2\}$$

2. $[\mathrm{AF}q]^{(i+1)} = [q] \cup \mathrm{AX}\,[\mathrm{AF}q]^{(i)}$

- For $i = 1$:

$$[\mathrm{AF}q]^{(2)} = [q] \cup \mathrm{AX}\,[\mathrm{AF}q]^{(1)} = \{2\} \cup \{1\} = \{1, 2\}$$

- For $i = 2$:

$$[\mathrm{AF}q]^{(3)} = [q] \cup \mathrm{AX}\,[\mathrm{AF}q]^{(2)} = \{2\} \cup \{1\} = \{1, 2\}$$

Thus, the process iterates and the denotation of AFq stabilizes at:

$$[\mathrm{AF}q] = \{1, 2\}$$

This process reflects the iterative nature of the tableau rule for the AF operator, where at each step we consider the set of states where q holds (directly), along with the set of states where AFq holds after one step (via the AX operator).

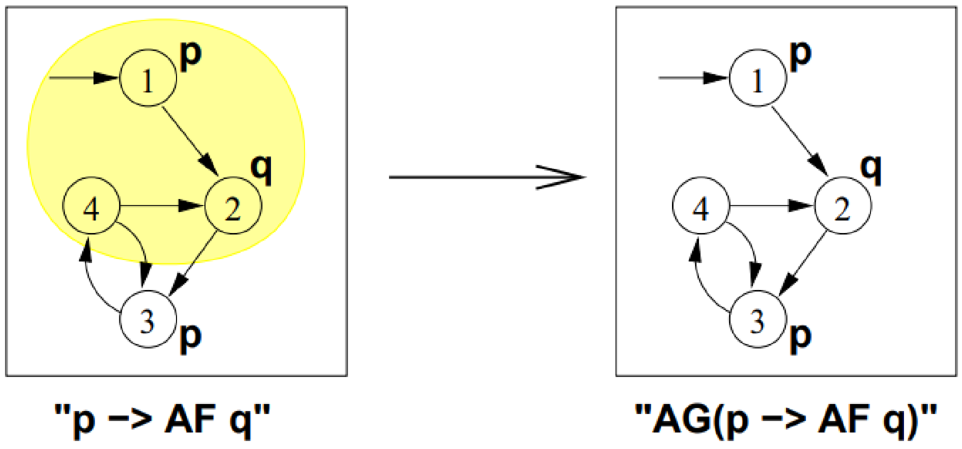

Figure 4.3: The second step of model checking.

Iteration:
1. $[\mathrm{AG}\varphi]^{(1)} = [\varphi] = [p \to \mathrm{AF}q]$

$$[\mathrm{AG}\varphi]^{(1)} = [\varphi] = [p \to \mathrm{AF}q] = \{1, 2, 4\}$$

2. $[\mathrm{AG}\varphi]^{(i+1)} = [\varphi] \cap \mathrm{AX}\,[\mathrm{AG}\varphi]^{(i)}$
- For $i = 1$:

$$[\mathrm{AG}\varphi]^{(2)} = [\varphi] \cap \mathrm{AX}\,[\mathrm{AG}\varphi]^{(1)}$$

$$= [\varphi] \cap \mathrm{AX}\,[\{1, 2, 4\}]$$

$$= \{1, 2, 4\} \cap \{1, 3\} = \{1\}$$

- For $i = 2$:

$$[\mathrm{AG}\varphi]^{(3)} = [\varphi] \cap \mathrm{AX}\,[\mathrm{AG}\varphi]^{(2)}$$

$$= [\varphi] \cap \mathrm{AX}[\{1\}]$$

$$= \{1, 2, 4\} \cap \emptyset = \emptyset$$

Thus, the process iterates and the denotation of $\mathrm{AG}\varphi$ stabilizes at the empty set:

$$[\mathrm{AG}\varphi] = \emptyset$$

If the set of states where the formula holds is empty, we can conclude the following:

$$[\mathrm{AG}(p \to \mathrm{AF}q)] = \emptyset \quad \Rightarrow \quad \text{The initial state does not satisfy the property}$$

$$\Rightarrow \quad M \not\models \mathrm{AG}(p \to \mathrm{AF}q)$$

This implies that the system M does not satisfy the formula $\mathrm{AG}(p \to \mathrm{AF}q)$. Hence, there exists a counterexample that demonstrates why the formula does not hold.

Counterexample: Consider a counterexample in the form of a path with a cycle, denoted as $1 \to 2 \to 3 \leftrightarrow 4$. Let the states in this cycle satisfy the following properties:
- p holds in state 1, but not in states 2, 3, or 4. - q holds in state 1, but not in states 2, 3, or 4.

Thus, the path forms a loop with the states $\{1, 2, 3, 4\}$, and the cycle satisfies the property:

$$\mathrm{EF}(p \wedge \mathrm{EG}\neg q)$$

This means that from state 1, there exists a future state (i.e., eventually) where p holds, and in all future states along the path from that state, q does not hold (i.e., $\neg q$).

Since this path satisfies $\mathrm{EF}(p \wedge \mathrm{EG}\neg q)$, it contradicts the formula $\mathrm{AG}(p \to \mathrm{AF}q)$, showing that $M \not\models \mathrm{AG}(p \to \mathrm{AF}q)$. Therefore, this path serves as a counterexample to the property.

2 Safety Property

A **safety property** asserts that *something bad never happens.* In CTL, this is expressed as:

Definition 2.1: Safety property

$$\mathbf{A}\,\neg\varphi$$

Where:

- $\mathbf{A}$: "on all paths",
- $\neg\varphi$: the undesirable condition φ never holds.

This ensures that on all execution paths, the system does not reach an undesirable state.

3 Liveness Property

A **liveness property** asserts that *something good eventually happens.* In CTL, this can be expressed as:

Definition 3.1: Liveness property

$$\mathbf{A}\,(\varphi \mathbf{U} \psi)$$

Where:

- $\mathbf{A}$: "on all paths",
- φ: condition that holds until ψ happens,
- ψ: the desirable state that must eventually occur.

Alternatively, we can use:

$$\mathbf{E}\,F\psi$$

Where:

- $\mathbf{E}$: "there exists a path",
- $F\psi$: eventually, ψ holds on some path.

4 Fairness Property

Fairness ensures that actions are not indefinitely postponed. In CTL, weak fairness (justice) and strong fairness (compassion) are expressed as:

- **Weak fairness**: If p happens infinitely often, eventually q must happen:

$$\mathbf{A}\,(p\mathbf{U}\,q)$$

- **Strong fairness**: If p happens infinitely often, then it will continue to happen infinitely often after q:

$$\mathbf{A}\,(p\mathbf{U}(q\mathbf{U}\,p))$$

5 Ordered Binary Decision Diagrams (OBDDs)

An *Ordered Binary Decision Diagram* (OBDD) is a directed acyclic graph (DAG) used to represent Boolean functions. It is a special form of Binary Decision Diagrams (BDDs), where the variables are ordered. Below is the formal definition and description of OBDDs.

Definition 5.1: Ordered Binary Decision Diagrams

Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function of n variables $x_1, x_2, \ldots, x_n$. An OBDD representing f is a directed acyclic graph $G = (V, E)$, where:

- V is the set of nodes in the graph.
- E is the set of directed edges between the nodes.
- Every non-terminal node $v \in V$ is labeled with a variable x_i, where $x_1, x_2, \ldots, x_n$ is a fixed total ordering of the variables, i.e., $x_1 < x_2 < \cdots < x_n$.
- Each non-terminal node has exactly two outgoing edges, labeled 0 and 1, representing the decision based on the Boolean value of the corresponding variable at that node.
- The terminal nodes represent the constant Boolean values 0 and 1.

Formal Representation

Given the Boolean function $f(x_1, x_2, \ldots, x_n)$, an OBDD is a graph $G = (V, E)$ that represents f as follows:

- The root node is labeled with the first variable x_1.
- If a node is labeled with x_i, then the two outgoing edges from this node lead to the nodes that represent the Boolean function when $x_i = 0$ and $x_i = 1$, respectively.

- The function f at a node labeled with x_i is determined by the following recursive relation:

$$f(x_1, x_2, \ldots, x_n) = \begin{cases} f_0(x_1, x_2, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) & \text{if the edge labeled 0 is taken} \\ f_1(x_1, x_2, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) & \text{if the edge labeled 1 is taken} \end{cases}$$

  where f_0 and f_1 represent the functions at the child nodes corresponding to $x_i = 0$ and $x_i = 1$, respectively.

5.1 Properties of OBDDs

An OBDD has several key properties that distinguish it from other representations of Boolean functions:

- **Ordered Structure:** The variables in the decision nodes must follow a fixed order. That is, for any node labeled x_i, all nodes in the path leading to this node must have labels from the set $\{x_1, x_2, \ldots, x_i\}$, maintaining a total order.
- **Reduction:** An OBDD can be reduced by eliminating redundant nodes and merging equivalent nodes. Specifically:
  - If two nodes represent the same function, they are merged.
  - If two edges have the same destination, they are merged into a single edge.
- **Canonical Representation:** If the OBDD is reduced and the variable ordering is fixed, then the OBDD provides a unique representation for each boolean function.

5.2 Example

Consider the boolean function:

$$f(a, b, c, d) = (a \wedge b) \vee (c \wedge d).$$

The construction of the OBDD proceeds as follows:

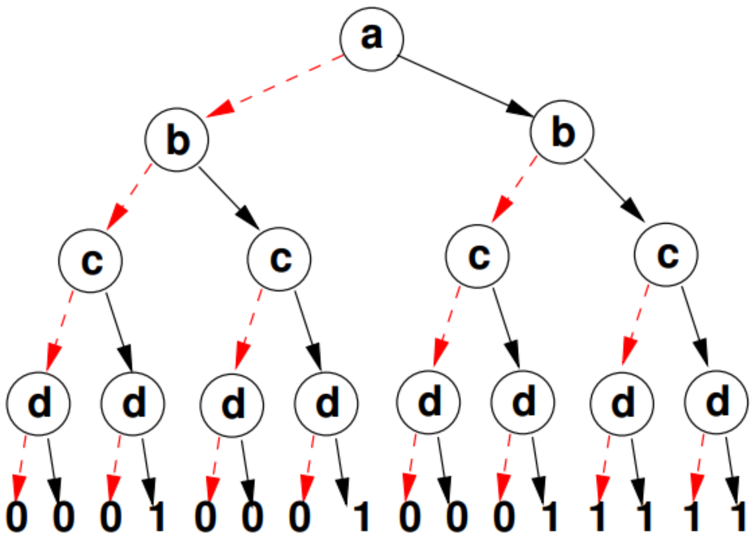

Figure 4.4: The 1st step of OBDDs construction.

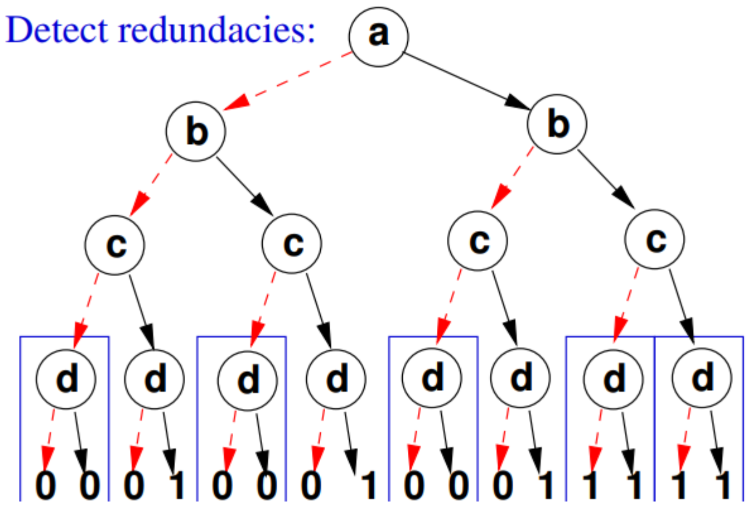

Figure 4.5: The 2nd step of OBDDs construction.

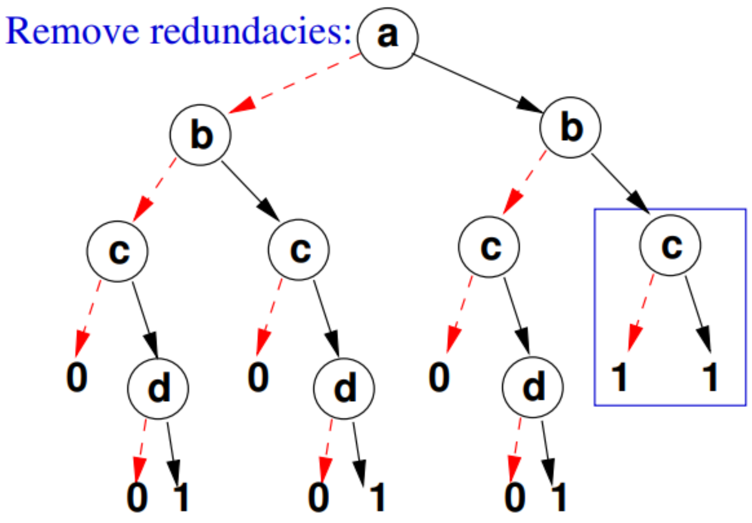

Figure 4.6: The 3rd step of OBDDs construction.

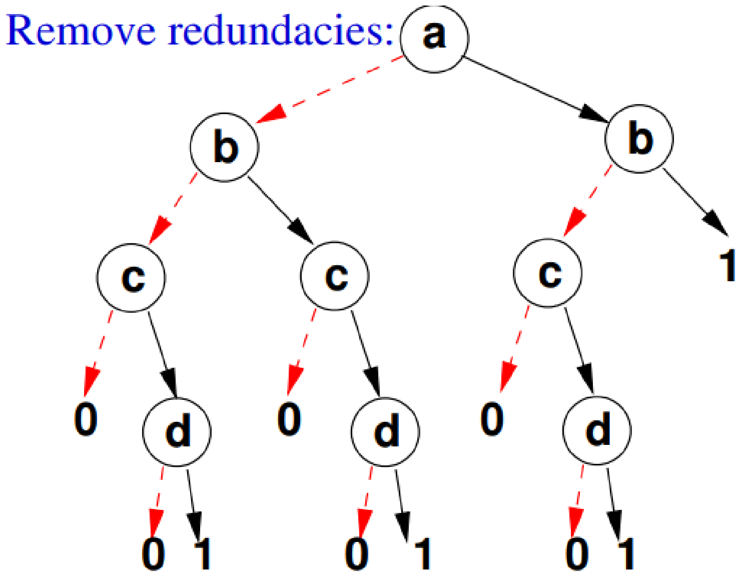

Figure 4.7: The 4th step of OBDDs construction.

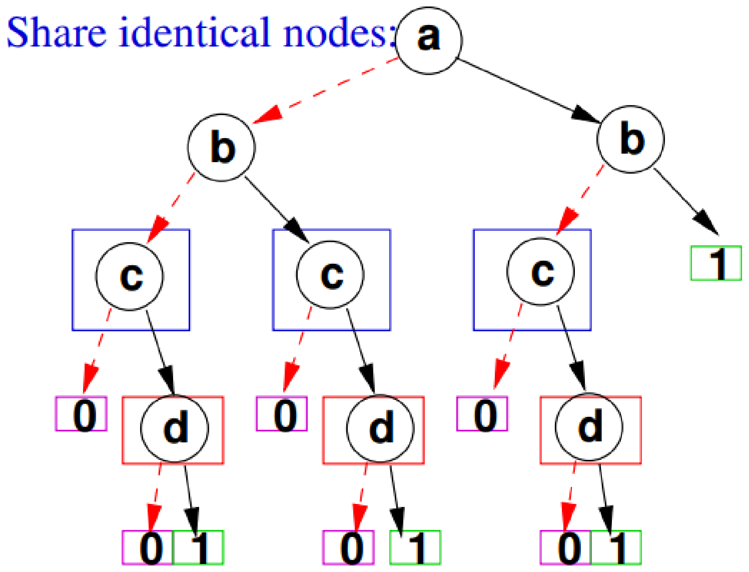

Figure 4.8: The 5th step of OBDDs construction

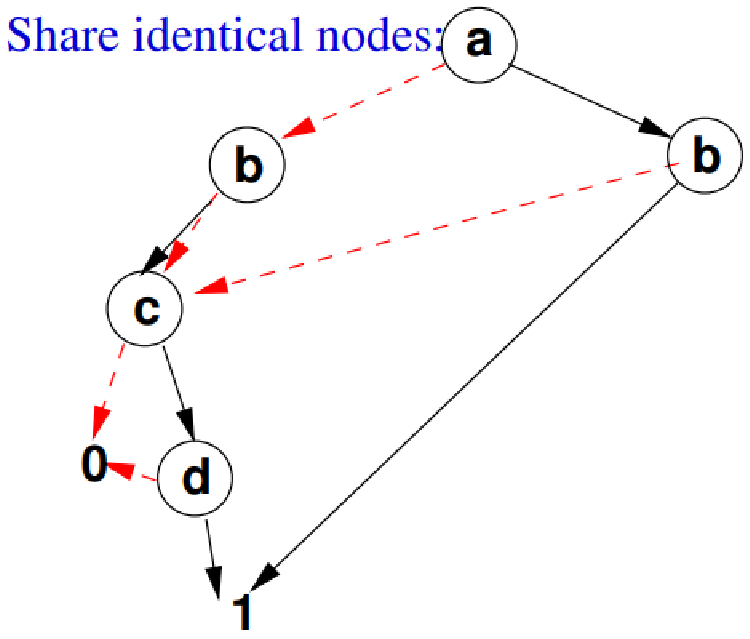

Figure 4.9: The 6th step of OBDDs construction

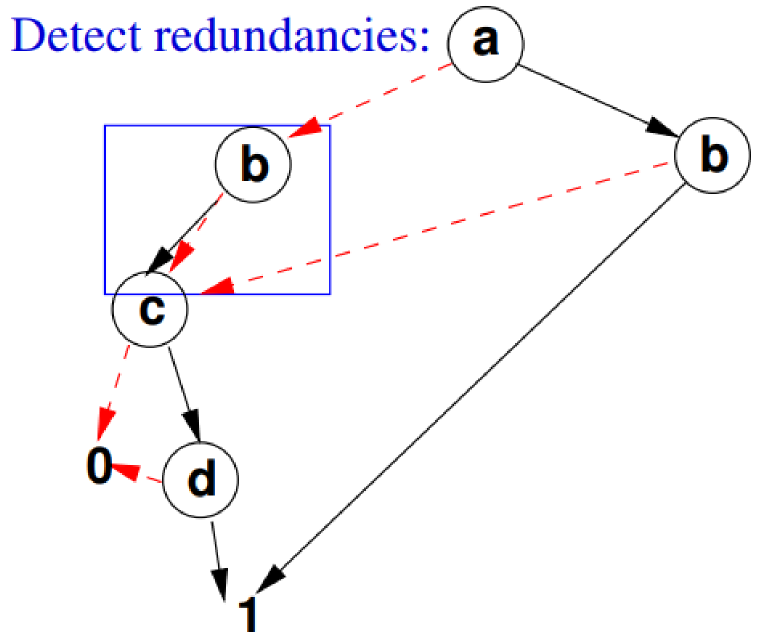

Figure 4.10: The 7th step of OBDDs construction.

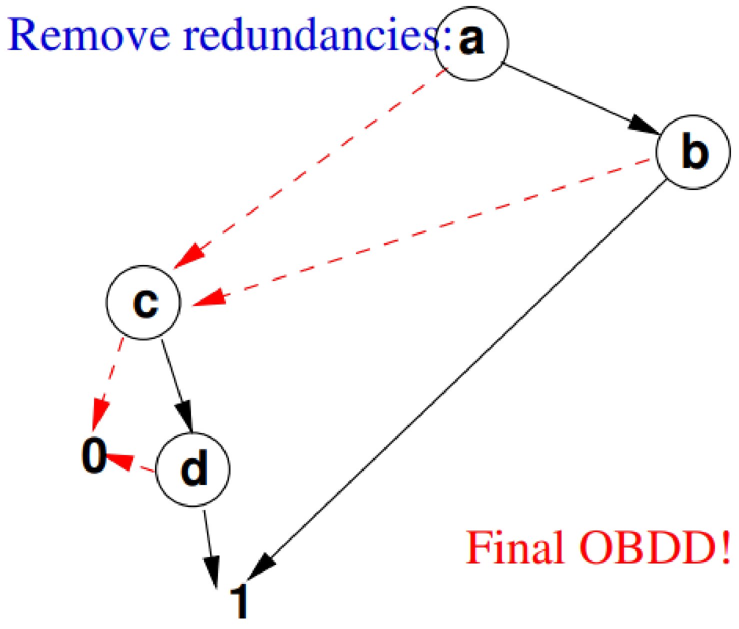

Figure 4.11: The 8th step of OBDDs construction.

CHAPTER 5

SAT-BASED MODEL CHECKING

In the context of model checking, the goal is to verify whether a given temporal property φ holds for a system $\mathcal{T}$. One approach for checking the validity of a property is based on SAT solvers. This method involves encoding the system model and the property as a boolean formula and then using a SAT solver to determine whether the property holds. The essence of the approach is that a **refutation** of the property corresponds to finding a **counterexample** that violates the property.

Given a system $\mathcal{T} = (X, \Sigma, \delta)$ with state variables $x_1, x_2, \ldots, x_n$, where Σ represents the set of states and δ represents the transition relation, we encode the system's dynamics and the temporal property φ as boolean formulas. The system's transitions over time are represented by a set of boolean clauses Φ_{system}, and the temporal property is encoded as another set of boolean clauses $\Phi_{\text{property}}(\varphi)$.

The combined boolean formula representing both the system and the property is:

$$\Phi(S, \varphi) = \Phi_{\text{system}} \wedge \Phi_{\text{property}}(\varphi)$$

The SAT solver checks the satisfiability of this formula. A satisfiable formula implies the existence of a **counterexample trace**, which is a sequence of states that violates the property φ, thus refuting it. If the formula is unsatisfiable, it indicates that the property holds, as no counterexample exists.

1 Techniques for SAT

1.1 Resolution

Let φ be a temporal property represented as a set of clauses $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$, where each clause C_i is a disjunction of literals $\ell \in L(C_i)$, i.e., $C_i = \bigvee_{\ell \in L(C_i)} \ell$, and $L(C_i)$ is the set of literals in clause C_i.

The goal is to search for a refutation of φ, which corresponds to finding a counterexample that violates the property. This search is carried out using the resolution rule, which combines pairs of clauses containing a conflicting literal.

Definition 1.1: Resolution Rule

$$C_{\text{res}} = (C_i \setminus \{\ell\}) \cup (C_j \setminus \{\neg\ell\})$$

where

- $C_i \in \mathcal{C}$ a clause that there exists a literal $\ell \in L(C_i)$
- $C_j \in \mathcal{C}$ a clause that there exists a literal $\neg\ell \in L(C_j)$

Definition 1.2: Subsumption Rule

$$C_j \text{ is subsumed by } C_i \quad \text{if} \quad C_j \subseteq C_i \quad \text{and} \quad C_i \neq C_j$$

where

- $C_i \in \mathcal{C}$ is a clause,
- $C_j \in \mathcal{C}$ is a clause,
- If $C_j \subseteq C_i$ and $C_i \neq C_j$, then C_j is removed from the set of clauses $\mathcal{C}$.

This clause C_{res} is then added to the set of clauses $\mathcal{C}$.

The process of applying the resolution rule is iterative. At each step, a pair of clauses $C_i, C_j \in \mathcal{C}$ containing conflicting literals is selected, and the resolution rule is applied to generate a new clause. The process continues until one of two conditions is met:

1. Contradiction: A false clause (empty clause) $C_{\text{res}} = \emptyset$ is generated, indicating that the set of clauses is unsatisfiable and thus φ is refuted.
2. Termination: No more pairs of clauses containing conflicting literals can be found, meaning no further resolution can be applied.

Refutation Search Process

Let $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ represent the set of clauses encoding the temporal property φ. The refutation search process is formally defined as follows:

1. **Select a pair of clauses** $C_i, C_j \in \mathcal{C}$ such that there exists a literal ℓ in $L(C_i)$ and its negation $\neg\ell$ in $L(C_j)$.
2. **Apply the resolution rule** on C_i and C_j:

$$C_{\text{res}} = (C_i \setminus \{\ell\}) \cup (C_j \setminus \{\neg\ell\})$$

3. **Add the resolved clause** C_{res} to the set of clauses $\mathcal{C}$.
4. **Repeat the process** until one of the following conditions is met:

(a) A contradiction is found, i.e., the resolution process generates a false clause:

$$C_{\text{res}} = \emptyset$$

In this case, φ is refuted, and the search terminates.

(b) The resolution rule is no longer applicable, i.e., no more pairs of clauses containing conflicting literals can be found, and the process terminates without finding a refutation.

Formulation of the Process

Let $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ be the initial set of clauses. The iterative application of the resolution rule can be formally described as follows:

$$\begin{gathered}
\text{For each pair of clauses } C_i, C_j \in \mathcal{C}, \quad \exists \ell \in L(C_i), \neg\ell \in L(C_j), \\
C_{\text{res}} = (C_i \setminus \{\ell\}) \cup (C_j \setminus \{\neg\ell\}) \\
\mathcal{C} \leftarrow \mathcal{C} \cup \{C_{\text{res}}\} \\
\text{Repeat the process until:} \\
1.\ C_{\text{res}} = \emptyset, \quad \text{refuting } \varphi \\
2.\ \text{No more conflicting literals exist, and the process terminates.}
\end{gathered}$$

Example

Example 1 : Let C_1 and C_2 be two clauses such that:

$$C_1 = (A \lor B) \quad \text{and} \quad C_2 = (\neg B \lor C)$$

The resolution rule is applied by removing B from C_1 and $\neg B$ from C_2, and combining the remaining literals:

$$C_{\text{res}} = (C_1 \setminus \{B\}) \cup (C_2 \setminus \{\neg B\})$$

Substituting the values for C_1 and C_2, we get:

$$C_{\text{res}} = (A \lor C)$$

Thus, the resolved clause is $C_{\text{res}} = (A \lor C)$.

Example 2 : Let C_1, C_2, C_3 be three clauses:

$$\begin{gathered}
C_1 = (A \lor B \lor C) \\
C_2 = (\neg A \lor \neg B) \\
C_3 = (B \lor \neg C \lor D)
\end{gathered}$$

We resolve C_1 and C_2 on the literal A and its negation $\neg A$. The resolution rule states that we remove A from C_1 and $\neg A$ from C_2, and then combine the remaining literals:

$$C_{\text{res1}} = (B \lor C) \lor (\neg B)$$

Next, we resolve $C_{\text{res1}} = (B \lor C) \lor (\neg B)$ with $C_3 = (B \lor \neg C \lor D)$ on the literal B and its negation $\neg B$. Again, we remove B from C_{res1} and $\neg B$ from C_3, and then combine the remaining literals:

$$C_{\text{res2}} = (C \lor D)$$

Thus, the final resolved clause is:

$$C_{\text{final}} = (C \lor D)$$

1.2 DPLL

The Davis-Putnam-Longeman-Loveland procedure (DPLL) procedure is an algorithm for determining whether a given propositional logic formula φ is satisfiable. It is a backtracking-based extension of the Davis-Putnam procedure, with two key elements: recursive truth assignment and deterministic decision making.

DPLL Procedure

The DPLL procedure can be defined recursively as follows:

1. **Base Case - Check for Satisfaction:** If the formula φ is satisfied under the current assignment μ, return **True**. This means that μ is a satisfying assignment for the formula.

2. **Check for Contradiction:** If φ contains an empty clause under μ (i.e., a clause that is false under the current assignment), return **False**. This indicates that the formula is unsatisfiable under the current assignment.

3. **Choose an Unassigned Literal:** Select an unassigned literal $\ell \in L(\varphi)$. This literal is chosen deterministically (based on a heuristic, if applicable).

4. **Unit Propagation and Pure Literal Elimination (optional):** If applicable, apply unit propagation (assigning truth values based on unit clauses) or pure literal elimination (assigning truth values to literals that appear with only one polarity) to simplify the formula.

5. **Recursively Check with $\ell =$ True** : Apply the DPLL procedure to the formula with $\mu(\ell) = \text{True}$, i.e., call $\text{DPLL}(\varphi(\mu(\ell) = \text{True}), \mu)$.

6. **Recursively Check with $\ell =$ False** : If the previous step fails, recurse on the formula with $\mu(\ell) = \text{False}$, i.e., call $\text{DPLL}(\varphi(\mu(\ell) = \text{False}), \mu)$.

7. **Return False if No Satisfying Assignment:** If no satisfying assignment is found after all recursive calls, return **False**. This means the formula is unsatisfiable.

Termination

The algorithm terminates when a satisfying assignment is found, or when the formula is determined to be unsatisfiable.

Algorithm 1 DPLL Algorithm

```
1: procedure DPLL($\varphi$, $\mu$)
2:     if $\varphi = \top$ then                                  ▷ Base Case: Satisfiable
3:         return True
4:     end if
5:     if $\varphi = \bot$ then                                  ▷ Backtrack Case: Unsatisfiable
6:         return False
7:     end if
8:     if there exists a unit clause $l$ in $\varphi$ then      ▷ Unit Propagation
9:         Return DPLL(assign($l, \varphi$), $\mu \wedge l$)
10:    end if
11:    $l \leftarrow$ choose-literal($\varphi$)                  ▷ Choose a literal $l$
12:    return DPLL(assign($l, \varphi$), $\mu \wedge l$) or DPLL(assign($\neg l, \varphi$), $\mu \wedge \neg l$)
13: end procedure
```

1.3 CDCL

CDCL is an extension of the DPLL algorithm that incorporates conflict-driven learning, backtracking, and non-chronological reasoning. The main idea behind CDCL is to learn new clauses based on conflicts that arise during the search for a solution, which enables the solver to avoid re-exploring regions of the search space that have already been proven to be unsatisfiable. This leads to significant improvements in the efficiency and performance of SAT solvers.

In CDCL-based solvers, when a conflict is detected (i.e., a contradiction is found under the current assignment), the solver analyzes the conflict and generates a new clause that excludes the conflicting assignment. This newly learned clause is added to the formula, and the search space is pruned to avoid revisiting the same conflict.

The key components of CDCL solvers are:

- **Conflict Analysis:** When a conflict occurs, the solver analyzes the conflict and learns a new clause that excludes the cause of the conflict.
- **Clause Learning:** The solver adds the learned clause to the formula, preventing the algorithm from revisiting the same conflict.
- **Backtracking:** The solver backtracks non-chronologically to a decision point that can potentially resolve the conflict by assigning different truth values to the literals involved.
- **Decision Heuristics:** CDCL solvers typically use advanced heuristics to decide the next literal to assign, balancing between exploration and exploitation of learned knowledge.

This enhancement leads to a dramatic improvement in the performance of SAT solvers in practice, especially for hard industrial and research problems.

Conflict Analysis

The intuition behind conflict analysis is as follows: when a branch μ fails (i.e., leads to a contradiction), we analyze the conflict to find a smaller subset of literals $\eta \subset \mu$, typically much smaller than μ, such that assigning only the literals in η would have falsified the same clause after a chain of unit propagations. Formally, let:

$$\eta \subseteq \mu$$

such that the assignment of η would have led to the same conflict as μ after unit propagation.

Intuition: The set η contains only the relevant assignments that caused the failure. These literals are the "culprits" that can explain the conflict, and finding η allows us to focus on the minimal set of conflicting assignments.

Learning

Once the conflict analysis and backjumping have been performed, the next step is to learn a new clause that will help avoid the same conflict in future branches. The learned clause contains the relevant information about why the conflict occurred and is added to the formula. The learning process can be formalized as follows:

$$\text{Learned clause: } C = \{\ell_1, \ell_2, \ldots, \ell_n\} \quad \text{where} \quad \ell_1 \text{ is the remaining literal in } \eta.$$

Intuition: The learned clause ensures that in future branches, once all literals except for one in η are assigned, the remaining literal is forced to be false by unit propagation. This avoids the solver repeating the same mistake that caused the conflict. Specifically, when the solver reaches the point where all but one literal of η are assigned, unit propagation will assign the last literal of η to false, preventing the same conflict from occurring again.

Formally, the solver will update the assignment μ with the learned clause C and propagate the values of the literals in C. The new assignment is:

$$\mu' = \mu \cup \{\ell_i \mid \ell_i \in C\}.$$

The unit propagation will assign $\ell_1 = \text{False}$ as soon as all literals except ℓ_1 are assigned.

Intuition: When about to repeat the same mistake, do the opposite of the last step (i.e., assign $\ell_1 = \text{False}$). This prevents the solver from revisiting the same conflict.

Backjumping

Backjumping is a technique that allows the solver to climb back up the decision stack to a higher decision level. This avoids unnecessary exploration of parts of the search space that would have led to the same failure. Formally, we can define the backjumping step as follows:

$$\text{Backjump to level } k, \text{ where } k \text{ is the decision level}$$

$$\text{where a different choice would have been made if } \eta \text{ had been known.}$$

Intuition: The idea is to "jump back" to the oldest decision level k where a different choice would have been made had we known η. This avoids redundant search by skipping over decision levels that would not have led to a different outcome.

The backjumping process can be summarized as:

$$\text{Backjump to level } k \quad \text{where } k = \max\left\{l \mid \forall \ell \in \eta, \ell \text{ is assigned at decision level } l \leq k\right\}.$$

Algorithm 2 CDCL Algorithm

```
procedure CDCL($\varphi$)
    $\mu \leftarrow \emptyset$                                    ▷ Initialize the partial assignment
    $decision\_stack \leftarrow \emptyset$                          ▷ Initialize the decision stack
    while True do
        $l \leftarrow$ Choose-Literal($\varphi, \mu$)                  ▷ Select a literal to assign
        $\mu \leftarrow \mu \cup \{l\}$                     ▷ Assign the literal to the partial assignment
        $decision\_stack \leftarrow decision\_stack \cup \{l\}$          ▷ Push literal onto the stack
        $\varphi' \leftarrow$ Unit-Propagation($\varphi, \mu$)             ▷ Apply unit propagation
        if $\varphi' = \emptyset$ then                         ▷ If conflict occurs (empty clause)
            $conflict\_analysis(\varphi, \mu, decision\_stack)$         ▷ Perform conflict analysis
            $\varphi \leftarrow$ Add-Learned-Clause($\varphi, \mu$)      ▷ Learn a new clause from the conflict
            Backjump()                        ▷ Backtrack based on learned information
        else
            if $\varphi'$ is satisfied then                       ▷ If formula is satisfied
                Return True                                            ▷ Satisfiable
            end if
        end if
    end while
end procedure
procedure CONFLICT_ANALYSIS($\varphi, \mu, decision\_stack$)
    $\eta \leftarrow$ Analyze-Conflict($\varphi, \mu$)        ▷ Analyze the conflict to find relevant assignments
    $learned\_clause \leftarrow$ Generate-Learned-Clause($\eta$)        ▷ Generate a clause based on conflict
    Add-Conflict-Clause($\varphi, learned\_clause$)      ▷ Add the learned clause to the formula
end procedure
procedure BACKJUMP(())
    level $\leftarrow$ Find-Backtrack-Level()              ▷ Find the decision level to backtrack to
    Backtrack(level)              ▷ Move back to the appropriate decision level in the stack
end procedure
```

2 Bounded Model Checking

Given a system model $M = (S, I, T)$, where S is the set of states, I is the set of initial states, and $T \subseteq S \times S$ is the transition relation, and a property φ.

We wish to determine if there exists a counterexample within the first k steps, which is a path $\{x_0, x_1, \ldots, x_k\}$ that violates φ. The SAT formula to check is:

$$\text{SAT}_k = \text{SAT}\left(\bigwedge_{t=0}^{k-1} T(x_t, x_{t+1}) \wedge (x_0 \in I) \wedge \bigvee_{t=0}^{k} \neg\varphi(x_t)\right)$$

where:

- $x_0 \in I$ ensures the initial state is valid,
- $T(x_t, x_{t+1})$ ensures valid transitions for each time step $t \in \{0, 1, \ldots, k-1\}$,
- $\neg\varphi(x_t)$ ensures that φ is violated at some time step.

2.1 Iterative Search for Increasing Length k

The search for counterexamples proceeds by increasing k and checking the satisfiability of the formula at each step:

$$\text{While } k \leq k_{max}, \text{ check if } \text{SAT}_k \text{ is satisfiable.}$$

If satisfiable, return the counterexample. If unsatisfiable, increment k and repeat

Algorithm 3 SAT-based Bounded Model Checking (BMC)

```
1:  procedure BMC_ITERATIVE($M, \varphi, k_{max}$)
2:      $k \leftarrow 1$                                         ▷ Start with a small bound
3:      while $k \leq k_{max}$ do
4:          $\varphi_{encoded} \leftarrow$ Encode-SAT($M, \varphi, k$)      ▷ Encode system and property for length $k$
5:          $result \leftarrow$ SAT-Solver($\varphi_{encoded}$)               ▷ Check for satisfiability
6:          if result is SAT then
7:              Return "Counterexample found at length $k$"
8:          else
9:              $k \leftarrow k + 1$                             ▷ Increase bound and try again
10:         end if
11:     end while
12:     Return "No counterexample found within bound $k_{max}$"
13: end procedure
```

3 K-induction

Given a system model $M = (S, I, T)$, where S is the set of states, I is the set of initial states, and $T \subseteq S \times S$ is the transition relation, and a property φ that we want to verify, the procedure for applying k-induction is as follows:

3.1 Base Case Check

Verify that the property φ holds at the initial state:

$$\varphi(x_0) \quad \text{for all} \quad x_0 \in I$$

3.2 Inductive Step Check

For each $t \in \{0, 1, \ldots, k-1\}$, ensure that:

$$\varphi(x_t) \Rightarrow \varphi(x_{t+1}) \quad \text{for all transitions} \quad (x_t, x_{t+1}) \in T$$

Algorithm 4 *K*-Induction in Model Checking

```
1: procedure K-INDUCTION($M, \varphi, k$)
2:     Input: System model $M = (S, I, T)$, Property $\varphi$, Bound $k$
3:     Output: "Property holds for all reachable states up to bound $k$" or "Counterexample
   found"
4:     for each $x_0 \in I$ do
5:         if $\neg\varphi(x_0)$ then
6:             Return "Counterexample found at $t = 0$"
7:         end if
8:     end for
9:     for each $t \in \{0, 1, \ldots, k-1\}$ do
10:        for each $x_t \in S$ do
11:            if $\varphi(x_t)$ then
12:                Check the transition $T(x_t, x_{t+1})$
13:                if $\neg\varphi(x_{t+1})$ then
14:                    Return "Counterexample found at $t + 1$"
15:                end if
16:            end if
17:        end for
18:    end for
19:    Return "Property holds for all reachable states up to bound $k$"
20: end procedure
```

BIBLIOGRAPHY

Clarke et al.: Handbook of Model Checking **Clarke2018**

Edmund M. Clarke et al., eds. *Handbook of Model Checking*. First edition. Springer Nature, 2018. ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8.

Sebastiani: Fondamenti di Matematica 2020 **fm2020**

Rocco Sebastiani. *Fondamenti di Matematica 2020*. 2020. URL: https://disi.unitn.it/rseba/DIDATTICA/fm2020/ (visited on 04/20/2025).