SATISFIABILITY JUDGMENT OF MODAL LOGICS AND THEIR APPLICATION TO VERIFICATION PROBLEMS

様相論理の充足可能性判定手続きと検証問題への応用

by

Yoshinori Tanabe

田辺 良則

A Doctor Thesis

博士論文

Submitted to Department of Computer Science,

Graduate School of Information Science and Technology,

the University of Tokyo,

on March 14, 2008

in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Thesis Supervisor: Masami Hagiya　萩谷 昌己

Professor

ABSTRACT

Extensive research has been conducted for applying modal logics to the analysis of systems that handle tree structures such as XML documents. A key factor that enables the analysis is the tree model property: a satisfiable formula is satisfied by a tree structure. Various modal logics, including the minimal modal logic **K**, have this property. The formulae in these logics can be used to describe the properties of tree structures owing to the tree model property. The tree model property also plays a central role in proving that the satisfiability problems of the logics are decidable.

The main purpose of this dissertation is to study modal logics that can be applied to the analysis of graph structures that are not necessarily in the tree shape. The logics should have enough expressive power so that properties of the graph structure to be analyzed can be described in them, their satisfiability problem should be decidable, and the decision procedures should be efficient for practical applications.

To meet the conditions described above, we introduce the following extensions to the modal logic **K**. (a) The fixed-point operators μ and ν: we also consider some restrictions on the nesting of the operators. Properties that occur frequently in applications, such as reachability, can be described using these operators. (b) Atomic formulae called nominals: the introduction of nominals breaks the tree model property, and it is a key extension for the application of logics to the analysis of graphs that are not trees. (c) Reverse modalities: they increase the expressive power of the logic. Moreover, the weakest preconditions for graph transformation operations, which are necessary in applications to verification problems, can be expressed by formulae with reverse modalities. (d) Restriction to functional Kripke structures: it is required when we handle certain types of graphs such as program heaps.

The abovementioned extensions and restrictions can be introduced independently. Appropriate combinations should be used depending on the graph structures or operations to be analyzed. However, arbitrary combinations cannot be utilized since the satisfiability problem of logics with all the extensions from (a) to (d) is known to be undecidable. On the other hand, any logic with three out of the four abovementioned extensions is decidable. In this study, for combinations that are important in terms of application, we provide concrete decision procedures and prove their correctness. Although these procedures require some restrictions for the nesting of the fixed point operators, they have less complexity as compared with known procedures and efficient implementations are possible. They enable practical applications, as described below.

We apply decision procedures of modal logics to verification problems in two domains. One target is shape analysis. We build a framework for the verification of the shapes of data structures resulting from pointer operations in procedural programming languages. We implement a tool by using the one of the abovementioned decision procedures and solve some verification problems; we also verify the correctness of Deutsch-Schorr-Waite marking algorithm, which is regarded as a measure of the effectiveness of a method in shape analysis. The other target is one-dimensional cellular automata. We present a method for building finite abstract transition systems for verifying the properties of infinite transition systems of cellular automata by using modal logics. An implementation based on this method is available.

論文要旨

様相論理を用いて，XML 文書をはじめとする木構造を扱うシステムを解析する研究が行われている．このような解析を可能とする鍵になるのは，充足可能な論理式は木構造によって充足される，という性質であり，これは木モデル性と呼ばれている．最小の様相論理 **K** をはじめとするいくつもの様相論理がこの性質を持っている．木モデル性により，木構造の性質を記述するために論理式を使うことができる．また，木モデル性は充足可能性問題が決定可能であることを証明する際にも重要な役割を演じる．

本論文では，必ずしも木構造ではないグラフ構造に関する解析に応用可能な様相論理について研究することを主目的とする．あつかう論理は，解析したいグラフ構造の性質を記述できるだけの表現力を持つ必要がある．また，充足可能性問題が決定可能であること，さらに，実用的な応用を想定すれば，効率的な充足可能性判定手続きが存在することが要求される．

上述の条件を満たすため，様相論理 **K** に対して，以下の拡張を行う: (a) 不動点演算子 μ および ν の導入．ただし，不動点演算子のネストにある種の制限を設ける．この導入により，到達可能性など，応用上頻繁に現れる性質が記述できるようになる．(b) ノミナルと呼ばれる原子論理式の導入．この導入により，木モデル性が失われる．木構造以外のグラフを記述するための鍵となる拡張である．(c) 逆様相の導入．論理の記述力が増加する．また，検証問題への応用において必要になる，グラフを変換する操作に関する最弱事前条件が，逆様相を用いた論理式で記述できる．(d) 関数型クリプキ構造への制限の導入．この制限は，ある種のグラフ，たとえばプログラムのヒープ構造などを扱う際に必要である．

以上の導入は，おのおの独立に行いうる．解析したいグラフの構造や操作に応じて，適当な組み合わせが考えられる．しかし，その任意の組合せが応用可能なわけではない．上記 (a) から (d) までのすべてを導入した論理の充足可能性判定問題は，決定不能であることが知られている．しかし，3 種類を導入した論理はどの組み合わせでも決定可能である．本研究では，応用上重要な 3 つの組み合わせに関して，具体的な判定手続きを与え，その正当性を証明する．この判定手続きが適用可能であるためには，不動点演算子のネストに制限が必要であるが，この制限によって，従来知られている手続きに比較して計算量が小さくなっており，効率的な実装が可能である．このため，以下に述べるような応用が実現可能となる．

論文の後半では，様相論理の充足可能性決定手続きを応用した，2 つの分野の検証問題を扱う．ひとつの応用分野は，シェープ解析である．手続き型プログラム言語におけるポインタ操作における，データ構造の形状に関する検証を行う枠組みを構築する．前述の組み合わせの 1 つに関する充足可能性判定手続きを用いて，ツールを実装し，いくつかの検証問題が解けることを示す．このなかには，シェープ解析における検証手段の実効性の目安となる例題である Deutsch-Schorr-Waite マーキングアルゴリズムと呼ばれるアルゴリズムの正当性の検証問題が含まれる．もうひとつの応用は，一次元セルオートマトンである．セルオートマトンのなす無限遷移系の性質を検証するために，様相論理を利用した有限抽象遷移系を構成する方法を与え，これに基づいた実装を行う．

Acknowledgements

First of all, I wish to express my sincere gratitude to Professor Masami Hagiya, my supervisor, for his kindly advice, warm encouragement, and helpful supports during the research.

I wish to say grateful thanks to Dr. Yoshiki Kinoshita, Director of Research Center for Verification and Semantics of National Institute of Advanced Industrial Science and Technology (AIST). He provided me a chance to work in academia and encourage me to take doctoral course.

I express my thanks to my colleagues in AIST. Discussions with members of our research group, Dr. Koichi Takahashi, the leader of the group, Dr. Toshinori Takai, Mr. Toshifusa Sekizawa, and Dr. Yoshifumi Yuasa were very helpful. I also would like to thank Dr. Makoto Takeyama, Dr. Misao Nagayama, Dr. Koki Nishizawa, and many others in AIST for their kind help and suggestions during the research.

I also thank members and ex-members of Hagiya Laboratory, especially Professor Mitsuharu Yamamoto of Chiba University, Mr. Akihiko Tozawa, and Mr. Takahiro Sato. Discussions with them greatly helped me to make vague ideas concrete.

Lastly, let me express my deep thanks to my wife Keiko for supporting me mentally as well as financially. Without her understanding and sacrifices, I could never have completed the thesis.

Contents

1 Introduction **1**

1.1 Background and Motivation 1

1.2 Overview 3

1.2.1 Dimensions to Extend the Modal Logic 3

1.2.2 Satisfiability and Decision Procedures 4

1.2.3 Applications 5

1.3 Contributions 6

1.4 Organization 7

2 Modal Logics **8**

2.1 The Modal Language 8

2.1.1 Syntax 8

2.1.2 Semantics 10

2.2 Fixed-point Operators 12

2.2.1 The Modal μ-calculus 12

2.2.2 Parity Games 15

2.3 The Alternation of Fixed-point Operators 18

2.4 Nominals 21

2.5 Reverse Modalities 24

2.6 Functional Kripke Structures 25

2.7 The Tree Model Property 25

3 Decision Procedures **27**

3.1 Tableau 27

3.2 Procedure 30

3.3 Soundness 32

3.4 Completeness 34

3.5 Discussion on Complexity 42

3.6 An Experimental Implementation 43
3.7 Related Work 45

4 Application to Shape Analysis **46**
4.1 Pointer Structure 46
4.2 Programming Language 47
4.3 Specifications 52
4.3.1 Heap Status 52
4.3.2 Specification Formulae 53
4.4 Weakest Preconditions 54
4.5 Abstract Structure 60
4.5.1 Predicate Abstraction 60
4.6 Steps of Analysis 62
4.7 Implementation 63
4.7.1 MLAT 63
4.7.2 Verification Examples 65
4.8 Related Work 66

5 Verification with Manually Constructed Abstraction **67**
5.1 Deutsch-Schorr-Waite Marking Algorithm 67
5.1.1 Algorithm 67
5.1.2 Specifications 69
5.1.3 Informal Correctness 71
5.2 Abstract Space 72
5.2.1 Construction of Abstract Space 72
5.2.2 Verification of the Transition System 75
5.3 Implementation 76
5.4 Related Work 76

6 Application to Cellular Automata **79**
6.1 Cellular Automata 79
6.2 Abstraction by Modal Logic Formulae 81
6.2.1 2LTL 81
6.2.2 Abstract Cells 82
6.2.3 Covering Set 83
6.2.4 Constructing a Kripke Structure 85
6.2.5 Kripke Structure for Next Generation 87

6.2.6 Analysis 88
6.3 Experimental Results 89
6.4 Related Work 90

7 Conclusion **92**
7.1 Summary 92
7.2 Future Work 92
7.2.1 Decision Procedures 92
7.2.2 Applications 93

Chapter 1

Introduction

In this chapter, we describe the background of and motivation for the research, present an outline of the thesis, and summarize our contributions.

1.1 Background and Motivation

There have been remarkable developments in the application of formal methods to system verification problems. One of the representative techniques is model checking. Target systems are regarded as state transition systems and their properties are represented in temporal formulae; effective complete enumeration algorithms ascertain whether the transition systems satisfy the properties. The technique can be applied to systems regarded as finite state transition systems; however, it is difficult to apply it to infinite systems.

Abstraction [13] is a key concept to overcome the difficulties. It is formulated as a relation, called abstraction relation, between a concrete state transition system, (which is possibly infinite,) and an abstract state transition system (which is finite). Further, the relation preserves the properties that must be verified. For this technique to be applied, the abstraction relation and abstract transition system must be systematically and effectively generated from the given concrete transition systems that naturally model the target system to be verified.

Among the problem domains to which abstraction has been applied is the verification of the properties of programs written in procedural programming languages such as C or Java. A technique called predicate abstraction [55, 20] has effectively been used in this domain. Various verification systems [4, 35, 24] have been proposed based on this technique. In such systems, predicates for abstraction are expressed in the first-order predicate logic (FOL) so that they can be handled by automatic provers of

the FOL [21, 54, 49]. However, it is difficult for them to verify the properties of data structures built with pointers. This is mainly because frequently appearing properties, for example, "an object is reachable from another object by following pointers," cannot be expressed in the FOL.

Various frameworks have been proposed to address the problem and to analyze programs manipulating pointers, i.e., to perform shape analysis. In one of the approaches a three-valued logic is used in addition to the FOL enhanced with an operator to take the transitive closure [57]. Another approach uses Separation Logic [23], which is an extension of the Hoare logic and has operators to handle the status of the heap. Both logics are regarded as extensions of the FOL.

For shape analysis, we use modal logics, which are regarded as restrictions on the FOL. Their satisfiability problems, which are in most cases decidable (unlike those of the FOL), have been studied and applied to several areas. For example, they have been used to synthesize a concurrent program from a specification expressed as a temporal formula by checking the satisfiability of the formula [26, 45].

The applications of the satisfiability problem of modal logics that are directly related to our study are the analyses of graph rewriting systems [33, 63, 71, 30]. The analyses utilize the fact that the basic properties of graphs, such as reachability or relationship between adjacent nodes, are expressed using formulae in modal logics. The problem of whether one state of the system is reachable from another reduces to the satisfiability problem of a modal logic formula.

Our approach to shape analysis is to abide by the predicate abstraction technique and use modal logics to express the predicates for abstraction. Since we expect that the satisfiability problem of the logic we use is decidable, the analysis will be feasible if we develop efficient decision procedures.

With the background described above, we set the objectives of the study as follows:

- To provide implementable decision procedures for modal logics that are appropriate to analyze programs that manipulate pointers.
- To show the feasibility of the application by analyzing systems by using decision procedures for modal logics.

1.2 Overview

1.2.1 Dimensions to Extend the Modal Logic

The weakest modal logic is called the logic **K**. It is well known that the satisfiability problem of the logic is decidable and efficient implementations have been reported [50, 51]. Unfortunately, however, its expressive power is not sufficiently strong. For example, the typical property appearing in verification problems on graph systems "a node is reachable from another node by following edges" cannot be expressed in this logic. We will introduce the following four extensions to the logic.

First, fixed-point operators μ and ν are considered. The logic with the operators is called the modal μ-calculus [41]. In this logic, various properties including the abovementioned reachability property can be expressed. Although the satisfiability of the modal μ-calculus is known to be decidable [28], its known decision procedures are very complex because of the possible nestings of the fixed-point operators. To reduce the complexity, we introduce a restriction; roughly speaking, we restrict ourselves to the formulae in which no nesting of fixed-point operators occurs. The resulting logic is called the alternation-free modal μ-calculus. It can still express various useful properties for verification. For example, the Computational Tree Logic (CTL), which is commonly used to express properties in model checking, is a sublogic of the alternation-free modal μ-calculus. On the other hand, we can develop a reasonably efficient decision procedure for the logic.

Second, we introduce nominals [5]. A nominal is a type of atomic formula. Unlike an ordinary atomic formula, which we term a propositional symbol, a nominal is satisfied by one and only one node in a Kripke structure. Logics that have both propositional symbols and nominals are called hybrid logics. In applications to the analysis of programs manipulating pointers, we regard the heap as a Kripke structure, and each object is a state of the Kripke structure. A nominal is given for each pointer-type program variable, and when a state satisfies a nominal, we consider that the object corresponding to the state is pointed to by the program variable corresponding to the nominal. Here, it is natural to use a nominal and not a propositional symbol since the program variable points to only one object.

Let us state here an important property of modal logics. A logic is said to have the tree property if any satisfiable formula of the logic is satisfied by a tree model. Various modal logics, including the logic **K** have this property and a natural decision procedure is often constructed on the basis of this property. All modal logics used in the study described above have this property. However, we cannot expect our modal logics to

have the tree model property. The systems to which we want to apply the logics handle data structures built with pointers, and such structures can contain shapes other than trees, such as DAGs or cycles.

It is the nominals that break the tree model property; using nominals we can express the conditions of a Kripke structure, for example, "it contains a cycle" or "two nodes have common successors". Therefore, introducing nominals is a key issue for the application of modal logics to shape analysis in our study. It is also a source of difficulty for building decision procedures since we must find a way to build models other than tree structures.

Third, we introduce reverse modalities [74]. For the modality m and formula φ, $\langle m \rangle \varphi$ expresses that there exists a successor node that satisfies φ. The reverse modality $\overline{m}$ of m is a modality such that $\langle \overline{m} \rangle \varphi$ expresses that there exists a predecessor node that satisfies φ. Reverse modalities allows us to handle both directions equally: we can state "node a is backward reachable from b" in the same manner as "node a is forward reachable from b". It is known that the expressive power of a logic with reverse modalities is strictly stronger than that of a logic without them. Moreover, we need reverse modalities to express the weakest preconditions [22] of pointer operations, which play an important role in the application of logics to program analysis.

Fourth, we introduce functional Kripke structures: these are Kripke structures whose transition relation is a function, i.e., each state has one successor. Very often in an application, we consider that the target system can be modeled with relations such that each node has one successor. A typical example is a singly linked list. A doubly linked list also falls into this category since we consider the forward and backward links as two different relations. In such cases, we are interested in whether a formula is satisfied by a functional Kripke structure and not in the Kripke structure in the ordinary sense. Problems concerning a given formula being satisfied by a functional Kripke structure are regarded as an extension to the satisfiability problem since we have restrictions on the models.

1.2.2 Satisfiability and Decision Procedures

The extensions described above can be independently introduced to the logic **K**. In applications, we should appropriately combine them according to the nature of the target system. Since we use decision procedures for judging the satisfiability of formulae, a basic premise for the application of a logic is that its satisfiability problem is decidable. Moreover, the decision procedure should be efficient for practical

applications.

From this point of view, not all combinations of the extensions are applicable since the satisfiability problem of the logic with all the abovementioned four extensions, i.e., (i) alternation-free fixed-point operators, (ii) nominals, (iii) reverse modalities, and (iv) functional Kripke structures, is known to be undecidable [8].

A natural question would be whether the logics that have three extensions out of the four are applicable. The answer is yes. The satisfiability problem of any such combination, i.e., (a) (ii)+(iii)+(iv), (b) (i)+(iii)+(iv), (c) (i)+(ii)+(iv), and (d) (i)+(ii)+(iii), is known to be decidable [70, 59, 7].

Combination (a) is not very interesting because it has poor expressive power for our application since it does not have fixed-point operators at all. The known decision procedures for the remaining logics (b), (c), and (d) are general (they can be applied to non alternation-free formulae) but very complex. They contain the determinization of automata for infinite words [56] and the calculation of the winning regions of parity games [48], and to the best of our knowledge, such procedures have not been implemented.

We build decision procedures that utilize the alternation-freeness of the fixed-point operators for (b), (c), and (d) so that efficient implementations are possible. Our decision procedures need less computation time in theory, and we actually implement them using binary decision diagrams (BDDs) [10]. The effectiveness of the use of BDDs in verification tools are well known; some classical examples are model checking tools such as SMV [46]. BDDs are also applied for satisfiability checking for logics, for example the logic **K** [50, 51] or WS2S [34]. Our decision procedures are also suitable for implementation along with the BDDs since they consist of iterations of set operations on finite sets. They are sufficiently efficient to enable applications to program analysis, as described below.

1.2.3 Applications

As applications to verification problems, we apply the decision procedures of modal logics to the following two domains.

The first application is "shape analysis using modal logics", that is, we verify the properties of programs that manipulate pointers by using the predicate abstraction technique. First, we define a small programming language called PML. It has pointer manipulation features extracted from C or Java. Subsequently, we show that we can calculate the weakest preconditions of the predicate expressed by the modal logic for-

mulae for the statements of the language. Combining this calculation and the decision procedures for the logics described above, an abstract transition system is constructed for a given program; this transition system is a sound abstraction with regard to the properties expressed in a property description language based on LTL, which we shall also define. We implement a prototype system called MLAT that automatically builds the abstract transition system from a given program and predicates and calls a model checker to verify the given properties.

We also implement a system that verifies the correctness of manually constructed abstract transition systems. With this system, the user can verify the properties of programs written in PML. We illustrate the effectiveness of the system by verifying the safety properties of an algorithm for garbage collection, the Deutsch-Schorr-Waite (DSW) marking algorithm [60].

The second application is the analysis of one-dimensional cellular automata. A one-dimensional cellular automaton is a system that consists of infinite nodes (called cells) in a line. The state of each node is either "zero" or "one" and repeatedly changes its state according to a rule expressed in terms of its state and the states of two adjacent cells. We use a logic called 2LTL, which is a sublogic of the two-way modal μ-calculus. We present a method to generate finite state transition systems by using the 2LTL logic. An experimental implementation is available.

1.3 Contributions

The first contribution of the study is to provide implementable decision procedures to check satisfiability for several modal logics and prove their correctness. These modal logics are extensions of the minimal modal logic **K** and the extensions are combinations of factors that are important for applying them to shape analysis. Since the logic with all four extensions is undecidable, the logics in our study, which have three extensions out of four, are optimal. We provide decision procedures for three combinations of extensions that are useful in applications.

The second contribution is to present a procedure for determining the weakest preconditions of the basic operations of pointer manipulating programs by using a modal logic with our extensions. The existence of the weakest preconditions is not trivially clear: we can prove that the weakest preconditions cannot be expressed in some modal logics. This fact suggests that our extensions are natural for applications to shape analysis.

The third contribution is to present a general framework for verifying the prop-

erties of programs manipulating pointers and implement verification tools based on it. Although there are various existing frameworks and tools for similar purposes, our framework can be used to verify both simple and complex properties automatically and manually, respectively; in the latter case, human reasoning can be naturally translated into expressions for the tool. With these implementations, we establish the feasibility of our proposed verification method.

1.4 Organization

The rest of the thesis is organized as follows.

In Chapter 2, we define the syntax and semantics of modal languages and formalize all the extensions discussed in this chapter. We also prepare several lemmas that are needed in the following chapters.

The satisfiability and decision procedures for the logics are discussed in Chapter 3. We first show that logic (a) in the previous section is decidable by proving that it has the small model property. Subsequently, we state the decision procedures for (b), (c), and (d), prove their correctness, and report an experimental implementation.

The following three chapters handle the applications of the satisfiability judgment of modal logics. In Chapter 4, we discuss a method for automatically verifying the properties of programs manipulating pointers; this method is based on the predicate abstraction technique. Experimental results obtained by using a prototype system called MLAT are reported. In Chapter 5, we illustrate the manual construction of a transition system for verification of complex properties by using the DSW algorithm. In Chapter 6, one-dimensional cellular automata are analyzed by using modal logics.

Finally, we summarize the thesis and discuss future work in Chapter 7.

Chapter 2

Modal Logics

In this chapter we introduce the syntax and semantics of modal logics including the extensions to the minimal modal logic. We also give some lemmas that we will use in later chapters.

2.1 The Modal Language

2.1.1 Syntax

We will define a language whose expression power is strong enough to develop the theories in the thesis and call it the *modal language*. The language depends on three parameters. The first is the set of *propositional symbols*, denoted by PS. Propositional symbols are typically referred to by symbols $p, q, \ldots$. The second is the set of *modality symbols*, denoted by MS. Each modality symbol is typically referred to by symbol m. The third is the set of *nominals*, denoted by Nom. Each nominal is typically referred to by symbol n. The cardinality of PS, MS and Nom can be anything, though in practical applications they are all finite.

For each modality symbol m, we consider expression $\overline{m}$ and call it the *reverse modality* or *backward modality* of m. The set of forward and backward modalities is denoted by Mod: $\mathrm{Mod(MS)} = \mathrm{MS} \cup \{\overline{m} \mid m \in \mathrm{MS}\}$. We write Mod instead of Mod(MS) if no confusion occurs. We define $\overline{\overline{m}} = m$ so that $\overline{m}$ can be considered for any modality $m \in \mathrm{Mod}$.

We also prepare countably many propositional variables $X_0, X_1, \ldots$ and denote by PV the set of propositional variables.

In the rest of the chapter, we fix PS, MS, and Nom.

Definition 2.1 The *least modal language* L_0 is the least set L that satisfies the fol-

lowing.

(1) $\mathrm{PS} \subseteq L$.

(2) $\varphi \in L \implies \neg\varphi \in L$.

(3) $\varphi_1, \varphi_2 \in L \implies \varphi_1 \vee \varphi_2,\ \varphi_1 \wedge \varphi_2 \in L$.

(4) $\varphi \in L,\ m \in \mathrm{MS} \implies \langle m \rangle \varphi,\ [m]\varphi \in L$.

Language $\mathrm{L}_0(\mu)$ is the least language that satisfies the following as well as from (1) to (4).

(5) $\mathrm{PV} \subseteq L$.

(6) If $\varphi \in L$, $X \in \mathrm{PV}$ and all free occurrences of X in φ are positive, then $\mu X \varphi$, $\nu X \varphi \in L$.

where an occurrence of propositional variable X in φ is *free* if it is not in the scope of μX or νX and it is *positive* if the number of negation symbols ($\neg$) whose scope contain the occurrence is even.

Language $\mathrm{L}_0(@)$ is the least language that satisfies the following as well as from (1) to (4).

(7) $\mathrm{Nom} \subseteq L$.

(8) $n \in \mathrm{Nom},\ \varphi \in L \implies @n\,\varphi \in L$

Language $\mathrm{L}_0(\bar{\ })$ is the least language that satisfies from (1) to (3) and the following.

(9) $\varphi \in L,\ m \in \mathrm{Mod} \implies \langle m \rangle \varphi,\ [m]\varphi \in L$. ∎

Language $\mathrm{L}_0(\mu, @)$ is the least language that satisfies from (1) to (8). Other combinations are defined similarly.

We denote by $\mathrm{free}(\varphi)$ the set of free propositional variables occurred in φ.

Standard abbreviations will be used such as $\varphi \to \psi \stackrel{\mathrm{def}}{=} \neg\varphi \vee \psi$, $\varphi \leftrightarrow \psi \stackrel{\mathrm{def}}{=} (\varphi \to \psi) \wedge (\psi \to \varphi)$, $\mathbf{true} \stackrel{\mathrm{def}}{=} p \vee \neg p$ for an arbitrarily chosen $p \in \mathrm{PS}$, and $\mathbf{false} \stackrel{\mathrm{def}}{=} \neg\mathbf{true}$. We also introduce a ternary operator "$\cdot \to \cdot\,;\cdot$". Its definition is $\varphi \to \psi\,;\chi \stackrel{\mathrm{def}}{=} (\varphi \wedge \psi) \vee (\neg\varphi \wedge \chi)$. It can easily be shown that $\varphi \to \psi\,;\chi \equiv (\varphi \to \psi) \wedge (\neg\varphi \to \chi)$.

A formula is said to be *closed* if it does not contain free occurrences of propositional variables.

2.1.2 Semantics

Semantics of modal logics are defined in terms of *Kripke structures*.

Definition 2.2 A *Kripke structure* is a triple $\mathcal{K} = (S, R, \lambda)$. S is a set and its elements are called *states*. R is a function whose domain contains all $m \in \mathrm{MS}$ and $R(m)$ is a relation called a *transition relation* on S, namely $R(m) \subseteq \mathscr{P}(S \times S)$. $\lambda : \mathrm{PS} \cup \mathrm{Nom} \to \mathscr{P}(S)$ is a function called the *labeling function*, and $\lambda(n)$ is a singleton for $n \in \mathrm{Nom}$. ∎

A Kripke structure is *total* if for any $m \in \mathrm{MS}$ and $s \in S$ there is $s' \in S$ such that $(s, s') \in R(m)$.

When $\mathcal{K} = (S, R, \lambda)$ is a Kripke structure, we denote S by $|\mathcal{K}|$.

A Kripke structure designates a transition relation $R(m)$ for each $m \in \mathrm{MS}$, and we extend it so that it is defined on Mod: $R(\overline{m}) = R(m)^{-1} = \{(s', s) \in S \times S \mid (s, s') \in R(m)\}$.

For $n \in \mathrm{Nom}$, $\lambda(n)$ is a singleton. We denote by $\tilde{\lambda}(n)$ the member of the singleton: $\lambda(n) = \{\tilde{\lambda}(n)\}$.

Before defining the satisfiability relation we introduce valuations for Kripke structures.

Definition 2.3 Let L be a modal language and $\mathcal{K} = (S, R, \lambda)$ be a Kripke structure for L. A valuation for $\mathcal{K}$ is a function $v : \mathrm{PV} \to \mathscr{P}(S)$. ∎

For a function f, we denote by $f[a \mapsto b]$ a function g whose domain is $\mathrm{dom}(f) \cup \{a\}$, $g(a) = b$ and $g(x) = f(x)$ for $x \in \mathrm{dom}(f) \setminus \{a\}$. We define the satisfiability relation as follows:

Definition 2.4 For formula φ, Kripke structure $\mathcal{K} = (S, R, \lambda)$, valuation v, and $s \in S$, we define relation $\mathcal{K}, s \models \varphi$. If no confusion occurs, we omit the references to $\mathcal{K}$ or v and write it as $v, s \models \varphi$ or $s \models \varphi$.

- For $x \in \mathrm{PS} \cup \mathrm{Nom}$, $s \models x \iff s \in \lambda(x)$.
- For $X \in \mathrm{PV}$, $v, s \models X \iff s \in v(X)$.
- $s \models \neg\varphi$ holds if and only if $s \models \varphi$ does not hold.
- $s \models \varphi_1 \vee \varphi_2 \iff s \models \varphi_1$ or $s \models \varphi_2$.
- $s \models \varphi_1 \wedge \varphi_2 \iff s \models \varphi_1$ and $s \models \varphi_2$.

- $s \models \langle m \rangle \varphi \iff$ There exists $s' \in S$ such that $(s, s') \in R(m)$ and $s' \models \varphi$.
- $s \models [m]\varphi \iff$ For any $s' \in S$, $(s, s') \in R(m)$ implies $s' \models \varphi$.
- $v, s \models \mu X \varphi \iff$
  $s \in \bigcap \{T \subseteq S \mid \{t \in S \mid v[X \mapsto T], t \models \varphi\} \subseteq T\}$
- $v, s \models \nu X \varphi \iff$
  $s \in \bigcup \{T \subseteq S \mid T \subseteq \{t \in S \mid v[X \mapsto T], t \models \varphi\}\}$
- $s \models @n\, \varphi \iff \tilde{\lambda}(n) \models \varphi$ ∎

We denote the set $\{s \in S \mid \mathcal{K}, v, s \models \varphi\}$ by $[\![\varphi]\!]^{\mathcal{K},v}$; or $[\![\varphi]\!]^v$ or $[\![\varphi]\!]$ for short. Note that $[\![@n\, \varphi]\!]$ equals to either S itself or the empty set.

If a formula φ is closed, whether $\mathcal{K}, v, s \models \varphi$ holds or not does not depend on the valuation v. A closed formula φ is said to be *valid* if for any Kripke structure $\mathcal{K}$ and for any state $s \in |\mathcal{K}|$, $\mathcal{K}, s \models \varphi$ holds. It is *satisfiable* if there is a Kripke structure $\mathcal{K}$ and $s \in |\mathcal{K}|$ such that $\mathcal{K}, s \models \varphi$. Thus a formula is valid if and only if its negation is not satisfiable.

The satisfiability problem for a language L is the problem to decide whether a given formula $\varphi \in L$ is satisfiable. The satisfiability problem for a logic is the satisfiability problem for the language of the logic. We write $\mathrm{Sat}(\mu)$ for the satisfiability problem for $\mathrm{L}_0(\mu)$. Similar notations are used for other languages as well.

It is well known that a formula φ is valid if and only if it is a theorem of a deduction system with finite axioms called **K**, which is the minimal modal logic [6]. The satisfiability problem of the logic **K**, i.e., Sat(), is known to be decidable. There are efficient decision procedures and their implementations are available [50, 51].

Two formulae φ and ψ are said to be *equivalent* if for any Kripke structure $\mathcal{K}$, valuation v, and $s \in |\mathcal{K}|$, $\mathcal{K}, v, s \models \varphi$ holds if and only if $\mathcal{K}, v, s \models \psi$ holds. We write $\varphi \equiv \psi$ when φ and ψ are equivalent.

A formula φ is said to be in *positive normal form*, or *PNF*, if φ satisfies the following conditions:

- All negation symbols ($\neg$) in φ appear immediately before propositional symbols or nominals.
- All propositional variables in φ are bound at most once.

It is clear that for every formula φ there is a formula in PNF that is equivalent to φ. (The second condition can be satisfied by performing the alpha conversions.)

Simulations between Kripke structures are defined as follows:

Definition 2.5 Let $\mathcal{K}_1 = (S_1, R_1, \lambda_1)$ and $\mathcal{K}_2 = (S_2, R_2, \lambda_2)$ be Kripke structures. A relation $h \subseteq S_1 \times S_2$ is a *simulation* from $\mathcal{K}_1$ to $\mathcal{K}_2$ if the following conditions are satisfied.

- For any $s_1 \in S_1$ there is $s_2 \in S_2$ such that $(s_1, s_2) \in h$.
- For any $m \in \mathrm{MS}$, $s_1, s_1' \in S_1$ and $s_2 \in S_2$, if $(s_1, s_2) \in h$ and $(s_1, s_1') \in R_1(m)$, then there is $s_2' \in S_2$ such that $(s_1', s_2') \in h$ and $(s_2, s_2') \in R_2(m)$.
- If the language contains the reverse modality, that is, $\mathrm{L}_0(^{-}) \subseteq L$, the same condition as above for $\overline{m}$ is satisfied. That is, for any $m \in \mathrm{MS}$, $s_1, s_1' \in S_1$ and $s_2 \in S_2$, if $(s_1, s_2) \in h$ and $(s_1', s_1) \in R_1(m)$, then there is $s_2' \in S_2$ such that $(s_1', s_2') \in h$ and $(s_2', s_2) \in R_2(m)$.
- Assume $p \in \mathrm{PS}$, $s_1 \in S_1$, $s_2 \in S_2$, and $(s_1, s_2) \in h$. Then $s_1 \in \lambda_1(p)$ if and only if $s_2 \in \lambda_2(p)$.

Kripke structure $\mathcal{K}_2$ is an *approximation* of $\mathcal{K}_1$, or $\mathcal{K}_2$ *approximates* $\mathcal{K}_1$, if there is a simulation from $\mathcal{K}_1$ to $\mathcal{K}_2$. We write $\mathcal{K}_1 \leq \mathcal{K}_2$ when $\mathcal{K}_2$ approximates $\mathcal{K}_1$. ∎

2.2 Fixed-point Operators

2.2.1 The Modal μ-calculus

For a language L, we denote by $L(\mu)$ the least language that includes L and has fixed-point operators. Language $\mathrm{L}_0(\mu)$ is called the language of the *modal μ-calculus*.

An abbreviation: $\nu X\varphi \stackrel{\mathrm{def}}{=} \neg\mu X\neg\varphi[\neg X/X]$, where $\varphi[\psi/X]$ is the formula constructed from φ by replacing all free occurrences of X with ψ.

We denote by On the class of all ordinal numbers. The following facts are well-known.

Lemma 2.6 *([41])* Let $\mathcal{K} = (S, R, \lambda)$ be a Kripke structure, v a valuation for $\mathcal{K}$, $s \in S$ and φ a formula.

(1) For $\alpha \in \mathrm{On}$, we define $T_\alpha \subseteq S$ as follows:

- $T_0 = \varnothing$.
- $T_{\alpha+1} = \{s \in S \mid v[X \mapsto T_\alpha], s \models \varphi\}$.
- $T_\alpha = \bigcup_{\beta<\alpha} T_\beta$ if α is limit.

Then there is $\alpha_0 \in \mathrm{On}$ such that $T_\alpha = \{s \in S \mid v, s \models \mu X\varphi\}$ holds for any $\alpha \geq \alpha_0$.

(2) For $\alpha \in \mathrm{On}$, we define $T_\alpha \subseteq S$ as follows:

- $T_0 = S$.
- $T_{\alpha+1} = \{s \in S \mid v[X \mapsto T_\alpha], s \models \varphi\}$.
- $T_\alpha = \bigcap_{\beta<\alpha} T_\beta$ if α is limit.

Then there is $\alpha_0 \in \mathrm{On}$ such that $T_\alpha = \{s \in S \mid v, s \models \nu X\varphi\}$ holds for any $\alpha \geq \alpha_0$.

∎

In the rest of this section, symbol λ is used to express either μ or ν. Thus $\lambda X\varphi$ is either $\mu X\varphi$ or $\nu X\varphi$. For formula $\lambda X\varphi$, we denote by $\exp(\lambda X\varphi)$ the formula $\varphi[\lambda X\varphi/X]$, the formula obtained from φ by replacing all free occurrences of X with $\lambda X\varphi$, and call it the *expansion* of $\lambda X\varphi$. For example, $\exp(\mu X(p \vee \langle m\rangle X)) = p \vee \langle m\rangle(\mu X(p \vee \langle m\rangle X))$.

Lemma 2.7 $\exp(\lambda X\varphi) \equiv \lambda X\varphi$.

Proof Let $(T_\alpha \mid \alpha \in \mathrm{On})$ be the sequence in Lemma 2.6 for $\lambda X\varphi$ and $(T'_\alpha \mid \alpha \in \mathrm{On})$ for $\exp(\lambda X\varphi)$. Then it is easy to show $T_{\alpha+1} = T'_\alpha$ by induction on α. ∎

We define relation E on the set of all formulae in PNF as the least relation that satisfies the following:

- $(\varphi_1 \vee \varphi_2, \varphi_1) \in E$, $(\varphi_1 \vee \varphi_2, \varphi_2) \in E$.
- $(\varphi_1 \wedge \varphi_2, \varphi_1) \in E$, $(\varphi_1 \wedge \varphi_2, \varphi_2) \in E$.
- $(\langle m\rangle\varphi, \varphi) \in E$.
- $([m]\varphi, \varphi) \in E$.
- $(\lambda X\varphi, \exp(\lambda X\varphi)) \in E$.
- $(@n\,\varphi, n) \in E$, $(@n\,\varphi, \varphi) \in E$.

E is almost the (direct) subformula-relation, except for formulas in the form of $\lambda X\varphi$.

Definition 2.8

(1) For a closed formula φ, the *closure* of φ is the least set of formulae that contains φ and is closed under the relation E. We denote by $\mathrm{cl}(\varphi)$ the closure of φ.

(2) For closed formula φ in PNF, the *lean* of φ is a subset of $\mathrm{cl}(\varphi)$ defined as follows: $\{\psi \in \mathrm{cl}(\varphi) \mid \psi \in \mathrm{PS} \cup \mathrm{Nom}$ or ψ is in the form of $\langle m \rangle \chi$ or $[m]\chi\}$. ∎

Thus, the following holds:

- $\varphi \in \mathrm{cl}(\varphi)$.
- $\psi \in \mathrm{cl}(\varphi)$, $(\psi, \chi) \in E \implies \chi \in \mathrm{cl}(\varphi)$.

Lemma 2.9 *([41])* For closed formula φ, there is one to one corresponding between the set of subformulae of φ and $\mathrm{cl}(\varphi)$, by mapping a subformula ψ of φ to formula $\psi[\chi_1/X_1][\chi_2/X_2]\cdots[\chi_k, /X_k]$, where $X_1, \ldots, X_k$ is an enumeration of the free propositional variables in ψ, χ_i is the subformula of φ in the form of $\mu X_i \chi_i'$, arranged in the manner that if χ_i is a subformula of χ_j then $i \leq j$. In particular, $\mathrm{cl}(\varphi)$ is a finite set. ∎

For example, for $\varphi = \mu X(p \vee \nu Y(\langle m_1 \rangle X \wedge [m_2]Y))$ with $p \in \mathrm{PS}$, let $\varphi_1 = \exp(\varphi) = p \vee \nu Y(\langle m_1 \rangle \varphi \wedge [m_2]Y)$, $\chi = \nu Y(\langle m_1 \rangle \varphi \wedge [m_2]Y)$, $\chi_1 = \exp(\chi) = \langle m_1 \rangle \varphi \wedge [m_2]\chi$. Then $\mathrm{cl}(\varphi) = \{\varphi, \varphi_1, p, \chi, \chi_1, \langle m_1 \rangle \varphi, [m_2]\chi\}$ and the lean of φ is $\{p, \langle m_1 \rangle \varphi, [m_2]\psi\}$. For example, χ_1 is the corresponding formula of $\langle m_1 \rangle X \wedge [m_2]Y$ in $\mathrm{cl}(\varphi)$ and it is calculated as follows:

$$
\begin{aligned}
& (\langle m_1 \rangle X \wedge [m_2]Y)[\chi/Y][\varphi/X] \\
= \ & (\langle m_1 \rangle X \wedge [m_2](\nu Y(\langle m_1 \rangle X \wedge [m_2]Y)))[\varphi/X] \\
= \ & \langle m_1 \rangle \varphi \wedge [m_2](\nu Y(\langle m_1 \rangle \varphi \wedge [m_2]Y)) \\
= \ & \langle m_1 \rangle \varphi \wedge [m_2]\chi \\
= \ & \chi_1
\end{aligned}
$$

Definition 2.10 Let φ be a closed formula. An occurrence of propositional variable X in φ is *guarded* if there is a formula ψ such that

- ψ contains the occurence.
- ψ is in the form of $\langle m \rangle \psi'$, $[m]\psi'$, or $@n\,\psi'$.
- Let $\lambda X \chi$ be the subformula of φ that binds the occurence. Then ψ is a subformula of χ.

A formula is *guarded* if all occurrences of propositional variables are guarded. ∎

For example, $\mu X(p \vee \langle m \rangle X \vee @n\,X)$ is guarded, but $\mu X(p \vee X)$ is not guarded.

Lemma 2.11 *(Kozen)* For every formula φ there is a guarded formula ψ in PNF such that $\varphi \equiv \psi$. ∎

For example, from an unguarded formula $\mu X(p \vee \langle m \rangle X \vee \nu Y(X \wedge Y \wedge [m]Y))$, we can obtain an equivalent guarded formula in the following way:

$$\begin{aligned}
& \mu X(p \vee \langle m \rangle \vee \nu Y((X \vee q) \wedge Y \wedge [m]Y)) \\
\equiv\ & \mu X(p \vee \langle m \rangle \vee \nu Y((X \vee q) \wedge \mathbf{true} \wedge [m]Y)) \\
\equiv\ & \mu X(p \vee \langle m \rangle \vee \nu Y((X \vee q) \wedge [m]Y)) \\
\equiv\ & \mu X(p \vee \langle m \rangle \vee \nu Y((X \vee q) \wedge [m]Y)) \\
\equiv\ & \mu X(p \vee \langle m \rangle \vee ((X \vee q) \wedge [m]\nu Y((X \vee q) \wedge [m]Y))) \\
\equiv\ & \mu X(p \vee \langle m \rangle \vee ((\mathbf{false} \vee q) \wedge [m]\nu Y((X \vee q) \wedge [m]Y))) \\
\equiv\ & \mu X(p \vee \langle m \rangle \vee (q \wedge [m]\nu Y((X \vee q) \wedge [m]Y)))
\end{aligned}$$

Refer to [76] for a proof of Lemma 2.11.

Let φ_{I} be a closed guarded formula in PNF and Lean be the lean of φ_{I}. Since it is guarded, the relation $\{(\varphi, \psi) \in E \mid \varphi \in \mathrm{cl}(\varphi_{\mathrm{I}}) \setminus \mathrm{Lean},\ \psi \in \mathrm{cl}(\varphi_{\mathrm{I}})\}$ on $\mathrm{cl}(\varphi_{\mathrm{I}})$ is well-founded. Therefore to prove a proposition for all formulae in $\mathrm{cl}(\varphi_{\mathrm{I}})$, it is enough to show the proposition:

- (base cases) for formulae $\varphi \in$ Lean and for formulae in the form of $\neg\psi \in \mathrm{cl}(\varphi_{\mathrm{I}})$ where $\psi \in$ Lean
- (induction) for formulae φ in $\mathrm{cl}(\varphi_{\mathrm{I}})$ in the form of $\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2$, $\lambda X \varphi_3$, and $@n\ \varphi_1$ assuming that the proposition for φ_1, φ_2, and $\exp(\lambda X \varphi_3)$ has already been proved.

We call this type of argument a proof by *induction based on the lean*. Definitions by induction based on the lean are also possible.

2.2.2 Parity Games

It is well-known that the model-checking for a formula in the modal μ-calculus can be done using a parity game [80]. Although our decision procedures do not directly refer such games, we can simplify some of the proofs of lemmas that we need in later chapters by using them. In this section we briefly review parity games and their relevance to the modal μ-calculus.

For function f and subset S of its domain $\mathrm{dom}(f)$, we denote by $f \restriction S$ the restriction f to S, that is, $f \restriction S$ is a function, its domain is S, and $(f \restriction S)(s) = f(s)$ for $s \in S$.

An *arena* for a *parity game* is a tuple (V_0, V_1, C, Ω) such that V_0 and V_1 are disjoint sets, $C \subseteq (V_0 \cup V_1) \times (V_0 \cup V_1)$ is a relation, and $\Omega : V \to \omega$ is a function, called the *priority function*, such that $\Omega[V]$ is a finite set, where $V = V_0 \cup V_1$. Elements of V_0 and V_1 are called *0-vertices* and *1-vertices* respectively. A vertex v is called a dead end if there is no $v' \in V$ such $(v, v') \in C$.

Games are played by *Player 0* and *Player 1*. A *pre-trace* τ is a function from a natural number $n \in \omega \setminus \{0\}$ to V satisfying $(\tau(i), \tau(i+1)) \in C$ for any i such that $i + 1 \in n$. A functoin τ is a *trace* if either of the following conditions holds:

- τ is a pre-trace and $\tau(\mathrm{dom}(\tau) - 1)$ is a dead end, or
- τ is a function from ω to V and $\tau \restriction n$ is a pre-trace for all $n \in \omega$.

The *winner* of a trace τ is Player 0 if

- $\mathrm{dom}(\tau) = n \in \omega$ and $\tau(n-1) \in V_1$, or
- $\mathrm{dom}(\tau) = \omega$ and $\max\{n \mid \{i \in \omega \mid \Omega(\tau(i)) = n\}$ is an infinite set$\}$ is an even number.

Otherwise the winner of the trace is Player 1.

Let T be the set of all pre-traces. A *strategy* σ of Player ι, where ι is either 0 or 1, is a partial function from $\{\tau \in T \mid \tau(\mathrm{dom}(\tau) - 1) \in V_\iota\}$ to V such that $(\tau(\mathrm{dom}(\tau) - 1), \sigma(\tau)) \in V$ for $\tau \in \mathrm{dom}(\sigma)$. Strategy σ is called *memoryless* if there is a partial function $f : V \to V$ such that $\mathrm{dom}(\sigma) = \{\tau \in T \mid \tau(\mathrm{dom}(\tau) - 1) \in \mathrm{dom}(f)\}$ and $\sigma(\tau) = f(\tau(\mathrm{dom}(\tau) - 1))$ for all $\tau \in \mathrm{dom}(\sigma)$. A trace τ *conforms* strategy σ of Player ι if $\sigma(\tau \restriction n) = \tau(n)$ for all $n \in \mathrm{dom}(\tau) \setminus \{0\}$ such that $\tau(n-1) \in V_\iota$. For $v \in V$, strategy σ_0 of Player 0 and strategy σ_1 of Player 1 there is at most one trace τ that conforms both σ_0 and σ_1 and $\tau(0) = v$. We denote this trace, if it exists, by $(\sigma_0 * \sigma_1)(v)$ or $(\sigma_1 * \sigma_0)(v)$. Strategy σ of Player ι is a *winning strategy* on $v \in V$ if for any strategy σ' of Player $1 - \iota$, $(\sigma * \sigma')(v)$ exists and its winner is Player ι. The subset W of all the vertices on which there is a winning strategy of Player ι is called the *winning region* of Player ι.

Let φ_{I} be a closed formula in PNF and $\mathcal{K} = (S, R, \lambda)$ be a Kripke structure. We define an arena $\mathcal{A}(\varphi_{\mathrm{I}}, \mathcal{K}) = (V_0, V_1, C, \Omega)$ as follows:

- $V = \mathrm{cl}(\varphi_{\mathrm{I}}) \times S$. The set V is divided into V_0 and V_1 so that the following conditions are satisfied:
  - For $p \in \mathrm{PS} \cup \mathrm{Nom}$,
    - $(p, s) \in V_0 \iff s \models p$ and $(p, s) \in V_1 \iff s \not\models p$.
    - $(\neg p, s) \in V_0 \iff s \not\models p$ and $(\neg p, s) \in V_1 \iff s \models p$.
  - $(\varphi_1 \vee \varphi_2, s) \in V_0$, $(\varphi_1 \wedge \varphi_2, s) \in V_1$.
  - $(\langle m \rangle \varphi, s) \in V_0$, $([m]\varphi, s) \in V_1$
  - $(\mu X \varphi, s) \in V_0$, $(\nu X \varphi, s) \in V_1$
  - $(@n\, X, s) \in V_0$.
- C is the least subset that satisfies the following:
  - $(\varphi, \varphi') \in E \implies ((\varphi, s), (\varphi', s)) \in C$.
  - $(s, s') \in R(m) \implies ((\langle m \rangle \varphi, s), (\varphi, s')) \in C$ and $(([m]\varphi, s), (\varphi, s')) \in C$.
  - $((@n\, \varphi, s), (\varphi, \tilde{\lambda}(n))) \in C$.

  Vertices in the form of (p, s) where $p \in \mathrm{PS} \cup \mathrm{Nom}$ are dead ends.
- $\Omega((\varphi, s)) = \Omega(\varphi)$, where $\Omega(\varphi)$ is defined as follows:[1]
  - $\Omega(\varphi) = 0$ if φ is not in the form of $\lambda X \varphi'$.
  - $\Omega(\overline{\mu X \varphi})$ is the least odd number greater or equal to $\Omega(\overline{\lambda Y \psi})$ where $\lambda Y \psi$ is a subformula of φ.
  - $\Omega(\overline{\nu X \varphi})$ is the least even number greater or equal to $\Omega(\overline{\lambda Y \psi})$ where $\lambda Y \psi$ is a subformula of φ.

The winning region of the game $\mathcal{G}$ defined by this arena consists of pairs (φ, s) such that $s \models \varphi$. More precisely the following lemma holds.

Lemma 2.12 Let $\varphi \in \mathrm{cl}(\varphi_{\mathrm{I}})$ and $s \in S$. The followings are equivalent.

(1) $\mathcal{K}, s \models \varphi$

(2) Player 0 has a winning strategy of $\mathcal{G}$ on (φ, s).

(3) Player 0 has a memoryless winning strategy of $\mathcal{G}$ on (φ, s).

Proof Refer to [80] and [42]. ∎

[1] Function Ω defined here is not optimal in the sense that $\Omega[S]$ is unnecessarily large ([80]). However, it does not lead to any inefficiency in this thesis since we concentrate on the alternation-free part of the modal μ-calculus. Therefore we employ this definition because it is simpler than the optimal one.

2.3 The Alternation of Fixed-point Operators

Definition 2.13 A formula φ is *alternation-free* if the following conditions are satisfied:

- For any subformula ψ of φ in the form of $\mu X\psi'$ and for any subformula χ of ψ' in the form of $\nu Y\chi'$, X does not occur freely in χ'.
- For any subformula ψ of φ in the form of $\nu Y\psi'$ and for any subformula χ of ψ' in the form of $\mu X\chi'$, Y does not occur freely in χ'.

■

For example $\mu X(\nu Y(p \wedge \langle m\rangle Y) \vee [m]X)$ is alternation-free, while $\mu X(\nu Y(p \wedge \langle m\rangle(X \wedge Y)) \vee [m]X)$ is not.

For a language L such that $\mathrm{L}_0(\mu) \subseteq L$, the *alternation-free part* of L is the subset of L that consists of the formulae in which all fixed-point operators are alternation-free. We denote by $\mathrm{L}_0(\mu_{\mathrm{AF}})$ the alternation free part of $\mathrm{L}_0(\mu)$. Other notations such as $\mathrm{L}_0(\mu_{\mathrm{AF}}, @)$ are defined similarly.

For formula φ in PNF, we regard $(\mathrm{cl}(\varphi), E)$ as a graph and denote by $\mathscr{D}(\varphi)$ the set of strongly connected components (SCCs) of the graph. Also we denote by $\mathscr{D}_\mu(\varphi)$ and $\mathscr{D}_\nu(\varphi)$ the set of SCCs of the graph that contains a formula in the form of $\mu X\psi$ and $\nu X\psi$, respectively. If φ is clear from the context, they are written as $\mathscr{D}_\mu$ and $\mathscr{D}_\nu$, respectively. Also we write $D_\mu = \bigcup \mathscr{D}_\mu$ and $D_\nu = \bigcup \mathscr{D}_\nu$. If $\psi \in \mathrm{cl}(\varphi)$ is an element of $\bigcup \mathscr{D}(\varphi)$, we denote by $D(\psi)$ the SCC that contains ψ.

Lemma 2.14

(1) $\mathscr{D}(\varphi) = \mathscr{D}_\mu(\varphi) \cup \mathscr{D}_\nu(\varphi)$.

(2) $\mathscr{D}_\mu(\varphi) \cap \mathscr{D}_\nu(\varphi) = \varnothing$ if and only if φ is alternation-free.

Proof (1) It is obvious that if $F \subseteq \mathrm{cl}(\varphi)$ does not contain formulae in the form of $\mu X\varphi$ or $\nu X\varphi$, there is no loop in F.
(2) For subformula ψ of φ, let us denote by $\overline{\psi}$ the formula in $\mathrm{cl}(\varphi)$ that corresponds to ψ with regard to the relation defined in Lemma 2.9. Also we denote by E^* the reflexive transitive closure of E. It is easy to see that for $(\overline{\varphi}, \overline{\psi}) \in E^*$ if and only if either ψ is a subformula of φ or there is a subformula $\chi = \lambda X\chi'$ of φ such that X occurs freely in φ and ψ is a subformula of χ.
($2\Rightarrow$) If φ is not alternation-free, there is a subformula $\psi = \mu X\psi_1$ of φ and a subformula $\chi = \nu Y\chi_1$ of ψ_1 such that X occurs freely in χ_1 (or μ and ν exchanged). Then

$(\overline{\psi}, \overline{\chi}) \in E^*$ and $(\overline{\chi}, \overline{\psi}) \in E^*$. Therefore there is an SCC that both contains $\overline{\psi}$ and $\overline{\chi}$, meaning it belongs to $\mathscr{D}_\mu \cap \mathscr{D}_\nu$.
($2\Leftarrow$) Assume $D \in \mathscr{D}_\mu \cap \mathscr{D}_\nu$. There are subformulae φ, and χ of φ such that $\overline{\psi}$, $\overline{\chi} \in D$ and the principal operators of ψ and χ are μX and νY respectively. Since D is an SCC, $(\overline{\psi}, \overline{\chi}), (\overline{\chi}, \overline{\psi}) \in E^*$. Without loss of generality we can assume that χ is a subset of ψ and X appears freely in χ, meaning that φ is not alternation-free. ∎

Let φ_{I} be a guarded alternation-free formula in PNF. Let $\mathcal{K} = (S, R, \lambda)$ be a Kripke structure and $D \in \mathscr{D}_\mu(\varphi_{\mathrm{I}})$. We define $U_\alpha \subseteq \mathrm{cl}(\varphi_{\mathrm{I}}) \times S$ for $\alpha \in \mathrm{On}$ as the least set that satisfies the following conditions:

- $U_0 \supseteq \{(\varphi, s) \mid \varphi \in \mathrm{cl}(\varphi_{\mathrm{I}}) \setminus D,\ s \in S\}$.
- If $\varphi \in \mathrm{cl}(\varphi_{\mathrm{I}})$ and one of the following holds, $(\varphi, s) \in U_\alpha$.
  - $\varphi = \varphi_1 \vee \varphi_2$ and either $(\varphi_1, s) \in U_\alpha$ or $(\varphi_2, s) \in U_\alpha$.
  - $\varphi = \varphi_1 \wedge \varphi_2$ and both $(\varphi_1, s) \in U_\alpha$ and $(\varphi_2, s) \in U_\alpha$.
  - $\varphi = \lambda X \varphi_1$ and $(\exp(\varphi), s) \in U_\alpha$.
  - $\varphi = \langle m \rangle \varphi_1$ and there is $s' \in S$ such that $(s, s') \in R(m)$ and $(\varphi_1, s') \in U_\beta$ for some $\beta < \alpha$.
  - $\varphi = [m] \varphi_1$ and for all $s' \in S$ if $(s, s') \in R(m)$ then $(\varphi_1, s') \in U_\beta$ for some $\beta < \alpha$.
  - $\varphi = @n\, \varphi_1$ and $(\varphi_1, \tilde{\lambda}(n)) \in U_\beta$ for some $\beta < \alpha$.

Lemma 2.15 $\{(\varphi, s) \in \mathrm{cl}(\varphi_{\mathrm{I}}) \times S \mid s \models \varphi\} = \bigcup_{\alpha \in \mathrm{On}} U_\alpha$.

Proof Let $U_\infty = \bigcup_{\alpha \in \mathrm{On}} U_\alpha$. It is enough to show that $(\varphi, s) \in \mathrm{cl}(\varphi_{\mathrm{I}}) \times S \setminus U_\infty \implies s \not\models \varphi$. So suppose $(\varphi, s) \in \mathrm{cl}(\varphi_{\mathrm{I}}) \times S \setminus U_\infty$.

We consider the game defined by φ_{I} and $\mathcal{K}$ on (φ, s). It is easy to see that regardless the strategy Player 0 uses, it is possible for Player 1 to keep the vertex outside of U_∞ during the play. Since $\mathrm{cl}(\varphi_{\mathrm{I}}) \times S \setminus U_\infty \subseteq D \times S$, the winner of the trace is Player 1. That is, (φ, s) belongs to the winning region of Player 1, therefore $s \not\models \varphi$. ∎

We define the rank function using $(U_\alpha \mid \alpha \in \mathrm{On})$, namely, $\mathrm{rank}_D : \{(\varphi, s) \in \mathrm{cl}(\varphi_{\mathrm{I}}) \times S \mid \mathcal{K}, s \models \varphi\} \to \mathrm{On}$, $\mathrm{rank}_D(\varphi, s) = \min\{\alpha \in \mathrm{On} \mid (\varphi, s) \in U_\alpha\}$.

Lemma 2.16 If $\varphi \in \mathrm{cl}(\varphi_{\mathrm{I}}) \setminus D$, $\mathrm{rank}_D(\varphi, s) = 0$. Otherwise, the following holds:

- $\mathrm{rank}_D(\varphi_1 \vee \varphi_2, s) = \min(\{\mathrm{rank}_D(\varphi_j, s) \mid s \models \varphi_j,\ j = 1, 2\})$.

- $\mathrm{rank}_D(\varphi_1 \wedge \varphi_2, s) = \max(\mathrm{rank}_D(\varphi_1, s), \mathrm{rank}_D(\varphi_2, s))$.
- $\mathrm{rank}_D(\langle m \rangle \varphi, s) = \min\{\mathrm{rank}_D(\varphi, s') + 1 \mid (s, s') \in R(m),\ s' \models \varphi\}$.
- $\mathrm{rank}_D([m]\varphi, s) = \sup\{\mathrm{rank}_D(\varphi, s') + 1 \mid (s, s') \in R(m)\}$.
- $\mathrm{rank}_D(\varphi, s) = \mathrm{rank}_D(\exp(\varphi), s)$ if $\varphi = \mu X \varphi'$ or $\nu X \varphi'$.
- $\mathrm{rank}_D(@n\, \varphi, s) = \mathrm{rank}_D(\varphi, \tilde{\lambda}(n)) + 1$.

Proof Immediate from the definition. ∎

Definition 2.17 Let φ be an alternation-free formula in PNF and $\mathcal{K} = (S, R, \lambda)$ a Kripke structure. A *choice function* c is a function that satisfies the following conditions:

- $\mathrm{dom}(c) = \Phi \times S$, where Φ is the set of formulae in $\mathrm{cl}(\varphi)$ that is in the form of either $\varphi_1 \vee \varphi_2$ or $\langle m \rangle \varphi$ where $m \in \mathrm{Mod}$.
- For $\varphi_1 \vee \varphi_2 \in \mathrm{cl}(\varphi)$ and $s \in S$, $c(\varphi_1 \vee \varphi_2, s)$ is either φ_1 or φ_2. And if $s \models \varphi_1 \vee \varphi_2$ then $s \models c(\varphi_1 \vee \varphi_2, s)$.
- For $\langle m \rangle \varphi \in \mathrm{cl}(\varphi)$ and $s \in S$, $s' = c(\langle m \rangle \varphi, s) \in S$. And if $s \models \langle m \rangle \varphi$, then $(s, s') \in R(m)$, and $s' \models \varphi$. ∎

Lemma 2.18 Let φ_{I} be an alternation-free formula in PNF, $\mathcal{K} = (S, R, \lambda)$ a Kripke structure, $Z \subseteq \mathrm{cl}(\varphi_{\mathrm{I}}) \times S$, and c is a choice function for φ_{I} and $\mathcal{K}$. If the following conditions are satisfied, $\mathcal{K}, s \models \varphi$ holds for all $(\varphi, s) \in Z$.

(1) Every dead end in Z is 1-vertex. That is, for $p \in \mathrm{PS} \cup \mathrm{Nom}$ and $s \in S$,
   - $(p, s) \in Z \implies s \models p$.
   - $(\neg p, s) \in Z \implies s \not\models p$.

(2) Z is a 1-trap for Player 1 with trapping strategy $\sigma(c)$. That is, the following holds for $s \in S$.
   - $(\varphi_1 \vee \varphi_2, s) \in Z \implies (c(\varphi_1 \vee \varphi_2, s), s) \in Z$.
   - $(\varphi_1 \wedge \varphi_2, s) \in Z \implies (\varphi_1, s) \in Z$ and $(\varphi_2, s) \in Z$.
   - $(\langle m \rangle \varphi, s) \in Z \implies (\varphi, c(\langle m \rangle \varphi, s)) \in Z$.
   - $([m]\varphi, s) \in Z \implies (\varphi, s') \in Z$ for all $s' \in S$ such that $(s, s') \in R(m)$.

- $(\varphi, s) \in Z \implies (\exp(\varphi), s) \in Z$ for $\varphi = \mu X\varphi'$ or $\nu X\varphi'$.
- $(@n\,\varphi, s) \in Z \implies (\varphi, \tilde{\lambda}(n)) \in Z$.

(3) Any play conforming with the strategy $\sigma(c)$ does not remain in an SCC in $\mathscr{D}_\mu$ for infinitely long steps. More precisely, for any $D \in \mathscr{D}_\mu(\varphi_{\mathrm{I}})$, $\varphi \in D$, $s \in S$, and strategy σ_1 of Player 1, if $(\varphi, s) \in Z$ then there is $k \in \omega$ such that $(\sigma(c) * \sigma_1)(\varphi, s)(k) \notin D \times S$.

Proof Using the choice function, a memoryless strategy σ of Player 0 is defined in an obvious manner. Let $(\varphi, s) \in Z$. We will show that σ is a winning strategy on (φ, s). It is easy to see using condition (2) that for any trace τ beginning at (φ, s) conforming σ and for $n \in \mathrm{dom}(\tau)$, $\tau(n) = (\varphi_n, s_n) \in Z$. If $\mathrm{dom}(\tau)$ is finite, the winner of τ is Player 0 by condition (1). If $\mathrm{dom}(\tau)$ is infinite, there is $N \in \omega$ and $D \in \mathscr{D}$ such that $\varphi_n \in D$ for all $n \geq N$. By condition (3), $D \in \mathscr{D}_\nu$. This means that all priorities (greater than zero) appearing infinitely often are even and therefore the winner of τ in this case is also Player 0. ∎

2.4 Nominals

In this section, we prepare some lemmas that will be used in later chapters. Let L be a modal language such that $\mathrm{L}_0(@, \mu_{\mathrm{AF}}) \subseteq L$. Symbols λ and λ' stand for either μ or ν as before.

Definition 2.19 A formula $\xi \in L$ is *FA-free* if for any its subformula in the form of $\lambda X\psi$, the at operator (@) does not occur in ψ. It is *AV-free* if for any its subformula in the form of $@n\,\psi$, no free variable occurs in ψ. ∎

In other words, a formula is FA-free if no *f*ixed operator has the *a*t operator in its scope and AV-free if no *a*t operator has a free *v*ariable in its scope.

Lemma 2.20 If a closed formula is FA-free, it is AV-free.

Proof If a closed formula ξ has a subformula $@n\,\varphi$ and φ has a free variable X, the variable X must be bound, meaning there is some subformula $\lambda X\psi$ of ξ such that $@n\,\varphi$ is a subformula of ψ. ∎

Lemma 2.21 For any formula $\xi \in L$, there is a FA-free formula $\eta \in L$ that is equivalent to ξ.

In order to prove Lemma 2.21, we need the following lemma:

Lemma 2.22 Assume $\xi \in L$ is in PNF and $\psi_0 = @n\,\psi$ is a subformula of ξ. Further assume $\lambda X\varphi$ is the innermost subformula of this form that contains ψ_0 as a subformula; that is, $\lambda X\varphi$ is a subformula of ξ, ψ_0 is a subformula of φ, but there is no subformula of φ in the form of $\lambda' Y\eta$ such that η contains ψ_0 as a subformula. Let $\varphi^{\mathrm{T}} = \varphi[\mathbf{true}/\psi_0]$ and $\varphi^{\mathrm{F}} = \varphi[\mathbf{false}/\psi_0]$, that is, φ^{T} is the formula obtained from φ by replacing all occurrences of ψ_0 with **true** and φ^{F} is similar. Let $\psi^{\mathrm{T}} = \psi[\lambda X\varphi^{\mathrm{T}}/X]$ and $\psi^{\mathrm{F}} = \psi[\lambda X\varphi^{\mathrm{F}}/X]$. Then if $\lambda = \mu$,

$$\mu X\varphi \equiv @n\,\psi^{\mathrm{F}} \rightarrow \mu X\varphi^{\mathrm{T}}\,;\mu X\varphi^{\mathrm{F}}$$

holds. Similarly if $\lambda = \nu$,

$$\nu X\varphi \equiv @n\,\psi^{\mathrm{T}} \rightarrow \nu X\varphi^{\mathrm{T}}\,;\nu X\varphi^{\mathrm{F}}$$

holds.

Example 2.23 Before proving the lemma, let us show an example: we construct a FA-free formula equivalent to

$$\xi = \mu X(p \vee \langle m\rangle X \vee @n\,(r \vee (q \wedge X))).$$

We start by setting $\varphi = p \vee \langle m\rangle X \vee @n\,\psi$ and $\psi = r \vee (q \wedge X)$. Then $\varphi^{\mathrm{T}} = \mathbf{true}$, $\varphi^{\mathrm{F}} = p \vee \langle m\rangle X$, $\psi^{\mathrm{T}} = r \vee q$, and $\psi^{\mathrm{F}} = r \vee (q \wedge \mu X(p \vee \langle m\rangle X))$. Thus according to the lemma,

$$\begin{aligned}\xi &\equiv @n\,(r \vee (q \wedge \mu X(p \vee \langle m\rangle X))) \;\rightarrow\; \mathbf{true}\;;\; \mu X(p \vee \langle m\rangle X)\\ &\equiv @n\,(r \vee (q \wedge \mu X(p \vee \langle m\rangle X))) \vee \mu X(p \vee \langle m\rangle X).\end{aligned}$$

∎

Proof of Lemma 2.22 We only show the first half. The second half is similar.

Let $\mathcal{K} = (S, R, \lambda)$ be a Kripke structure and v a valuation for $\mathcal{K}$ and $s \in S$. We show:

(a) When $\mathcal{K}, v \models @n\,\psi^{\mathrm{F}}$, $\quad \mathcal{K}, v, s \models \mu X\varphi^{\mathrm{T}} \iff \mathcal{K}, v, s \models \mu X\varphi$.

(b) When $\mathcal{K}, v \not\models @n\,\psi^{\mathrm{F}}$, $\quad \mathcal{K}, v, s \models \mu X\varphi \iff \mathcal{K}, v, s \models \mu X\varphi^{\mathrm{F}}$.

Note that whether $@n\,\psi^{\mathrm{F}}$ is satisfied or not is independent from $s \in S$. In both cases the direction from right to left is clear since φ is in PNF, so we show the other direction.

We define $S(\xi, X, \alpha) \subseteq S$ for formula ξ, propositional variable X, and ordinal number α as follows. We omit X and write $S(\xi, \alpha)$ if it is clear from the context.

- $S(\xi, 0) = \varnothing$
- $S(\xi, \alpha+1) = [\![\varphi]\!]^{\mathcal{K}, v[X \mapsto S(\xi,\alpha)]}$
- $S(\xi, \alpha) = \bigcup_{\beta<\alpha} S(\xi, \beta)$ (if α is limit)

Let κ, κ^{T}, and κ^{F} be the ordinal number at which $S(\varphi, \cdot)$, $S(\varphi^{\mathrm{T}}, \cdot)$, and $S(\varphi^{\mathrm{F}}, \cdot)$, respectively, converges. Namely, $S(\varphi, \kappa) = [\![\mu X \varphi]\!]$ holds and similarly for the others.

(a) Assume $\mathcal{K}, v \models @n\, \psi^{\mathrm{F}}$. We will show on induction α that

$$S(\varphi^{\mathrm{T}}, \alpha) \subseteq S(\varphi, \kappa^{\mathrm{F}} + \alpha). \tag{2.1}$$

Once it is established, by setting $\alpha = \kappa^{\mathrm{T}}$, we have $[\![\mu X \varphi^{\mathrm{T}}]\!] = S(\varphi^{\mathrm{T}}, \kappa^{\mathrm{T}}) \subseteq S(\varphi, \kappa^{\mathrm{F}} + \kappa^{\mathrm{T}}) \subseteq [\![\mu X \varphi]\!]$, which is to be proved in (a).

When $\alpha = 0$ or α is limit, (2.1) is clearly satisfied. For $\alpha + 1$, by induction hypothesis we have $S(\varphi^{\mathrm{T}}, \alpha+1) = [\![\varphi^{\mathrm{T}}]\!]^{v[X \mapsto S(\varphi^{\mathrm{T}}, \alpha)]} \subseteq [\![\varphi^{\mathrm{T}}]\!]^{v[X \mapsto S(\varphi, \kappa^{\mathrm{F}}+\alpha)]}$. On the other hand, by definition, $S(\varphi, \kappa^{\mathrm{F}}+\alpha+1) = [\![\varphi]\!]^{v[X \mapsto S(\varphi, \kappa^{\mathrm{F}}+\alpha)]}$. Therefore it is enough to show

$$\mathcal{K}, v[X \mapsto S(\varphi, \kappa^{\mathrm{F}} + \alpha)], s \models \varphi \leftrightarrow \varphi^{\mathrm{T}}. \tag{2.2}$$

From the assumption

$$\mathcal{K}, v \models @n\, \psi[\mu X \varphi^{\mathrm{F}} / X],$$

and therefore

$$\mathcal{K}, v[X \mapsto S(\varphi^{\mathrm{F}}, \kappa^{\mathrm{F}})] \models @n\, \psi.$$

Since $S(\varphi^{\mathrm{F}}, \kappa^{\mathrm{F}}) \subseteq S(\varphi, \kappa^{\mathrm{F}}) \subseteq S(\varphi, \kappa^{\mathrm{F}} + \alpha)$,

$$\mathcal{K}, v[X \mapsto S(\varphi, \kappa^{\mathrm{F}} + \alpha)] \models @n\, \psi,$$

which means

$$\mathcal{K}, v[X \mapsto S(\varphi, \kappa^{\mathrm{F}} + \alpha)] \models @n\, \psi \leftrightarrow \mathbf{true}.$$

Now (2.2) follows from the definition of φ^{T}.

(b) Assume $\mathcal{K}, v \not\models @n\, \psi^{\mathrm{F}}$. With similar argument as in part (a), it implies $\mathcal{K}, v[X \mapsto S(\varphi^{\mathrm{F}}, \kappa^{\mathrm{F}})], s \models \varphi \leftrightarrow \varphi^{\mathrm{F}}$.

We show by induction on α, $S(\varphi, \alpha) \subseteq S(\varphi^{\mathrm{F}}, \alpha)$. Cases for $\alpha = 0$ and limit ordinal α are trivial. For a successor ordinal, we have $S(\varphi, \alpha+1) = [\![\varphi]\!]^{v[X \mapsto S(\varphi, \alpha)]} \subseteq [\![\varphi]\!]^{v[X \mapsto S(\varphi^{\mathrm{F}}, \alpha)]} = [\![\varphi^{\mathrm{F}}]\!]^{v[X \mapsto S(\varphi^{\mathrm{F}}, \alpha)]} = S(\varphi^{\mathrm{F}}, \alpha+1)$.

Then, $[\![\mu X \varphi]\!] = S(\varphi, \kappa) \subseteq S(\varphi^{\mathrm{F}}, \kappa) \subseteq [\![\mu X \varphi^{\mathrm{F}}]\!]$ and we are done. ∎

Lemma 2.24 Assume $\xi = \lambda X \varphi \in L$ and φ is FA-free. Then there is a FA-free formula that is equivalent to ξ

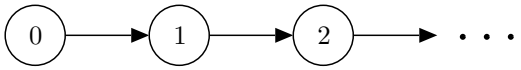


Figure 2.1: Kripke Structure $\mathcal{K}_1$

Proof If ξ itself is not FA-free, pick a subformula $\psi_0 = @n\ \psi$ of φ such that ψ is @-free. Then Lemma 2.22 can be applied and let ξ' be the resulting formula. Note that every subformula of ξ' in the form of $\lambda' Y \psi$, where $Y' \neq X$, is identical to a formula that exists in ξ. Also a subformula of ξ' whose principal operator is λX is either $\lambda X \varphi^{\mathrm{T}}$ or $\lambda X \varphi^{\mathrm{F}}$ and in both φ^{T} and φ^{F} the number of occurrences of @ is strictly less than that in $\lambda X \varphi$. Therefore if we repeat this step for those subformulae whose principal operator is λX, the procedure completes after finitely many steps and we get a FA-free formula equivalent to ξ. ∎

Proof of Lemma 2.21 We prove the lemma by induction on the construction of the formula. In the case of μ and ν we can use Lemma 2.24 and other cases are trivial. ∎

2.5 Reverse Modalities

The main reason we introduce the reverse modalities is that they are useful to express the weakest preconditions of pointer manipulating programs as we will see in Chapter 4. We also see that introducing them strengthens the expression power: the following example shows that the language $\mathrm{L}_0(\mu_{\mathrm{AF}}, ^{-})$ does not have the finite model property, while it is known that the language $\mathrm{L}_0(\mu_{\mathrm{AF}})$ does have the property.

Example 2.25 Let $\psi = \mu X([\overline{m}]X)$ and $\varphi = \nu Y(\psi \wedge \langle m \rangle Y)$. Note that for Kripke structure $\mathcal{K} = (S, R, \lambda)$ and $s \in S$, $s \models \psi$ holds if and only if there is no infinite sequence $(s_i \mid i < \omega)$ of states such that $s_0 = s$, $(s_{i+1}, s_i) \in R(m)$ for $i < \omega$. Also $s \models \varphi$ holds if and only if there is an infinite sequence $(t_j \mid j < \omega)$ of states such that $s = t_0$, $(t_j, t_{j+1}) \in R(m)$ and $t_j \models \psi$ for $j < \omega$.

Let $\mathcal{K}_1$ be a Kripke structure with $\mathcal{K}_1 = (\omega, R, \lambda)$ and $R(m) = \{(i, i+1) \mid i < \omega\}$ (see Figure 2.1). It is easy to see that $\mathcal{K}_1, 0 \models \varphi$ from the notes above, thus φ is satisfiable. However φ cannot be satisfied by a finite Kripke structure. To show this suppose there is a finite Kripke structure $\mathcal{K} = (S, R, \lambda)$ and $s \in S$ such that $\mathcal{K}, s \models \varphi$. Let $(t_j \mid j < \omega)$ be the sequence of states with the property stated above. Since S is finite there are natural numbers $j < j'$ such that $t_j = t_{j'}$. In other words, there is a circle in the sequence. Then by taking a sequence of states starting with t_j,

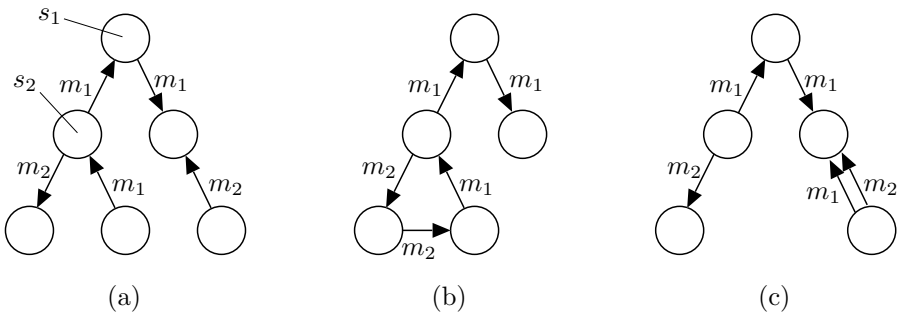


Figure 2.2: Shape of a Tree

and following the circle in reverse order, we can conclude $t_j \not\models \psi$, contradicting the property for the sequence $(t_j \mid j < \omega)$. ∎

2.6 Functional Kripke Structures

Definition 2.26 A Kripke structure $\mathcal{K} = (S, R, \lambda)$ is *functional* if for all $m \in \mathrm{MS}$, $R(m)$ is a function, that is, for any $s \in S$ there is at most one $s' \in S$ such that $(s, s') \in R(m)$. ∎

Note that $R(m)$ is required to be a function only for *forward* modalities m. Relation $R(\overline{m})$ may or may not be a function for $m \in \mathrm{MS}$.

When $\mathcal{K} = (K, R, \lambda)$ is a functional Kripke structure and $(s, s') \in R(m)$ for $m \in \mathrm{MS}$, we denote s' by $R(m)(s)$.

Formula φ is said to be *functionally satisfiable* if there is a function Kripke structure $\mathcal{K}$ and $s \in |\mathcal{K}|$ such that $\mathcal{K}, s \models \varphi$. Every functionally satisfiable formula is satisfiable, but the converse is not true. For example formula $\langle m \rangle p \wedge \langle m \rangle \neg p$ is satisfiable and functionally unsatisfiable.

The *functional satisfiability problem* for a language L is the problem to decide for a given formula φ in L whether φ is functionally satisfiable. We write $\mathrm{Sat}(\mu, \mathrm{func})$ for the functional satisfiability problem for $\mathrm{L}_0(\mu)$. Similar notations are used for other combinations as well.

2.7 The Tree Model Property

The satisfiability problems for various logics are decidable and many cases depend on the fact that the logics satisfy the tree model property.

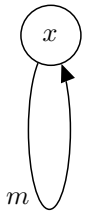


Figure 2.3: Kripke Structure That Satisfies $@x\ \langle m\rangle x$

Definition 2.27 A Kripke structure $\mathcal{K} = (S, R, \lambda)$ is *in the shape of a tree* if there is a tree relation $\overline{R}$ on S such that

- if $(s, s') \in \overline{R}$ then there exists unique $m \in \mathrm{Mod}$ such that $(s, s') \in R(m)$, and
- if $(s, s') \in R(m)$ for some $m \in \mathrm{Mod}$ then either $(s, s') \in \overline{R}$ or $(s', s) \in \overline{R}$.

A language has the *tree model property* if any satisfiable formula of the language is satisfied by a Kripke structure in the shape of a tree. ∎

Assuming the language has the reverse modalities, Kripke structure (a) shown in Figure 2.2 is in the shape of a tree, while (b) and (c) are not. Note that $(s_1, s_2) \in R(\overline{m_1})$ and $\overline{m_1} \in \mathrm{Mod}$ in (a).

Not only the minimal modal logic **K** itself but also its various extensions, such as CTL and the modal μ calculus, have the tree model property. As Vardi mentioned in [73], the property is a key for the satisfiability problem of a logic to be decidable. On the other hand, it naturally restricts the application of the logic in question to such domains in which data can be represented in trees. Although there are plenty of such domains, our target applications do not necessarily fall under the category.

If a logic has nominals it does not have the tree model property. Consider formula $\varphi = @x\ \langle m\rangle x$. It is clearly satisfied by the Kripke structure shown in Figure 2.3 thus it is satisfiable. However, it cannot be satisfied by a Kripke structure in the shape of a tree because there should be a self loop at the node $\tilde{\lambda}(x)$ like the one shown in the figure.

Introducing the reverse modalities or restriction to the functional Kripke structure usually preserves the tree model property. The decision procedures we will discuss in Chapter 3 constructs a Kripke structure (functional Kripke structure, respectively) in the shape of a tree for a formula in $\mathrm{L}_0(\mu_{\mathrm{AF}}, {}^{-})$ if it is satisfiable (functionally satisfiable, respectively).

Chapter 3

Decision Procedures

In Chapter 2, we introduce four extensions to the minimal modal logic **K**. Although they can be introduced independently, the satisfiability problem for the logic with all the four extensions are undecidable. On the other hand, all the satisfiability problems for the logics with three extensions out of the four are decidable.

The logic without fixed-point operators is not suitable for our application since the expressive power is not strong enough. Therefore we concentrate on the other three combinations: i.e., $\mathrm{Sat}(\mu_{\mathrm{AF}}, @, {}^{-})$, $\mathrm{Sat}(\mu_{\mathrm{AF}}, {}^{-}, \mathrm{func})$, and $\mathrm{Sat}(\mu_{\mathrm{AF}}, @, \mathrm{func})$. In this chapter we provide decision procedures for these problems. They are more efficient than known decision procedures for general fixed-point operators (i.e., not restricted to the alternation-free part) and implementations are available.

The procedure for the ordinary satisfiability (we call it the non-FUNC procedure) and that for the functional satisfiability (the FUNC procedure) differ in some points. In such points we will describe the non-FUNC procedure first and then describe the difference for the FUNC procedure.

Decision procedures described in this chapter are extensions of our previous results [66, 67].

3.1 Tableau

Let φ_{I} be given formula of which we will judge the satisfiability. For this purpose we can ignore any predicate symbols, modality symbols, or nominals that do not occur in φ_{I}. Therefore we assume that sets PS, MS, and Nom are all finite.

For $D \in \mathscr{D}_{\mu}$, we denote by BMod_D the set of modality symbol m such that both m and $\overline{m}$ appear as the modality for the principal operator of some elements of D. The subset of D that consists of the formulae whose principal operator is a diamond

$\langle m \rangle$ or a box $[m]$ with $m \in \mathrm{BMod}_D$ or $\overline{m} \in \mathrm{BMod}_D$ is denoted by BForm_D.

Our decision procedures are variants of the tableau method. A node of our tableau is a pair (x, y) of functions that satisfies the following conditions:

- x is a function and its domain is Lean. For $\varphi \in \mathrm{Lean}$, $x(\varphi)$ is either a natural number or value "∞". If $\varphi \notin D_\mu$ or $\mathrm{BForm}_{D(\varphi)} = \varnothing$, $x(\varphi)$ is either 0 or ∞; otherwise either $x(\varphi) < |\mathrm{BForm}_{D(\varphi)}|$ or $x(\varphi) = \infty$. We regard $0 < 1 < \cdots < \infty$.
- y is a function and its domain is $\{\varphi \in D_\mu \cap \mathrm{Lean} \mid x(\varphi) < \infty\}$. For $\varphi \in \mathrm{dom}(y)$, $y(\varphi)$ is a natural number and $y(\varphi) \leq |\mathrm{Nom}| \cdot |D(\varphi)|$.
- If $\varphi_1, \varphi_2 \in D \in \mathscr{D}_\mu$ and $x(\varphi_1) < x(\varphi_2) < \infty$, then $y(\varphi_1) \leq y(\varphi_2)$.

Note that there are only finitely many pairs (x, y) that satisfy the conditions. We denote by Tab the set of all pairs that satisfy the above conditions. When $t = (x, y) \in \mathrm{Tab}$, x is denoted by x_t and y is denoted by y_t.

The intention is that at node $t = (x, y)$, $\varphi \in \mathrm{Lean}$ is satisfied if $x(\varphi) < \infty$. In the case $\varphi \in D_\mu$, there should be "evidences" that φ is satisfied at t. For example if $\varphi = \mu X(p \vee \langle m \rangle X)$ and $t \models \varphi$ but $t \not\models p$, and there is t' such that $(t, t') \in R(m)$ and $t' \models p$, then t' is the evidence for t to satisfy φ.

The values $x(\varphi)$ and $y(\varphi)$ both express sorts of the distance between t and such evidences. The value $x(\varphi)$ is used to compare the distances of formulae in the same node, while the value $y(\varphi)$ is used to compare the distance of a formula with that of another formula in a different node. For more precise meaning, please refer to Section 3.3.

A function g from Nom to Tab is called a *naming function* when it satisfies the following conditions:

- $x_{g(n)}(n) = 0$ for all $n \in \mathrm{Nom}$.
- $x_{g(n_1)}(n_2) = 0 \implies g(n_1) = g(n_2)$ for any $n_1, n_2 \in \mathrm{Nom}$.

We denote by NF the set of naming functions. Note that NF is a finite set. A naming function can be regarded that it specifies in which node each nominal should be satisfied. In that sense, the first condition requires that a nominal should be satisfied at the node the nominal is satisfied. The second condition says that if n_2 is satisfied at the node where n_1 is satisfied, then the nodes that the two nominal are satisfied should be identical.

Values $x(\varphi)$ are defined only on Lean but we can extend its domain to $\mathrm{cl}(\varphi_{\mathrm{I}})$ by induction based on the lean in the following way. Assume $t = (x, y)$ and $D = D(\varphi)$.

- $x(\neg p) = \begin{cases} 0 & \text{if } x(p) = \infty \\ \infty & \text{if } x(p) = 0 \end{cases}$ for $p \in \mathrm{PS} \cup \mathrm{Nom}$
- If $\varphi = \varphi_1 \vee \varphi_2$, $\quad x(\varphi) = \min(\tilde{x}(\varphi_1, D), \tilde{x}(\varphi_2, D))$
- If $\varphi = \varphi_1 \wedge \varphi_2$, $\quad x(\varphi) = \max(\tilde{x}(\varphi_1, D), \tilde{x}(\varphi_2, D))$
- If $\varphi = \mu X\psi$ or $\nu X\psi$, $\quad x(\varphi) = x(\exp(\varphi))$

where $\tilde{x}$ is defined as follows:

$$\tilde{x}(\varphi, D) = \begin{cases} 0 & \text{if } x(\varphi) < \infty \text{ and } \varphi \notin D \\ x(\varphi) & \text{otherwise} \end{cases}$$

Then the domain of y is extended to $\{\varphi \in D_\mu \mid x(\varphi) < \infty\}$ in the same way as in x:

- If $\varphi = \varphi_1 \vee \varphi_2$, $\quad y(\varphi) = \min(\tilde{y}(\varphi_1, D), \tilde{y}(\varphi_2, D))$
- If $\varphi = \varphi_1 \wedge \varphi_2$, $\quad y(\varphi) = \max(\tilde{y}(\varphi_1, D), \tilde{y}(\varphi_2, D))$
- If $\varphi = \mu X\psi$ or $\nu X\psi$, $\quad y(\varphi) = y(\exp(\varphi))$

where $\tilde{y}$ is:

$$\tilde{y}(\varphi, D) = \begin{cases} 0 & \text{if } x(\varphi) < \infty \text{ and } \varphi \notin D \\ y(\varphi) & \text{otherwise} \end{cases}$$

Note that $x(\varphi)$ is used in the condition of the definition of $\tilde{y}$.
We write $t \Vdash \varphi$ when $x_t(\varphi) < \infty$. A node $t \in \mathrm{Tab}$ *has a name* if there is $n \in \mathrm{Nom}$ such that $t \Vdash n$.

Values of x other than 0 and ∞ are used to define a relation loop-freeness. Let $m \in \mathrm{Mod}$, $t_1, t_2 \in \mathrm{Tab}$. A formula $\varphi \in D_\mu$ in the form of $\langle m \rangle \varphi'$ or $[m]\varphi'$ that satisfies $t_1 \models \varphi$ is *loop-free* for the transition from t_1 to t_2 if there does *not* exist formula $\psi \in D(\varphi)$ in the form of $[\overline{m}]\psi'$ such that both of the following holds.

- $x_{t_1}(\varphi) \leq x_{t_1}(\psi') < \infty$.
- $x_{t_2}(\psi) \leq x_{t_2}(\varphi') < \infty$.

In the case of the FUNC procedure, replace "in the form of $[m]\psi'$" with "in the form of $\langle m \rangle \psi'$ or $[m]\psi'$. We write $\mathrm{LoopFree}(t_1, t_2, \varphi)$ if this condition is satisfied.

We next define the transition relation $\mathrm{Tr}(m)$ on Tab for $m \in \mathrm{MS}$. Assume $t = (x, y)$ and $t' = (x, y')$ are elements of Tab. Then $(t, t') \in \mathrm{Tr}(m)$ if the following conditions are satisfied:

(1) For any $\varphi \in$ Lean in the form of $[m]\varphi'$, if $t \Vdash \varphi$ then $t' \Vdash \varphi'$. Moreover if $\varphi \in D_\mu$, $y(\varphi) \geq y'(\varphi')$. Moreover if t' has a name, then $y(\varphi) > y'(\varphi')$.

(2) For any $\varphi \in$ Lean in the form of $[\overline{m}]\varphi'$, if $t' \Vdash \varphi$ then $t \Vdash \varphi'$. Moreover if $\varphi \in D_\mu$, $y'(\varphi) \geq y(\varphi')$. Moreover if t has a name, then $y'(\varphi) > y(\varphi')$.

(3) For any $\varphi \in$ Lean in the form of $[m]\varphi'$, LoopFree(t, t', φ) holds. Note that this implies LoopFree(t', t, ψ) for any $\psi \in$ Lean in the form of $[\overline{m}]\psi'$.

In the case of the FUNC procedure, replace "in the form of $[m]\varphi'$" in conditions (1) and (3) with "in the form of $\langle m\rangle\varphi'$ or $[m]\varphi'$".

For $m \in$ MS, $\mathrm{Tr}(\overline{m})$ is defined as $\mathrm{Tr}(\overline{m}) = \mathrm{Tr}(m)^{-1}$. For formula $\varphi \in \mathrm{cl}(\varphi_{\mathrm{I}})$ in the form of $\overline{m}\varphi$, where $m \in$ Mod, we also define relation $\mathrm{Tr}(\langle m\rangle\varphi)$ on Tab by $(t, t') \in \mathrm{Tr}(\langle m\rangle\varphi)$ if and only if:

- $(t, t') \in \mathrm{Tr}(m)$, $t \Vdash \langle m\rangle\varphi$, and $t' \Vdash \varphi$.
- If $\langle m\rangle\varphi \in D_\mu$, then LoopFree$(t, t', \langle m\rangle\varphi)$ and $y_t(\langle m\rangle\varphi) \geq y_{t'}(\varphi)$. Moreover if t' has a name, $y_t(\langle m\rangle\varphi) > y_{t'}(\varphi)$.

Note that in the case of the FUNC procedure, if $(t, t') \in \mathrm{Tr}(m)$ and $t \Vdash \langle m\rangle\varphi$, $(t, t') \in \mathrm{Tr}(\langle m\rangle\varphi)$ is satisfied.

3.2 Procedure

We fix a naming function g and describe a sub-procedure for g. The whole procedure is to execute the sub-procedure for all $g \in$ NF. If there is a g for which the procedure succeeds, then we judge that φ_{I} is satisfiable. If the procedure fails for all $g \in$ NF, we judge that φ_{I} is not satisfiable. In the rest of this subsection we describe the sub-procedure.

A sequence $(T_k)_{k \leq K}$ of subsets of Tab is constructed so that $T_0 \supseteq T_1 \supseteq \cdots$. We repeat the construction until $T_k = T_{k+1}$ holds and set $K = k$. Since Tab is a finite set there must be such $k \in \omega$. The procedure succeeds if

- there is $t \in T_K$ such that $t \Vdash \varphi_{\mathrm{I}}$, and
- for all $n \in$ Nom, $g(n) \in T_K$.

The initial set T_0 consists of the elements t of Tab that satisfy the following conditions:

- For all $n \in$ Nom, $t \Vdash n \implies t = g(n)$.

- For all $@n\,\varphi \in \mathrm{cl}(\varphi_{\mathrm{I}})$, where $n \in \mathrm{Nom}$, if $t \Vdash @n\,\varphi$ then $g(n) \Vdash \varphi$. Moreover if $@n\,\varphi \in D_\mu$, $y_t(@n\,\varphi) > y_{g(n)}(\varphi)$.

T_{k+1} is obtained from T_k by removing $\Diamond$-inconsistent nodes and μ-inconsistent nodes in T_k. We give their definitions below. Let T be a subset of Tab. $t \in T$ is *$\Diamond$-consistent* in T if for all formulae in Lean in the form of $\langle m\rangle\varphi$, where $m \in \mathrm{Mod}$, if $t \Vdash \varphi$, there exists $t' \in T$ such that $(t, t') \in \mathrm{Tr}(\langle m\rangle\varphi)$. We say $t \in T$ is *$\Diamond$-inconsistent* if it is not $\Diamond$-consistent.

In order to define μ-consistency, we will construct a sequence $(V_j)_{j\leq J}$ of subsets of $T \times \mathscr{P}(D_\mu \cap \mathrm{Lean})$ so that $V_0 \subseteq V_1 \subseteq \cdots$. We repeat it until $V_j = V_{j+1}$ holds and set $J = j$. Since $T \times \mathscr{P}(D_\mu \cap \mathrm{Lean})$ is a finite set, there must be such $j \in \omega$. A node $t \in T$ is *μ-consistent* in T if for all $D \in \mathscr{D}_\mu$, $(t, \{\varphi \in D \cap \mathrm{Lean} \mid t \Vdash \varphi\}) \in V_J$, and t is *μ-inconsistent* if it is not μ-consistent.

Let $t = (x, y) \in \mathrm{Tab}$ and $E \subseteq D_\mu$. We define $x^E(\varphi)$, for $\varphi \in \mathrm{Lean}$ first by:

$$x^E(\varphi) = \begin{cases} \infty & \text{if } \varphi \in D_\mu \setminus E \\ x(\varphi) & \text{otherwise} \end{cases}$$

then we extend the domain of x^E to $\mathrm{cl}(\varphi_{\mathrm{I}})$ in the same way as we extended the domain of x. We denote by $t^E \Vdash \varphi$ if $x^E(\varphi) < \infty$.

The initial set V_0 is defined as $V_0 = \{(t, \varnothing) \mid t \in T\}$. The general case is: $(t, E) \in V_{j+1}$ if and only if $(t, E) \in V_j$ or the following conditions are satisfied:

(a) E is a set of formulae that "holds" at t and it is closed under downward order of x and y within a SCC, that is, the following holds for all $\varphi \in E$.

- $t \Vdash \varphi$.
- If $\psi \in D(\varphi) \cap \mathrm{Lean} \cap \mathrm{dom}(y_t)$ and $y_t(\varphi) > y_t(\psi)$, then $\psi \in E$.
- If $\psi \in D(\varphi) \cap \mathrm{Lean}$ and $x_t(\varphi) > x_t(\psi)$, then $\psi \in E$.

(b) For all $m \in \mathrm{Mod}$ and formulae in Lean in the form of $\langle m\rangle\varphi$, if $t \Vdash \langle m\rangle\varphi$, then there is $(t', E') \in V_j$ such that:

- $(t, t') \in \mathrm{Tr}(\langle m\rangle\varphi)$.
- For all formulae in E in the form of $[m]\psi$, $\quad t'^{E'} \Vdash \psi$.
- If $\langle m\rangle\varphi \in E$, $\quad t'^{E'} \Vdash \varphi$.

In the case of the FUNC procedure, replace "For all $m \in \mathrm{Mod}$" at the first line of condition (b) with "For all $m \in \mathrm{Mod} \setminus \mathrm{MS}$". That is, condition (b) is applied only

to the backward modalities. And the following condition for the forward modalities should be added:

(c) For all $m \in \mathrm{MS}$, if there is a formula in the form of $\langle m \rangle \varphi$ such that $t \Vdash \varphi$, then there is $(t', E') \in V_j$ such that:

- $(t, t') \in \mathrm{Tr}(m)$.
- For all formulae in E in the form of $\langle m \rangle \psi$ and $[m]\psi$, $\quad t'^{E'} \Vdash \psi'$.

That completes the description of the decision procedure.

3.3 Soundness

In this subsection we prove that the decision procedure described in Section 3.2 is sound[1], namely the following theorem holds.

Theorem 3.1 If there is a Kripke structure $\mathcal{K} = (S, R, \lambda)$ and $s_{\mathrm{I}} \in S$ such that $\mathcal{K}, s_{\mathrm{I}} \models \varphi_{\mathrm{I}}$, the non-FUNC procedure described in Section 3.2 succeeds. When $\mathcal{K}$ is functional, the FUNC procedure also succeeds.

This theorem holds even when both nominals and reverse modalities exist, while the completeness proof for this combination fails as we see in Section 3.4.

We start a proof of the theorem by defining a function $h : S \to \mathrm{Tab}$. Take $s \in S$ and let us write $h(s) = (x, y)$. Then x and y are defined as

$$x(\varphi) = \begin{cases} \infty & \text{if } s \not\models \varphi \\ 0 & \text{if } s \models \varphi \text{ and } \varphi \in \mathrm{Lean} \setminus D_\mu \\ |X(\varphi, s)| + 1 & \text{if } s \models \varphi \text{ and } \varphi \in D_\mu \end{cases}$$

where $X(\varphi, s) = \{\mathrm{rank}_D(\psi, s) \mid \psi \in \mathrm{BForm}_D,\ \mathrm{rank}_D(\psi, s) \leq \mathrm{rank}_D(\varphi, s)\}$ with $D = D(\varphi)$, and

$$y(\varphi) = |\{(n, \psi) \in \mathrm{Nom} \times D(\varphi) \mid \mathrm{rank}_{D(\varphi)}(\psi, \tilde{\lambda}(n)) < \mathrm{rank}_{D(\varphi)}(\varphi, s)\}|.$$

Note that the set appeared in the third case for the definition of x is finite since BForm_D is.

[1] It might be arguable if we should use the word "sound" for this direction. Since we discuss a procedure for *satisfiability*, it might be more appropriate to use the word "complete" for the direction. However, the terms "sound" and "complete" are frequently used for procedures for *validity*, which satisfiability is the dual of, and this direction is referred as "sound" in that context. Therefore we believe that our choice of the terminology causes less confusion.

Lemma 3.2 For $\varphi \in \mathrm{cl}(\varphi_{\mathrm{I}})$ and $s \in S$, $\quad \mathcal{K}, s \models \varphi \iff h(s) \Vdash \varphi$.

Proof It is clear from the definition of h that the lemma holds for $\varphi \in$ Lean. Other cases are also easily shown using the definition of the extension of the domain of x. ∎

Lemma 3.3 Assume $s \in S$, $t = (x, y) = h(s)$, $D \in \mathscr{D}_\mu$, $\varphi, \psi \in D$, $s \models \varphi$, and $s \models \psi$. Then, $x(\varphi) \leq x(\psi) \iff \mathrm{rank}_D(\varphi, s) \leq \mathrm{rank}_D(\psi, s)$.

Proof Clear from the definition of h. ∎

Lemma 3.4 $(s, s') \in R(m) \implies (h(s), h(s')) \in \mathrm{Tr}(m)$ holds for $s, s' \in S$ and $m \in$ Mod.

Proof It is enough to show the lemma for $m \in$ MS. Let $t = (x, y) = h(s)$ and $t' = (x', y') = h(s')$.

The first part of the condition (1) follows from Lemma 3.2. The second part from $\mathrm{rank}_D(\varphi, s) > \mathrm{rank}_D(\varphi', s')$ where $D = D(\varphi)$. The last part is clear from the definition of y. The condition (2) can be checked in a similar way.

The condition (3) follows from Lemma 3.3.

The case for FUNC procedure can be checked similarly, because $(s, s') \in R(m)$ and $s \models \langle m \rangle \varphi'$ implies $s' \models \varphi'$ when $\mathcal{K}$ is functional. ∎

Corollary 3.5 If $h[S] \subseteq T \subseteq$ Tab, A node $t \in h[S]$ is $\Diamond$-consistent in T.

Proof Combine Lemmas 3.2 and 3.4. ∎

Let On be the class of all ordinal numbers. For $D \in \mathscr{D}_\mu$, $\alpha \in$ On, and $s \in S$, let us denote by $F_D(\alpha, s)$ the set $\{\varphi \in D \cap \text{Lean} \mid s \models \varphi, \mathrm{rank}_D(\varphi, s) < \alpha\}$.

Lemma 3.6 For all $\alpha \in$ On and $D \in \mathscr{D}_\mu$ there exists $j \in \omega$ such that for all $s \in S$, $(h(s), F_D(\alpha, s)) \in V_j$.

Proof We prove the lemma by induction on α. The case $\alpha = 0$ is trivial since $F_D(0, s) = \varnothing$. The case in which α is limit is also clear since there exists $J \in \omega$ such that $V_j = V_J$ for all $j \geq J$.

For the remaining case, we assume that $(h(s), F_D(\alpha, s)) \in V_j$ for all $s \in S$ and prove that $(h(s), F_D(\alpha + 1, s)) \in V_{j+1}$ for all $s \in S$. Take $s \in S$ and let $t = h(s)$ and $E = F_D(\alpha + 1, s)$.

Take $m \in$ Mod and $\langle m \rangle \varphi \in$ Lean such that $t \models \langle m \rangle \varphi$. We need to find an appropriate $(t', E') \in V_j$. From Lemma 3.2, $s \models \langle m \rangle \varphi$ so we can pick $s' \in S$ such

that $s' \models \varphi$. If $\langle m \rangle \varphi \in D$, we take s' so that $\mathrm{rank}_D(\varphi, s') < \mathrm{rank}_D(\langle m \rangle \varphi, s)$ is also satisfied. Let $t' = h(s')$ and $E' = F_D(\alpha, s')$. We have $(t', E') \in V_j$ by the induction hypothesis.

The first condition $(t, t') \in \mathrm{Tr}(m)$ and $t' \Vdash \varphi$ follows from Lemmas 3.4 and 3.2. For the second condition take $[m]\psi \in E$. We need to show $t'^{E'} \Vdash \psi$. Let us write $t'^{E'} = (\hat{x}, y')$. Since $\mathrm{rank}_D([m]\psi, s) < \alpha + 1$, $\mathrm{rank}_D(\psi, s') < \alpha$. That means $\psi \in \alpha$, therefore $\hat{x}(\psi) = x'(\psi) < \infty$, thus $t'^{E'} \Vdash \psi$. The third condition is clear from $\mathrm{rank}_D(\langle m \rangle \varphi, s) > \mathrm{rank}_D(\varphi, s')$. When t' has a name, it means that $s' = \tilde{\lambda}(n)$ for some $n \in \mathrm{Nom}$, therefore the corresponding set in the definition of $y(\varphi)$ is a proper subset of that of $y(\langle m \rangle \varphi)$.

For "moreover" part, assume $\langle m \rangle \varphi \in E$. In this case we can take $s' \in S$ so that it further satisfies $\mathrm{rank}_D(\varphi, s') < \alpha$. and let $t' = h(s')$ and $E' = F_D(\alpha, s')$ as before. Then the two conditions can be proved in the same way as in the proof of Lemma 3.4.

In the case of the FUNC procedure, since $t \Vdash \langle m \rangle \varphi$, we have $s \models \langle m \rangle \varphi$, and there is unique $s' \in S$ such that $(s, s') \in R(m)$. Take this s' and the rest of the proof is similar to the above. ∎

Corollary 3.7 If $h[S] \subseteq T \subseteq \mathrm{Tab}$, A node $t \in h[S]$ is μ-consistent in T.

Proof Note that $F_D(\alpha, s) = \{\varphi \in D \cap \mathrm{Lean} \mid s \models \varphi\}$ if α is sufficiently large, for example if α is a larger cardinal than the cardinality of S. Therefore the corollary immediately follows from Lemmas 3.2 and 3.6. ∎

Proof of Theorem 3.1 Let g be a naming function that satisfies $g(n) = h(\tilde{\lambda}(n))$ for $n \in \mathrm{Nom}$. We claim that $h[S] \subseteq T_k$ for this g. This claim, combined with Lemma 3.2, is enough for the proof.

We prove the claim by induction on k. For $k = 0$, take $s \in S$ and assume $h(s) \Vdash n$ for some $n \in \mathrm{Nom}$. By Lemma 3.2, $s \models n$, that is, $\tilde{\lambda}(n) = s$. Therefore $g(n) = h(\tilde{\lambda}(n)) = h(s)$. The other condition can be checked similarly. Thus we have $h(s) \in T_0$.

Assume $h[S] \subseteq T_k$. Then by Corollaries 3.5 and 3.7, $h[S] \subseteq T_{k+1}$. This completes the proof. ∎

3.4 Completeness

In this subsection we prove that the decision procedure described in Section 3.2 is complete, namely if the procedure succeeds, there is a functional Kripke structure $\mathcal{K} = (S, R, \lambda)$ and $s \in S$ such that $\mathcal{K}, s \models \tilde{\varphi}$.

First we prepare a lemma that is used later in this section.

Lemma 3.8 Assume $D \in \mathscr{D}_\mu$, $E \subseteq D \cap \mathrm{Lean}$, and E is closed under downward order of x within D, that is, $\psi \in D \cap \mathrm{Lean}$, $\psi' \in E$, $x(\psi) < x(\psi')$ implies $\psi \in E$. Then for φ_1, $\varphi_2 \in D$, if $x(\varphi_1) < x(\varphi_2)$ then $x^E(\varphi_1) < x^E(\varphi_2)$ or $x^E(\varphi_1) = x^E(\varphi_2) = \infty$.

Proof Double induction, first on φ_1, then on φ_2, with base cases for Lean and general cases for operators $\vee$ and $\wedge$. We omit details, which are tedious but not difficult.

Let g be the naming function for which the procedure succeeds. We use symbols such as T_K, J, and t_I in the same meaning as in the description of the procedure.

We assume without loss of generality that $\mathscr{D}_\mu \neq \varnothing$ since formula $\varphi_\mathrm{I} \vee \mu X X$ is equivalent to φ_I and $\mathscr{D}_\mu \neq \varnothing$ for this formula. We fix an element D_0 of $\mathscr{D}_\mu$ and a cyclic permutation τ on $\mathscr{D}_\mu$.

Let Nom' be a set of representatives of the equivalence class induced by the equivalence relation $\{(n, n') \in \mathrm{Nom} \mid g(n') \Vdash n\}$, that is,

- If $n_1, n_2 \in \mathrm{Nom}'$ and $n_1 \neq n_2$, then $g(n_1) \not\Vdash n_2$.
- For any $n \in \mathrm{Nom}$ there is $n' \in \mathrm{Nom}'$ such that $g(n') \Vdash n$.

We construct a (possibly infinite) forest (W, R^W), where W is the underlying set and R^W is the forest relation on W, together with functions $t : W \to T_K$, $l : R^W \to \mathrm{Mod}$. $D : W \to \mathscr{D}_\mu$, $E : W \to D_\mu \cap \mathrm{Lean}$, and $j : W \to \omega$, During the construction we will keep the following invariants: for $w, w' \in W$,

$$j(w) > 0 \implies (t(w), E(w)) \in V_{j(w)} \setminus V_{j(w)-1}.$$

At the first stage of the construction, we create elements w_I and w_n for $n \in \mathrm{Nom}'$ and let $W = \{w_\mathrm{I}\} \cup \{w_n \mid n \in \mathrm{Nom}'\}$ and $R^W = \varnothing$. Also we define $t(w_\mathrm{I}) = t_\mathrm{I}$ and $t(w_n) = g(n)$ for $n \in \mathrm{Nom}'$. And for $w \in W$, we define $D(w) = D_0$, $E(w) = \{\varphi \in D_0 \cap \mathrm{Lean} \mid t(w) \Vdash \varphi\}$, and $j(w) = \min\{j \in \omega \mid (t(w), E(w)) \in V_j\}$. The invariant holds for the first stage: since $t(w) \in T_K$, $t(w)$ is μ-consistent and therefore the set in the right hand side of the definition of $j(w)$ is not empty.

At the second and succeeding stage of the loop, we pick a leaf node w of W which has not yet processed and is located at the shallowest level of the forest among such nodes. If $w \neq w_n$ and there is $n \in \mathrm{Nom}'$ such that $t(w) = g(n)$, there is nothing to do. Otherwise, for each $\varphi \in \mathrm{Lean}$ in the form of $\langle m \rangle \varphi'$ such that $t(w) \Vdash \varphi$, we create a node w' add it to W. A pair (w, w') is added to R^W and we define $l(w, w') = m$.

If $j(w) = 0$, since $t(w) \in T_K$ and therefore it is $\Diamond$-consistent, we can find a $t' \in T_K$ such that $(t(w), t') \in \mathrm{Tr}(m)$, and set $t(w') = t'$. Also we define $D(w') = \tau(D(w))$,

$E(w') = \{\psi \in D(w') \cap \mathrm{Lean} \mid t(w') \Vdash \psi\}$, and $j(w') = \min\{j \in \omega \mid (t(w'), E(w')) \in V_j\}$. The invariant can be checked as before.

If $j(w) > 0$, from the invariant there is $(t', E') \in V_{j(w)-1}$ that satisfies the conditions in the description of the procedure. We define $t(w') = t'$, $D(w') = D(w)$, $E(w') = E'$ and $j(w') = \min\{j \in \omega \mid (t', E') \in V_j\}$. The invariant trivially holds.

In the case of the FUNC procedure, the above is only applied for $m \in \mathrm{Mod} \setminus \mathrm{MS}$. And for all $m \in \mathrm{MS}$ that satisfies the following conditions, we create a node w' and add it to W and set $l(w, w') = m$.

- There is no $\hat{w} \in W$ such that $(\hat{w}, w) \in R^W$ and $l(\hat{w}, w) = \overline{m}$.
- There is $\varphi \in \mathrm{Lean}$ in the form of $\langle m \rangle \varphi'$ and $t(w) \Vdash \varphi$.

The rest is quite similar to the non-FUNC procedure. If $j(w) = 0$, take $t' \in T_K$ such that $(t, t') \in \mathrm{Tr}(m)$ and define $t(w') = t'$, $D(w') = \tau(D(w))$, $E(w') = \{\psi \in D(w') \cap \mathrm{Lean} \mid t(w') \Vdash \psi\}$, and $j(w') = \min\{j \in \omega \mid (t(w'), E(w')) \in V_j\}$. Otherwise, take $(t', E') \in V_{j(w)-1}$ that satisfies the conditions described in the FUNC procedure, and define $t(w') = t'$, $D(w') = D(w)$, $E(w') = E'$ and $j(w') = \min\{j \in \omega \mid (t', E') \in V_j\}$. The invariant can be checked easily.

That completes the construction of the forest and the auxiliary functions. We summarize the properties of the forest as a lemma.

Lemma 3.9 The following holds for all $w \in W$:

(1) $j(w) > 0 \implies (t(w), E(w)) \in V_{j(w)} \setminus V_{j(w)-1}$.

(2) $E(w) \subseteq D(w)$

(3) If $t(w) \Vdash \langle m \rangle \varphi$, where $m \in \mathrm{Mod}$ and $\langle m \rangle \varphi \in \mathrm{Lean}$, there exists $w' \in W$ such that:

- $(t(w), t(w')) \in \mathrm{Tr}(\langle m \rangle \varphi)$.
- If $\langle m \rangle \varphi \in E(w)$ then $t(w')^{E(w')} \Vdash \varphi$.
- In the case of the non-FUNC procedure, $(w, w') \in R^W$.

(4) If $(w, w') \in R^W$ and $l(w, w') = m$, $(t(w), t(w')) \in \mathrm{Tr}(m)$ and $(t(w'), t(w)) \in \mathrm{Tr}(\overline{m})$.

(5) Suppose $(w, w') \in R^W$.

- If $j(w) > 0$, then $j(w) > j(w')$, $D(w') = D(w)$, and for any formula ψ, $[m]\psi \in E(w) \implies t(w')^{E(w')} \Vdash \psi$, where $m = l(w, w')$.

- If $j(w) = 0$, then $E(w) = \varnothing$, $D(w') = \tau(D(w))$ and $E(w') = \{\varphi \in D(w') \cap \mathrm{Lean} \mid t(w') \Vdash \varphi\}$.

(6) In the case of FUNC procedure, for each $m \in \mathrm{MS}$, w has at most one $w' \in W$ such that $(w, w') \in R^W$.

Proof (1) is the invariant we keep through the construction and the others can be checked easily. ∎

Lemma 3.10 Suppose that $(w_i \mid i \in \omega)$ is a sequence of elements of W and $(w_i, w_{i+1}) \in R^W$ for all $i \in \omega$. Also suppose $D \in \mathscr{D}_\mu$ and $i \in \omega$.

(1) There exists $k \in \omega$ such that $i \leq k$ and $E(w_k) = \varnothing$.

(2) There exists $k \in \omega$ such that $i \leq k$, $D(w_k) = D$, and $E(w_k) = \{\varphi \in D \cap \mathrm{Lean} \mid t(w_k) \Vdash \varphi\}$.

Proof

(1) Clear from Lemma 3.9 (5).

(2) By Lemma 3.9 (5), there is $k' \geq$ such that $D(w_{k'}) = \tau(D(w_i))$. Since $\mathscr{D}_\mu$ is finite and τ is a cyclic permutation on $\mathscr{D}_\mu$, by repeating this finitely many times, we have $k > i$ such that $j(w_{k-1}) = 0$, $E(w_{k-1}) = 0$, $D(w_k) = \tau(D(w_{k-1})) = D$, $E(w_k) = \{\varphi \in D \cap \mathrm{Lean} \mid t(w_k) \Vdash \varphi\}$. ∎

We define an equivalence relation $\sim$ on W: $w_1 \sim w_2$ if and only if $w_1 = w_2$ or there exists $n \in \mathrm{Nom}'$ such that $t(w_1) = t(w_2) = g(n)$. Using this, we define a Kripke structure $\mathcal{K} = (S, R, \lambda)$ as follows:

- The underlying set S is W divided by the equivalence relation $\sim$. If there is no confusion, for $w \in W$, we also write w for the equivalence class that contains w.
- $R(m) = \{(w_1, w_2) \mid (w_1, w_2) \in R^W,\ l(w_1, w_2) = m \text{ or } l(w_2, w_1) = \overline{m}\}$.
- $\lambda(p) = \{w \mid w \in S,\ t(w) \Vdash p\}$.

We will show that $\mathcal{K}, w_{\mathrm{I}} \models \varphi_{\mathrm{I}}$. For this purpose we take a choice function c that further satisfies the following conditions:

- For $\varphi = \varphi_0 \vee \varphi_1 \in \mathrm{cl}(\varphi_{\mathrm{I}})$ and $w \in S$, let $(x, y) = t = t(w), D = D(w), E = E(w)$. If $\varphi \in D_\mu$ and $x(\varphi) < \infty$, then for $j = 0, 1$:
  - $\varphi_j \notin D$, $x(\varphi_j) < \infty$, $\varphi_{1-j} \in D \implies c(\varphi, w) = \varphi_j$

- $\varphi_0, \varphi_1 \in D$, $y(\varphi_j) < y(\varphi_{1-j}) \implies c(\varphi, w) = \varphi_j$.
- $\varphi_0, \varphi_1 \in D$, $y(\varphi_0) = y(\varphi_1)$, $x(\varphi_j) < x(\varphi_{1-j}) \implies c(\varphi, w) = \varphi_j$.
- $\varphi_0, \varphi_1 \in D$, $y(\varphi_0) = y(\varphi_1)$, $x(\varphi_j) = x(\varphi_{1-j})$, $x^E(\varphi_j) < x^E(\varphi_{1-j}) \implies c(\varphi, w) = \varphi_j$.

- For any formula φ and $w \in S$, if $t(w) \Vdash \langle m \rangle \varphi$, then the following is satisfied for $w' = c(\varphi, w)$.
  - $(t(w), t(w')) \in \mathrm{Tr}(\langle m \rangle \varphi)$.
  - $\langle m \rangle \varphi \in E(w) \implies t(w')^{E(w')} \Vdash \varphi$.
  - $(w, w') \in R^W$ in the case of the non-FUNC procedure.

  This is guaranteed by Lemma 3.9 (3). Note that in the case of $t(w) = g(n)$ for some $n \in \mathrm{Nom}'$, there is only one w, namely, w_n, in the equivalence class that has successors.

We define a region Z of the underlying set $\mathrm{cl}(\varphi_{\mathrm{I}}) \times S$ of the arena by $Z = \{(\varphi, w) \in \mathrm{cl}(\varphi_{\mathrm{I}}) \times S \mid t(w) \Vdash \varphi\}$. We will check that c and Z satisfy the conditions in Lemma 2.18. Once it is established, $\mathcal{K}, w_{\mathrm{I}} \models \varphi_{\mathrm{I}}$ follows since $(\varphi_{\mathrm{I}}, w_{\mathrm{I}}) \in Z$.

Conditions (1) and (2) can be checked easily from the definitions.

To show Condition (3), we assume on the contrary that there exist $D \in \mathscr{D}_\mu$, $\varphi_0 \in D$, $w_0 \in S$, and a strategy σ_1 for Player 1 such that $(\varphi_0, w_0) \in Z$, $(\sigma(c) * \sigma_1)(\varphi_0, w_0) = ((\varphi_k, w_k) \mid k \in \omega)$ is an infinite trace, and $\varphi_k \in D$ for all $k \in \omega$.

Lemma 3.11 Nominals occur only finitely many times in the sequence. Namely, there is $k_0 \in \omega$ such that for all $k \geq k_0$ and $n \in \mathrm{Nom}$ $w_k \not\Vdash n$.

Proof We observe that

$$y_{t(w_k)}(\varphi_k) \geq y_{t(w_{k+1})}(\varphi_{k+1}) \tag{3.1}$$

holds. When $\varphi_k = \varphi \vee \psi$, it is clear since c is a choice function. Note that $x(\varphi) > x(\psi)$ implies $y(\varphi) \geq y(\psi)$. When $\varphi_k = \varphi \wedge \psi$, $\mu X \varphi$, or $\nu X \varphi$, (3.1) follows from the definition of y for such formulae. When $\varphi_k = \langle m \rangle \varphi$, it follows from the fact that c is a choice function. For $\varphi_k = [m]\varphi$, use Lemma 3.9 (4) and for $\varphi_k = @n\,\varphi$, use the definition of T_0. And φ_k cannot be an atomic formula or its negation since $\varphi_k \in D$. That covers all the cases and we have established (3.1).

Therefore there is $l_1 \in \omega$ and $i \in \omega$ such that for all $k \geq l_1$, $y_{t(w_k)}(\varphi_k) = i$. Since φ_{I} is guarded, so are all φ_k. Therefore there is $l_2 \geq l_1$ such that $\varphi_{l_2} \in$ Lean. If $k > l_2$

and $t(w_k)$ has a name, $y_{t(w_k)}(\varphi_k)$ must be strictly smaller than $y_{t(w_{l_2})}(\varphi_{l_2})$, which is impossible. The proof completes by taking $k_0 = l_2 + 1$. ∎

Lemma 3.12 For any subsequence $((\varphi_k, w_k) \mid K \leq k \leq L)$, where $k_0 \leq K \leq L \in \omega$, if $w_K = w_L$ (we denote it by w), then $x_{t(w)}(\varphi_K) \geq x_{t(w)}(\varphi_L)$. Moreover, if there is k' such that $K < k' < L$ and $w_K \neq w_{k'}$, $x_{t(w)}(\varphi_K) > x_{t(w)}(\varphi_L)$.

Proof Induction on the length of the subsequence, namely $L - K$. It is trivial when $L - K = 0$, so we assume $K > L$.

If φ_K is in the form of $\varphi \vee \psi$, $\varphi \wedge \psi$, $\mu X\varphi$, or $\nu X\varphi$, $w_{K+1} = w_K$ and $x_{t(w)}(\varphi_K) \geq x_{t(w)}(\varphi_{K+1})$. (The first case comes from the fact that c is a condition and the rest the definition of x). Then using the induction hypothesis, we have the conclusion. As φ_K cannot be in the form of $@n\,\varphi$ since $K \geq k_0$ or an atomic formula or its negation since it is an element of D, the remaining cases for φ_K are $\langle m\rangle\varphi_{K+1}$ and $[m]\varphi_{K+1}$ where $m \in \mathrm{Mod}$.

In such cases $w_{K+1} \neq w_K$, since W is a forest and does not have a self-loop. Let M be the least index of the subsequence such that $K < M$ and $w_M = w_K$. Clearly $K + 1 \leq M - 1$ and $w_{K+1} = w_{M-1}$ since W is a forest. Let $w' = w_{K+1} = w_{M-1}$. Then by the induction hypothesis, we have

$$x_{t(w')}(\varphi_{K+1}) \geq x_{t(w')}(\varphi_{M-1}) \tag{3.2}$$

$$x_{t(w)}(\varphi_M) \geq x_{t(w)}(\varphi_L) \tag{3.3}$$

Therefore if we show

$$x_{t(w)}(\varphi_K) > x_{t(w)}(\varphi_M) \tag{3.4}$$

we have $x_{t(w)}\varphi_K > x_{t(w)}\varphi_L$ as desired by combining (3.3) and (3.4).

Note that φ_{M-1} is either $\langle\overline{m}\rangle\varphi_M$ or $[\overline{m}]\varphi_M$ since $w_{M-1} \neq w_M$. To prove (3.4), it is enough to show either $\mathrm{LoopFree}(t(w), t(w'), \varphi_K)$ or $\mathrm{LoopFree}(t(w'), t(w), \varphi_{M-1})$ since we have (3.2).

First assume $\varphi_K = \langle m\rangle\varphi_{K+1}$. Then since c is a choice function, $(t(w), t(w')) \in \mathrm{Tr}(\varphi_K)$. Therefore if $\varphi_{M-1} = [\overline{m}]\varphi_M$, $\mathrm{LoopFree}(t(w), t(w'), \varphi_K)$ holds. What happens if, on the contrary, $\varphi_{M-1} = \langle\overline{m}\rangle\varphi_M$? In the case of the FUNC procedure, we still have $\mathrm{LoopFree}(t(w), t(w'))$ from the definition of $\mathrm{Tr}(m)$ and $\mathrm{Tr}(\langle m\rangle\varphi)$. In the case of the non-FUNC procedure, the combination is not possible because then we have $(w, w') \in R^W$ and $(w', w) \in R^W$ by the condition of c, contradicting the fact that W is a forest.

The same argument can be used if we assume $\varphi_{M-1} = \langle m \rangle \varphi_M$. Then the only case remained is $\varphi_K = [m]\varphi_{K+1}$ and $\varphi_{M-1} = [\overline{m}]\varphi_M$. Since (φ_{K+1}, w_{K+1}) is chosen by the Player 1, $(w_K, w_{K+1}) \in R(m)$. Therefore either $(w_K, w_{K+1}) \in R^W$ and $l(w_K, w_{K+1}) = m$ or $(w_{K+1}, w_K) \in R^W$ and $l(w_{K+1}, w_K) = \overline{m}$. Then by Lemma 3.9 (4), $(t(w_K), t(w_{K+1})) \in \mathrm{Tr}(m)$, and therefore we have LoopFree($t(w)$, $t(w')$, φ_K) as desired. ∎

Lemma 3.13 For any $w \in W$, the set $\{k \in \omega \mid w = w_k\}$ is finite.

Proof If $\{k \in \omega \mid w = w_k\}$ is infinite, by Lemma 3.12 $(x(w_k) \mid k \geq k_0,\ w = w_k)$ is an infinite descendant sequence and strictly decrease at infinitely many times since φ_{I} is guarded. That is impossible. ∎

For $w \in W$, we define $m(w) = \max\{k \in \omega \mid w_k = w\}$. And we define a sequence $(L(i) \mid i \in \omega)$ of natural numbers by $L(0) = m(w_{k_0})$ and $L(i+1) = m(w_{L(i)+1})$. It is clear that $w_{L(i+1)} = w_{L(i)+1}$. Also note that $\varphi_{L(i)} \in \mathrm{Lean}$ since $w_{L(i)+1} \neq w_{L(i)}$.

Lemma 3.14 There is a natural number $i_0 \in \omega$ such that for all $i \in \omega$, $i \geq i_0 \implies (w_{L(i)}, w_{L(i+1)}) \in R^W$.

Proof Since $w_{L(i)} \neq w_{L(i)+1}$ and $L(i) \geq k_0$, $\varphi_{L(i)}$ must be either $\langle m \rangle \varphi_{L(i)+1}$ or $[m]\varphi_{L(i)+1}$ for some $m \in \mathrm{Mod}$. Therefore $(w_{L(i)}, w_{L(i)+1}) \in R(m)$, that is, either $(w_{L(i)}, w_{L(i)+1}) \in R^W$ or $(w_{L(i)+1}, w_{L(i)}) \in R^W$ holds. Since W is a forest, the relation R^W is well-founded, so there is at least one $i_0 \in \omega$ that $(w_{L(i)}, w_{L(i)+1}) \in R^W$. Then we show by induction on i if $i \geq i_0$ then $(w_{L(i)}, w_{L(i+1)}) \in R^W$. The case $i = i_0$ is trivial. The case $i + 1$: as we see before either $(w_{L(i+1)}, w_{L(i+1)+1}) \in R^W$ or $(w_{L(i+1)+1}, w_{L(i+1)}) \in R^W$ holds. If the latter is the case, since $(w_{L(i)}, w_{L(i)+1}) \in R^W$, $w_{L(i)+1} = w_{L(i+1)}$, and W is a forest, we have $w_{L(i)} = w_{l_{i+1}+1}$, contradicting the fact that $L(i)$ is the largest index of $w_{L(i)}$ noticing $l_i < L(i+1) + 1$. Therefore the former should be the case, that is, $(w_{L(i+1)}, w_{L(i+1)+1}) \in R^W$. ∎

Let us denote by R^{W+} the reflexive transitive closure of R^W.

Lemma 3.15 For $j \in \omega$, If $k > L(i_j)$, $(w_{L(i_j)}, w_k) \in R^{W+}$.

Proof Induction on k. The base case $k = L(i_j) + 1$ is clear from Lemma 3.14. Assume for the general case that $(w_{L(i_j)}, w_k) \in R^{W+}$. Then if $(w_{L(i_j),w_{k+1}}) \notin R^{W+}$, since $k \leq k_0$ either $\varphi_k = \langle m \rangle \varphi_{k+1}$ or $\varphi_k = [m]\varphi_{k+1}$ and $w_{k+1} = w_{L(i_j)}$, contradicting the fact that $L(i_j)$ is the maximum index. ∎

By applying Lemma 3.10 (2) to the sequence $(w_{L(i)} \mid i_0 \leq i < \omega)$, we find $i_1 \in \omega$ such that $i_1 \geq i_0$, $D(w_{L(i_1)}) = D$, $E(w_{L(i_1)}) = \{\varphi \in D \cap \mathrm{Lean} \mid t(w_{L(i_1)}) \Vdash \varphi\}$. In particular, $\varphi_{L(i_1)} \in E_{L(i_1)}$.

Lemma 3.16 For all $k \geq L(i_1)$, $D(w_k) = D$ and $t(w_k)^{E(w_k)} \Vdash \varphi_k$

Proof Induction on k. The base case $k = L(i_1)$ is trivial.

Case $\varphi_k = \varphi_{k+1} \vee \psi$. Since $w_k = w_{k+1}$, $D(w_{k+1}) = D$. Note that $\psi \in D$. Let $(x, y) = t(w_k)$ and $E = E(w_k)$. Since c is a choice function, either $x(\varphi_{k+1}) < x(\psi)$ or $x(\varphi_{k+1}) = x(\psi)$ and $x^E(\varphi_{k+1}) \leq x^E(\psi)$. When $x(\varphi_{k+1}) < x(\psi)$, since $t(w_{k+1}) \Vdash \varphi_{k+1}$ and $w_k = w_{k+1}$, $x^E(\varphi_{k+1}) < x^E(\psi)$ by Lemma 3.8. Therefore in both cases we have $x^E(\varphi_{k+1}) = x^E(\varphi_k) < \infty$.

Cases $\varphi_k = \varphi_{k+1} \wedge \psi$ and $\mu X \psi$ are easy and we omit them.

The remaining cases are $\varphi_k = \langle m \rangle \varphi_{k+1}$ and $[m]\varphi_{k+1}$. In these two cases, either $(w_{k+1}, w_k) \in R^W$ or $(w_k, w_{k+1}) \in R^W$ holds. Assume first $(w_{k+1}, w_k) \in R^W$. Clearly $D(w_{k+1}) = D(w_i) = D$. Since for all j such that $L(i_1) \leq j < k$, one of $(w_{j+1}, w_j) \in R^W$, $(w_j, w_{j+1}) \in R^W$, or $w_j = w_{j+1}$ holds, so by Lemma 3.15, there is i such that $L(i_1) \leq i < k+1$ such that $w_i = w_{k+1}$. Let $E = E(w_i)$ and $(x, y) = t(w_i)$. By induction hypothesis we have $x(\varphi_i) < \infty$. On the other hand we have $x(\varphi_i) > x(\varphi_{k+1})$ by Lemma 3.12. Therefore by Lemma 3.8, $x^E(\varphi_{k+1}) < x^E(\varphi_i) < \infty$.

Assume next $(w_k, w_{k+1}) \in R^W$. If $\varphi_k = \langle m \rangle \varphi_{k+1}$, we have $t(w_{k+1})^{E(w_{k+1})} \Vdash \varphi_{k+1}$ since $\varphi_k \in E(w_k)$ by induction hypothesis and c is a choice function. If $\varphi_k = [m]\varphi_{k+1}$, $j(w_k) > 0$ since otherwise $\varphi_k \in E(w_k)$, but $E(w_k) = \varnothing$ by Lemma 3.9 (5). Then again by Lemma 3.9 (5), $D(w_{k+1}) = D(w_k) = D$ and $t(w_{k+1})^{E(w_{k+1})} \Vdash \varphi_{k+1}$. ∎

Lemma 3.10 (1) and Lemma 3.16 contradicts each other. Thus condition (3) of Lemma 2.18 has established.

Theorem 3.17 Assume a formula $\varphi_{\mathrm{I}} \in L_0(\mu_{\mathrm{AF}}, @, {}^{-})$ is given. If the non-FUNC version of the decision procedure described in Section 3.2 succeeds, there is a Kripke structure $\mathcal{K}$ and $w_{\mathrm{I}} \in |\mathcal{K}|$ such that $\mathcal{K}, w_{\mathrm{I}} \models \varphi_{\mathrm{I}}$. If the FUNC version succeeds, unless both nominals and reverse modalities occur in φ_{I}, there is a functional Kripke structure $\mathcal{K}$ and $w_{\mathrm{I}} \in |\mathcal{K}|$ such that $\mathcal{K}, w_{\mathrm{I}} \models \varphi_{\mathrm{I}}$.

Proof Since we have checked the conditions (1), (2), and (3) of Lemma 2.18 for c and Z, $\mathcal{K}, w_{\mathrm{I}} \models \varphi_{\mathrm{I}}$ holds since $(w_{\mathrm{I}}, \varphi_{\mathrm{I}}) \in Z$.

For the second half, we also need to check that $\mathcal{K}$ constructed in this section is functional. When there is no nominal, it directly follows from Lemma 3.9 (6), since

in this case the equivalence relation $\sim$ coincides with the equality. When there is no reverse modality $(w, w_1), (w, w_2) \in R(m)$ implies $(w, w_1), (w, w_2) \in R^W$ and again by Lemma 3.9 (6), we have $w_1 = w_2$. ∎

3.5 Discussion on Complexity

In Section 3.2, we provided decision procedures for three satisfiability judging problems $\mathrm{Sat}(\mu_{\mathrm{AF}}, @, {}^{-})$, $\mathrm{Sat}(\mu_{\mathrm{AF}}, {}^{-}, \mathrm{func})$, and $\mathrm{Sat}(\mu_{\mathrm{AF}}, @, \mathrm{func})$. Their theoretical complexities have already been known: they are all EXPTIME-complete [7]. Our procedures are simpler than known procedures for the full μ-calculus, because we confine ourselves to the alternation-free part of the μ-calculus. The simplicity of our procedures are evaluated in two aspects.

First, they have less computation time. The time complexity of the known procedure for $\mathrm{Sat}(\mu)$ described in [32] is $2^{\mathcal{O}(n^6)}$, where n is the length of the formula. The time complexity of our decision procedures for $\mathrm{Sat}(\mu_{\mathrm{AF}}, {}^{-}, \mathrm{func})$, $\mathrm{Sat}(\mu_{\mathrm{AF}}, @, {}^{-})$, and $\mathrm{Sat}(\mu_{\mathrm{AF}}, @, \mathrm{func})$ are $2^{\mathcal{O}(n \log n)}$, $2^{\mathcal{O}(n^2)}$, and $2^{\mathcal{O}(n^2)}$ respectively as we will see later in this section.

Second, and more importantly, our procedures are based on the tableau method and consist of set operations. Therefore we can easily implement them with BDD [10] as we will see Section 3.6 and Chapters 4 and 5. On the other hand, known decision procedures contain the determinization of automata for infinite words and the calculation of the winning regions of parity games, and to the best of our knowledge, they have not been implemented.

In the rest of this section, we calculate the time complexity of our procedures. Let n be the length of the formula φ_{I}.

Proposition 3.18 Time complexity of the decision procedure provided in Section 3.2 is $2^{\mathcal{O}(n^2)}$ for $\mathrm{Sat}(\mu_{\mathrm{AF}}, {}^{-}, \mathrm{func})$ and $2^{\mathcal{O}(n \log n)}$ for $\mathrm{Sat}(\mu_{\mathrm{AF}}, @, \ )$ and $\mathrm{Sat}(\mu_{\mathrm{AF}}, @, \mathrm{func})$.

Proof Let us count the number of naming functions. The domain of a naming function is Nom and the range is $\mathscr{P}(\mathrm{Lean})$. Such functions exist at most $(2^n)^n = 2^{n^2}$. In the case of $\mathrm{Sat}(\mu_{\mathrm{AF}}, {}^{-}, \mathrm{func})$, the number is 1 since there is no nominals. Therefore it is enough to show that the time complexity of the sub-procedure for a naming function is $2^{\mathcal{O}(n \log n)}$.

Next we count the number A of nodes in the tableau. Component x can be regarded as a function from Lean to $\{0, 1, \ldots, n-1\}$. The number of such functions are at most

n^n. Component y can be regarded as a function from Lean to $\{0, 1, \ldots, n^2 - 1\}$. The number of such functions are at most $(n^2)^n$. Therefore $A \leq n^n \cdot (n^2)^n = 2^{\mathcal{O}(n \log n)}$.

The body of the sub-procedure is a double-loop. In the outer loop, $T_0 \supset T_1 \supset \cdots \supset T_K$ are calculated. Since $|T_0| \leq A$, the number of repetition, K, does not exceed A. The inner loop calculates $V_0 \subset V_1 \subset \cdots \subset V_J$. Since each V_j is a subset of $T_0 \times \mathscr{P}(\text{Lean})$, the number of repetition, J, does not exceed $B = A \cdot 2^n$.

In each repetition of the inner loop, V_{j+1} is calculated from V_j. This is done by checking if each (t, E) is an element of V_{j+1}. The number of such (t, E) does not exceed B. The check for (t, E) consists of checking conditions (a), (b), and (c). They can be performed in polynomial time C once pair (t', E') is fixed and the number of candidates (t', E') does not exceed B.

Therefore time required to perform the sub-procedure is $A \cdot B \cdot B \cdot C \cdot B = C \cdot A^4 \cdot (2^n)^3 = 2^{\mathcal{O}(n \log n)}$. ∎

3.6 An Experimental Implementation

We will discuss applications of the satisfiability judgment based on our procedures in Chapters 4 and 5. Nominals play an important role in these applications. In this section, we report an implementation the decision procedure for $\mathrm{Sat}(\mu_{\mathrm{AF}}, {}^{-})$. This combination has applications to the analysis of transformation manipulating XML documents [71].

Table 3.1 shows the results for a few formulae. The column v is the number of the BDD variables, n is the number of the BDD nodes, and t is the execution elapsed time in milliseconds.

The formulae are defined in Figure 3.1. The intention of `lap`n, for example, is "p_0 holds at the start point and if p_{i-1} holds one can find a point where p_i holds by following arrows labelled by a. But one cannot find a point where p_n holds and connected to a point where p_0 holds with reverse arrows labelled by a." No model satisfies this formula. There is a $D \in \mathscr{D}_\mu$ such that the size of BForm_D is $2n$ in the formula `loop`n, while BForm_D is an empty set in the other formulae for all $D \in \mathscr{D}_\mu$. The formula `tree`n are satisfiable but they do not have finite models, while the other formulae are unsatisfiable.

We have also tested the implementation against automatically generated formulae. Refer to [61] for details.

The experimental implementation is written in Java with JavaBDD 1.0 [39], which calls BuDDy [11]. We use the following values for the tuning parameters: nodenum =

Table 3.1: Experiments

lap$n$			
n	v	b	t
2	15	1.7×10^3	560
6	35	1.7×10^4	610
10	55	2.0×10^5	1020
14	75	2.0×10^6	7140

loop$n$			
n	v	b	t
1	21	4.2×10^3	352
3	49	6.7×10^4	453
5	73	5.9×10^5	1047
7	113	4.1×10^6	43016

tree$n$			
n	v	b	t
4	16	1.4×10^3	313
8	32	6.9×10^3	343
12	48	9.8×10^4	453
16	64	1.4×10^6	5875

$$
\begin{aligned}
\texttt{lap}n &= p_0 \wedge \bigwedge_{i=1}^{n} \nu X((p_{i-1} \rightarrow \mu Y(p_i \vee \langle a \rangle Y)) \wedge [a]X) \\
&\qquad \wedge \neg \mu Z((p_n \wedge \mu W(p_0 \vee \langle \overline{a} \rangle W)) \vee \langle a \rangle Z))) \\
\alpha_0 &= p \vee [a_0](p \vee [\overline{a_0}]x_1) \\
\alpha_n &= p \vee [a_n]\mu X_n([\overline{a_n}]X_{n+1} \vee \alpha_{n-1}) \\
\beta_0 &= \neg p \\
\beta_n &= \neg p \wedge \langle a_{n-1} \rangle \beta_{n-1} \\
\texttt{loop}n &= \mu X_{n+1}(\alpha_n) \wedge \beta_n \\
\texttt{tree}0 &= \mathbf{true} \\
\texttt{tree}n &= \nu Y_n(\langle a_n \rangle(\mu X_n([\overline{a_n}]X_n) \wedge Y_n \wedge \texttt{tree}(n-1)))
\end{aligned}
$$

Figure 3.1: Definitions of Formulae

2^{20}, maxincrease = 2^{18}, cachesize = variable, cacheratio = 1/64.0. Our experiments have been performed on a Pentium 4 2.4 GHz machine with 512 megabytes of RAM, running Microsoft Windows XP, Sun Java Development Kit version 1.5.0.

3.7 Related Work

Studies have been conducted related to the satisfiability of the extensions discussed in this chapter.

The propositional modal μ-calculus was introduced by Kozen [41] and he proved that it is decidable. Emerson and Jutla proved that the complexity of its satisfiability problem is EXPTIME-complete [28]. Bonatti `et al.` proved that the problems $\mathrm{Sat}(\mu, {}^{-}, \mathrm{func})$, $\mathrm{Sat}(\mu, @, {}^{-})$, and $\mathrm{Sat}(\mu, @, \mathrm{func})$ are all decidable and their complexity is EXPTIME-complete [7]. The result is more strong: they use a operator called graded modality instead of functional Kripke structures. The modality can express "there are less than n successors" for natural number n, and a functional relation can be regarded as a special case of $n = 1$.

Bonatti and Peron showed that the problem $\mathrm{Sat}(\mu, @, {}^{-}, \mathrm{func})$ is undecidable [8]. They also used the graded modality. By checking their proof, one can see that $\mathrm{Sat}(\mu_{\mathrm{AF}}, @, {}^{-}, \mathrm{func})$ is also undecidable.

Tobies showed that $\mathrm{Sat}(@, {}^{-}, \mathrm{func})$ is decidable [70]. Its complexity is NEXPTIME-complete.

The following researches are on implementations of decision procedures. There are variants of modal logics that have decision procedure based on the tableau method. For example, Emerson gave the decision procedure for CTL based on the tableau method [27]. This procedure is easy to implement. Pan *et al.* gave effective implementations of the decision procedure using the BDD for the minimal modal logic **K** [50, 51]. MONA [34] is a famous tool that implements decision procedures for WS1S and WS2S. This tool also utilizes BDD for its implementation. Eijck built a theorem prover for hybrid logics [72] based on the tableau method.

The decision procedures discussed in this section are not covered by any of the abovementioned implementations. The first three does not contain nominals and the last does not contain fixed-point operators.

Chapter 4

Application to Shape Analysis

This chapter and the next chapter discusses our first application, "shape analysis using modal logics".

In this chapter we provide a framework to verify the properties of programs that manipulate pointers by using the predicate abstraction technique.

We use a small programming language called PML, which has pointer manipulation features as languages C or Java. We provide a method to calculate the weakest preconditions of the predicate expressed by the modal logic formulae for the statements of the language. Combining this calculation and a decision procedure for the logic, an abstract transition system is automatically constructed for a given program. This transition system is a sound abstraction with regard to the properties expressed in a property description language based on LTL, which we shall also define.

We implement a prototype system called MLAT based on the framework [65, 62]. The decision procedure used in MLAT is a variant of our decision procedures described in Chapter 3.

4.1 Pointer Structure

In this section we define a mathematical structure that expresses program heaps.

Typically numerical data and pointers are stored in heaps. To model heaps there are at least two ways to view them. One is to view a heap as an array of integers such as in [53]. Pointers are stored in the array by interpreting integers as addresses. The other is to view it as a set of nodes that store pointers such as in [57]. We take the latter way to apply modal logics.

Let Var, BFld and PFld be nonempty final sets. We call an element of Var a program variable, an element of BFld a Boolean field, and an element of PFld a

pointer field.

Definition 4.1 We call a tuple $\mathcal{S} = (N, p, v, \rho, \mathrm{nil})$ satisfying the following conditions a *pointer structure*.

- N is a finite set. An element of N is called a *node*.
- $p : N \times \mathrm{PFld} \to N$ is a function.
- $v : \mathrm{BFld} \to \mathscr{P}(N)$ is a function.
- $\rho : \mathrm{Var} \to N$ is a function.
- nil is a member of N. ∎

For pointer structure $\mathcal{S} = (N, p, v, \rho, \mathrm{nil})$, we write $N_{\mathcal{S}}$, $p_{\mathcal{S}}$, $v_{\mathcal{S}}$, $\rho_{\mathcal{S}}$ and $\mathrm{nil}_{\mathcal{S}}$ for N, p, v, ρ and nil, respectively

We denote by PStr' the class of all pointer structures and let $\mathrm{PStr} = \mathrm{PStr}' \cup \{\bot\}$.

The intuitive meaning is as follows. We regard a heap as a finite set of nodes. Set N is the underlying set. There are several pointers stored in each node and they are distinguished by their names. The names are called pointer fields. When pointer field f of node n_1 points to node n_2, we represent it by $p(n_1, f) = n_2$. Each node also stores several Boolean values and their names are called Boolean fields. When the value of Boolean field b of node n is **true** (or **false**, respectively), we represent it by $n \in v(b)$ (or $n \notin v(b)$, respectively). We also have "program variables" and they point to nodes. When variable x points to node n, we represent it by $\rho(x) = n$. The special element $\mathrm{nil} \in N$ are introduced to represent that a pointer field or a program variable points to nowhere: it is represented by $p(n, f) = \mathrm{nil}$ or $\rho(v) = \mathrm{nil}$. By the introduction of nil, we can guarantee that p and ρ are total functions. The special element $\bot$ of PStr is used to represent that a program has "aborted".

4.2 Programming Language

In this section we introduce a programming language called PML [40] that has enough expressive power to describe programs that manipulates pointer structures defined in the previous section.

Figure 4.1 shows the syntax of PML.[1] We denote by Prog the set of PML programs by AProg the set of atomic programs by CProg the set of compound pro-

[1] In order to make the concrete and abstract transition system simple, PML employs minimum command sets. For example it is not possible to write `x := y.f1.f2;`. However, the same operation can be described using a temporary variable: `tmp := y.f1; x := tmp.f2;` .

⟨*program*⟩ ::= ⟨*atomic prog*⟩ | ⟨*compound prog*⟩

⟨*atomic prog*⟩ ::= `skip;` | `abort;` | ⟨*var*⟩`:=NULL;`

| ⟨*var*⟩`:=`⟨*var*⟩`;` | ⟨*var*⟩`:=`⟨*var*⟩`.`⟨*pfld*⟩`;`

| ⟨*var*⟩`.`⟨*bfld*⟩`:=`⟨*bval*⟩`;` | ⟨*var*⟩`.`⟨*pfld*⟩`:=`⟨*var*⟩`;`

| ⟨*var*⟩`:=new();`

⟨*compound prog*⟩ ::= ⟨*program*⟩⟨*program*⟩

| `if(`⟨*cond*⟩`){`⟨*program*⟩`}else{`⟨*program*⟩`}`

| `while(`⟨*cond*⟩`){`⟨*program*⟩`}`

⟨*cond*⟩ ::= ⟨*var*⟩`==NULL` | ⟨*var*⟩`==`⟨*var*⟩

| ⟨*var*⟩`.`⟨*bfld*⟩ | `!`⟨*cond*⟩ | ⟨*cond*⟩`||`⟨*cond*⟩

⟨*var*⟩ ::= (any element of Var)

⟨*bfld*⟩ ::= (any element of BFld)

⟨*pfld*⟩ ::= (any element of PFld)

⟨*bval*⟩ ::= `true` | `false`

Figure 4.1: Syntax of PML

grams, and by Cond the set of conditions. They are same as ⟨*program*⟩, ⟨*atomic prog*⟩, ⟨*compound prog*⟩, and ⟨*cond*⟩ respectively in Figure 4.1. We define the set Action of actions by Action = AProg ∪ Cond.

Intuitive meaning of a PML program is as follows: for a given initial state, the execution of a PML program is either normally terminates, abnormally terminates or never terminates. The simplest program that normally terminates is `skip;` and the simplest program that abnormally terminates is `abort;`. It also abnormally terminates when a null pointer is dereferenced. There are six types of assignment statements. The first three assignments simply changes the node that a variable points to, the last three changes the heap itself. Assignments x`.b:=true;` and x`.b:=false;` changes the Boolean value of a node, assignment x`.`f`:=`y`;` changes the node that a pointer of a node points to, and assignment x`:=new();` adds a node to the heap.

An example PML program called *reverse* is shown in Figure 4.2. Symbols `x`, `y`, and `t` are program variables and symbol `next` is a pointer field. Assume in the initial state, a list is given as `x` (i.e. the node that `x` points to reaches the nil by following `next` pointers). By executing the program reverse, the list is reversed and variable `y` points to the head of the reversed list at the final state.

We will define the semantics of a PML program P by defining a transition system. To prepare that, we define $[\![a]\!] \subseteq \mathrm{PtrStr} \times \mathrm{PtrStr}$ for $a \in \mathrm{Action}$. For $\mathcal{S}, \mathcal{S}' \in \mathrm{PtrStr}'$,

```
y := NULL;
while (!(x == NULL)) {
  t := y;
  y := x;
  x := x.next;
  y.next := t;
}
```

Figure 4.2: PML Program *reverse*

Table 4.1: Predicates cb And ct

c	$\mathrm{cb}(c, \mathcal{S})$	$\mathrm{ct}(c, \mathcal{S})$
x==NULL	false	$\rho(x) = \mathrm{nil}$
x_1==x_2	false	$\rho(x_1) = \rho(x_2)$
$x\,.\,g$	$\rho(x) = \mathrm{nil}$	$\rho(x) \in v(g)$
!c	$\mathrm{cb}(c, \mathcal{S})$	$\neg\mathrm{ct}(c, \mathcal{S})$
c_1 || c_2	$\mathrm{cb}(c_1, \mathcal{S}) \vee \mathrm{cb}(c_2, \mathcal{S})$	$\mathrm{ct}(c_1, \mathcal{S}) \vee \mathrm{ct}(c_2, \mathcal{S})$

$(\mathcal{S}, \mathcal{S}') \in [\![a]\!]$ intuitively means that action a can be executed in $\mathcal{S}$ and the result of the action is $\mathcal{S}'$. When a is a condition, a is regarded to be "executable" only if a is satisfied, and there is no change by the "execution".

Table 4.1 defines two predicates cb and ct for condition c and pointer structure $\mathcal{S} = (N, p, v, \rho, \mathrm{nil})$; x, x_1, x_2 are program variables, g a Boolean field, and c_1 and c_2 are conditions. The meaning of $\mathrm{cb}(c, \mathcal{S})$ is evaluation of c on $\mathcal{S}$ aborts, and $\mathrm{ct}(c, \mathcal{S})$ means that c is evaluated to true on $\mathcal{S}$. Using them, for $c \in \mathrm{Cond}$, we define $[\![c]\!]$ to be the least set that satisfies the following conditions:

- $(\bot, \bot) \in [\![c]\!]$
- $\mathrm{cb}(c, \mathcal{S}) \implies (\mathcal{S}, \bot) \in [\![c]\!]$
- $\neg\mathrm{cb}(c, \mathcal{S}) \wedge \mathrm{ct}(c, \mathcal{S}) \implies (\mathcal{S}, \mathcal{S}) \in [\![c]\!]$.

Table 4.2 defines two predicates pb and pt for atomic program P and pointer structures $\mathcal{S} = (N, p, v, \rho, \mathrm{nil})$ and $\mathcal{S}' = (N', p', v', \rho', \mathrm{nil}')$; x and y are program variables, f a pointer field, and g a Boolean field. Recall that for a function f, we denote by $f[a \mapsto b]$ a function g whose domain is $\mathrm{dom}(f) \cup \{a\}$, $g(a) = b$ and $g(x) = f(x)$ for $x \in \mathrm{dom}(f) \setminus \{a\}$. For seven assignments, components of $\mathcal{S}'$ that are not explicitly

Table 4.2: Predicates pb And pt

P	$\mathrm{pb}(P,\mathcal{S})$	$\mathrm{pt}(P,\mathcal{S},\mathcal{S}')$[2]
`skip;`	false	$\mathcal{S}=\mathcal{S}'$
`abort;`	true	false
x`:=NULL;`	false	$\rho'=\rho[x\mapsto \mathrm{nil}]$
x`:=`y`;`	false	$\rho'=\rho[x\mapsto \rho(y)]$
x`:=`y`.`f`;`	$\rho(y)=\mathrm{nil}$	$\rho'=\rho[x\mapsto p(\rho(y),f)]$
x`.`g`:=true;`	$\rho(x)=\mathrm{nil}$	$v'=v[g\mapsto v(g)\cup\{\rho(x)\}]$
x`.`g`:=false;`	$\rho(x)=\mathrm{nil}$	$v'=v[g\mapsto v(g)\setminus\{\rho(x)\}]$
x`.`f`:=`y`;`	$\rho(x)=\mathrm{nil}$	$p'=p[(\rho(x),f)\mapsto\rho(y)]$
x`:=new();`	false	$N'=N\uplus\{n\}$ $p'=p[(n,f)\mapsto n \mid f\in\mathrm{PFld}]$ $v'=v[(n,g)\mapsto$ `false` $\mid g\in\mathrm{BFld}]$ $\rho'=\rho[x\mapsto n]$

shown in the table should be the same as those in $\mathcal{S}$. For example for $P=$ "x:= `NULL;`", $N'=N$, $p'=p$, $v'=v$ and $\mathrm{nil}'=\mathrm{nil}$. $\mathrm{pb}(P,\mathcal{S})$ means that the execution of P aborts on $\mathcal{S}$ and $\mathrm{pt}(P,\mathcal{S},\mathcal{S}')$ means that the execution of P on $\mathcal{S}$ yields $\mathcal{S}'$.[3] Using them, for $P\in\mathrm{AProg}$, we define $[\![P]\!]$ to be the least set that satisfies the following conditions:

- $(\bot,\bot)\in[\![P]\!]$,
- $\mathrm{pb}(P,\mathcal{S})\implies(\mathcal{S},\bot)\in[\![P]\!]$,
- $\neg\mathrm{pb}(P,\mathcal{S})\wedge\mathrm{pt}(P,\mathcal{S},\mathcal{S}')\implies(\mathcal{S},\mathcal{S}')\in[\![P]\!]$.

Next we define the *control flow graph* $\mathrm{CFG}(P)$ of PML program P. It is a tuple (G,s,e,E,l) that consists of the following:

- A finite set G of *nodes*. It is also denoted by $|\mathrm{CFG}(P)|$.
- An element s of G. It is called the *start node*.
- An element e of G. It is called the *end node*.
- A finite set $E\subseteq G\times G$ of *edges*.
- A function $l:E\to\mathrm{Action}$. For $e\in E$, $l(e)$ is called the *label* of e.

[3] Definitions of pt for `new()` might not seem very natural. This definition is employed to make the calculation of the weakest preconditions simple. If a pointer field need to be initialized by nil, the following code can be used: `x:=new(); x.f:=NULL;`.

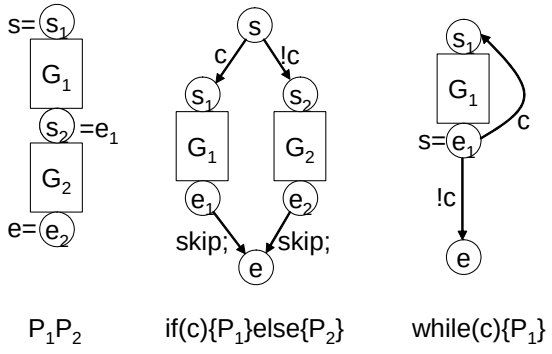


Figure 4.3: Control Flow Graph

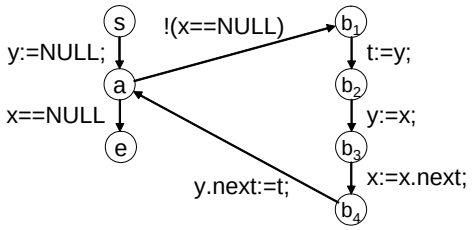


Figure 4.4: Control Flow Graph of Program reverse

(1) For atomic program P, G consists of two nodes s and e. E is a singleton set $\{(s, e)\}$ and $l(e) = P$.

(2) For each compound program, we define CFG(P) as in Figure 4.3. This figure assumes that $\mathcal{G}_i = \mathrm{CFG}(P_i)$ and shows the control flow graph for each compound program.

As an example the control flow graph for the program reverse is shown in Figure 4.4.

For a subset G' of G, we denote by path(G') the set of sequences $(g_0, \ldots, g_n)$ $(n \geq 1)$ of elements of G that satisfy $g_0, g_n \in G'$, $g_i \in G \setminus G'$ $(0 < i < n)$, and $(g_i, g_{i+1}) \in E$ $(0 \leq i < n)$.

Now we can define the transition relation associated to a PML program.

Definition 4.2 Let P be a PML program and $G \subseteq |\mathrm{CFG}(P)|$. We define transition relation $\mathcal{T}(G) = (\mathrm{PtrStr} \times G, \rightarrow_G)$ where to_G is the least relation on PtrStr $\times\, G$ that satisfies the following conditions:

- $\mathcal{S}, \mathcal{S}' \in \mathrm{PtrStr}$,

- $(g_0, \ldots, g_n) \in \mathrm{path}(G)$,
- $(\mathcal{S}, \mathcal{S}') \in [\![l(g_0, g_1)]\!] \circ \cdots \circ [\![l(g_{n-1}, g_n)]\!]$ implies $(S, g_0) \to_G (S', g_n)$, where $\circ$ is the function composition.

We call $\mathcal{T}(|\mathrm{CFG}(P)|)$ the transition relation associated to P. ∎

4.3 Specifications

In this section we introduce a language to express specifications of PML programs. The language need to express (1) the heap status at a time, and (2) how the heap status changes during while the program executes.

4.3.1 Heap Status

Since a pointer structure is very alike to a functional Kripke structure, our basic idea is to use the modal logic $\mathrm{L}_0(\mu_{\mathrm{AF}}, @, \bar{\ })$ or its appropriate sublogic to describe properties of pointer structures.

Recall that the language L_0 has hidden parameters PS, Nom, and MS. In this section we set PS = BFld, Nom = Var ∪ {NULL}, and MS = PFld.

Definition 4.3 Let $\mathcal{S} = (N, p, v, \rho, \mathrm{nil})$ be a pointer structure. The Kripke Structure *associated to* $\mathcal{S}$ is a tuple $\mathcal{K} = (N, R, \lambda)$ defined as follows: for $f \in \mathrm{Fld}$, $d \in \mathrm{Val}$ and $x \in \mathrm{Var}$,

- $R(f) = \{(n, p(n, f)) \mid n \in N\}$
- $\lambda(g) = v(g)$
- $\lambda(\mathrm{NULL}) = \{\mathrm{nil}\}$

For $s \in N$ and $\varphi \in \mathrm{L}(\mu_{\mathrm{AF}}, @, \bar{\ })$, we write $\mathcal{S}, s \models \varphi$ for $\mathcal{K}, s \models \varphi$. ∎

A formula $\varphi \in \mathrm{L}_0(\mu_{\mathrm{AF}}, @, \bar{\ })$ describes the status of a pointer structure. For example formula $\varphi_1 = @x\, \mu X(y \vee \langle \mathrm{next} \rangle X)$, where x and y are program variables, means that one can reach to a node pointed to by variable y by starting a node pointed to by variable x and follows the pointer next. As another example, $\varphi_2 = @x\, \langle \mathrm{next} \rangle \langle \overline{\mathrm{next}} \rangle y$ means that the node pointed to by variable x and the node pointed to by variable y point to a same node.

However, not all formula in $\mathrm{L}_0(\mu_{\mathrm{AF}}, @, \bar{\ })$ can be used to describe the status. For example consider formula $\varphi_3 = x \wedge y$. For pointer structure $\mathcal{S}$ and $s \in N_{\mathcal{S}}$, whether

$\mathcal{S}, s \models \varphi_3$ or not depends not only on $\mathcal{S}$ but also on s. In other words, formula φ_3 describes the status of a node in a pointer structure rather than the structure. On the other hand, whether formula $\mathcal{S}, s \models \varphi_1$ or not depends only on $\mathcal{S}$, not on s. The same applies for φ_2. Therefore we can use such formula as φ_1 or φ_2 to describe the status of a pointer structure.

Definition 4.4 We call either of the following a *p-formula*:

- a formula in $\mathrm{L}_0(\mu_{\mathrm{AF}}, @, \bar{\ })$ in the form of $@x\varphi$, where x is a nominal and "at" operator ($@$) does not occur in φ.[4]
- a special atomic formula abort,
- their Boolean combination.

We denote by PFormthe set of p-formulae. ∎

The atomic formula abort is used to express that the program has aborted.
For $\mathcal{S} \in$ PtrStr and p-formula φ, we define relation $\mathcal{S} \models \varphi$ as follows:

- $\mathcal{S} \models @x\,\varphi \iff \mathcal{S} \neq \bot$ and there is $s \in N_{\mathcal{S}}$ such that $\mathcal{S}, s \models @x\,\varphi$ (hence for all $s \in N_{\mathcal{S}}$, $\mathcal{S}, s \models @x\,\varphi$).
- $\mathcal{S} \models \mathrm{abort} \iff S = \bot$.

We naturally extend the definition to the Boolean combinations.

4.3.2 Specification Formulae

Specifications of PML program P can be described with temporal formulae that are to be satisfied with the transition system $\mathcal{T}(G)$ for appropriate $G \subseteq |\mathrm{CFG}(P)|$. We will define a specification description language based on a well-known temporal logic LTL (Linear Temporal Logic).

Let $Q \in$ PForm and $g \in G$. We call the formulae defined as σ in the following BNF *specification formulae*.

$$\sigma ::= Q \mid g \mid \neg\sigma \mid \sigma \vee \sigma \mid \sigma\,\mathcal{U}\,\sigma$$

where $\mathcal{U}$ is the standard until operator of LTL. Standard abbreviations such as $\Diamond\sigma \stackrel{\mathrm{def}}{=} \mathrm{true}\,\mathcal{U}\,\sigma$ and $\Box\sigma \stackrel{\mathrm{def}}{=} \neg\Diamond\neg\sigma$ will be used.

Interpretations of atomic propositions in $\mathcal{T}(G)$ is defined as follows:

[4]The limitation that "at" operator should not occur in φ is not essential. Even if we lift the limitation, we can find an equivalent formula within the limitation.

Table 4.3: Specification Formula Examples

spec	formula
(1)	$\sigma_1 = \Box(s \wedge @u\, \neg\mathrm{NULL} \wedge @x\, \mu X(u \vee \langle \mathtt{next} \rangle X)$
	$\quad \rightarrow \Box(e \rightarrow @y\, \mu X(u \vee \langle \mathtt{next} \rangle X)))$
(2)	$\sigma_2 = \Box(s \wedge @u_2\, \neg\mathrm{NULL} \wedge @x\, \mu X(\mathrm{NULL} \vee \langle \mathtt{next} \rangle X)$
	$\quad \wedge @x\, \mu X(u_1 \vee \langle \mathtt{next} \rangle X) \wedge @u_1\, \langle \mathtt{next} \rangle u_2)$
	$\quad \rightarrow \Box(e \rightarrow @u_2\, \langle \mathtt{next} \rangle u_1))$
(3)	$\sigma_3 = \Box(s \wedge \neg\mathrm{abort} \rightarrow \Box\neg\mathrm{abort})$
(4)	$\sigma_4 = \Diamond e$

- $(\mathcal{S}, g) \models Q \iff \mathcal{S} \models Q$,
- $(\mathcal{S}, g) \models g' \iff g = g'$

where $\mathcal{S} \in \mathrm{PtrStr}$, $Q \in \mathrm{PForm}$ and $g, g' \in G$.

Example 4.5 The following specifications are supposed to be satisfied by the program reverse.

(1) A node that can be reached from variable x before the executing of the program can be reached from variable y after the execution.

(2) If, before the execution of the program, program variable x points to a list and the next field of a node u_1 that is reachable from x points to u_2, then, after the execution, the next field of u_2 points to u_1.

(3) The program does not abort.

(4) The program terminates after finitely many execution of the while loop.

Table 4.3 shows formulae that represent these specifications. We take the set G to the one that consists of s, e, and a in Figure 4.4. Symbols u, u_1 and u_2 are nominals that do not appear in the program reverse.

4.4 Weakest Preconditions

In this section, for each atomic program P in PML and formula φ in $L_0(\mu, @, \bar{})$, we define a formula called the *weakest precondition* of φ with regard to P.

We work with language $L_0(\mu, @, \overline{\ })$ unless explicitly stated otherwise.

We define two auxiliary formulae, $\mathrm{wpl}(P, \varphi)$ and $\mathrm{nn}(x, \varphi)$ for atomic program P, formula φ and program variable x. As we will see in Lemma 4.6, wpl is the "weakest precondition calculated locally", meaning that a node in a pointer structure satisfies $\mathrm{wpl}(P, \varphi)$ if and only if the same node in the resulting pointer structure by the execution of P satisfies φ. And $\mathrm{nn}(x, \varphi)$ is the formula expressing whether φ will be satisfied by the "new node" added by x:= `new();`.

First we give the definition of wpl:

- $\mathrm{wpl}(x_1\texttt{:=}x_2\texttt{;}, \varphi) = \varphi[x_2/x_1]$. This is the formula obtained from φ by replacing occurrences of x_1 with x_2.
- $\mathrm{wpl}(x_1\texttt{:=NULL;}, \varphi) = \varphi[\mathrm{NULL}/x_1]$.
- $\mathrm{wpl}(x_1.g\texttt{:=}b\texttt{;}, \varphi)$ is defined as follows:
  - $\mathrm{wpl}(x_1.g\texttt{:=true;}, \varphi) = \varphi[g \vee x_1/g]$. This is the formula obtained from φ by replacing occurrences of g with $g \vee x_1$. Note that since g is a predicate symbol and x_1 is a nominal, the substitution yields a well-formed formula.
  - $\mathrm{wpl}(x_1.g\texttt{:=false;}, \varphi) = \varphi[g \wedge \neg x_1/g]$.
- $\mathrm{wpl}(x_1\texttt{:=}x_2.f\texttt{;}, \varphi)$. Roughly speaking, this is obtained from φ by replacing x_1 with $\langle\overline{f}\rangle x_2$. However, if x_1 occurs in φ in the form of $@x_1\ \psi$, the substitution would not yield a well-formed formula. In that case we replace $@x_1\psi$ with $@y\langle f\rangle\psi$. Precise definition by induction on the construction of φ is as follows. We write P for $x_1\texttt{:=}x_2.f\texttt{;}$.
  - $\mathrm{wpl}(P, \xi) = \xi \quad$ for $\xi \in (\mathrm{PS} \cup \mathrm{Nom} \cup \mathrm{PV}) \setminus \{x_1\}$.
  - $\mathrm{wpl}(P, x_1) = \langle\overline{f}\rangle x_2$.
  - $\mathrm{wpl}(P, \neg\varphi) = \neg\mathrm{wpl}(P, \varphi)$.
  - $\mathrm{wpl}(P, \varphi_1 \vee \varphi_2) = \mathrm{wpl}(P, \varphi_1) \vee \mathrm{wpl}(P, \varphi_2)$.
  - $\mathrm{wpl}(P, \langle m\rangle\varphi) = \langle m\rangle\mathrm{wpl}(P, \varphi) \quad$ for $m \in \mathrm{Mod}$.
  - $\mathrm{wpl}(P, \mu X\varphi) = \mu X\mathrm{wpl}(P, \varphi)$.
  - $\mathrm{wpl}(P, @x\ \varphi) = @x\ \mathrm{wpl}(P, \varphi) \quad$ for $x \in \mathrm{Nom} \setminus \{x_1\}$.
  - $\mathrm{wpl}(P, @x_1\ \varphi) = @x_2\ \langle f\rangle\mathrm{wpl}(P, \varphi)$.
- $\mathrm{wpl}(x_1.f\texttt{:=}x_2\texttt{;}, \varphi)$. This is again defined by induction on the construction of φ. We write P for $x_1.f\texttt{:=}x_2\texttt{;}$.

- $\mathrm{wpl}(P, \xi) = \xi \quad$ for $\xi \in \mathrm{PS} \cup \mathrm{Nom} \cup \mathrm{PV}$.
- $\mathrm{wpl}(P, \neg\varphi) = \neg\mathrm{wpl}(P, \varphi)$.
- $\mathrm{wpl}(P, \varphi_1 \vee \varphi_2) = \mathrm{wpl}(P, \varphi_1) \vee \mathrm{wpl}(P, \varphi_2)$.
- $\mathrm{wpl}(P, \langle m\rangle\varphi) = \langle m\rangle\mathrm{wpl}(P, \varphi) \quad$ for $m \in \mathrm{Mod} \setminus \{f, \overline{f}\}$.
- $\mathrm{wpl}(P, \langle f\rangle\varphi) = v_1 \rightarrow @v_2\ \mathrm{wpl}(P, \varphi)\ ; \langle f\rangle\mathrm{wpl}(P, \varphi)$.
- $\mathrm{wpl}(P, \langle \overline{f}\rangle\varphi) = (v_2 \wedge @v_1\ \mathrm{wpl}(P, \varphi)) \wedge \langle \overline{f}\rangle(\neg v_1 \wedge \mathrm{wpl}(P, \varphi))$.
- $\mathrm{wpl}(P, \mu X\varphi) = \mu X\mathrm{wpl}(P, \varphi)$.
- $\mathrm{wpl}(P, @x\ \varphi) = @x\ \mathrm{wpl}(P, \varphi) \quad$ for $x \in \mathrm{Nom}$.

- $\mathrm{wpl}(x\texttt{:=new();}, \varphi)$. This is again defined by induction on the construction of φ. We also need formula nn here, so the whole definition should be regarded as mutual induction for wpl and nn. We write P for $x\texttt{:=new();}$.
  - $\mathrm{wpl}(P, \xi) = \xi \quad$ for $\xi \in (\mathrm{PS} \cup \mathrm{Nom} \cup \mathrm{PV}) \setminus \{x\}$.
  - $\mathrm{wpl}(P, x) = \mathbf{false}$.
  - $\mathrm{wpl}(P, \neg\varphi) = \neg\mathrm{wpl}(P, \varphi)$.
  - $\mathrm{wpl}(P, \varphi_1 \vee \varphi_2) = \mathrm{wpl}(P, \varphi_1) \vee \mathrm{wpl}(P, \varphi_2)$.
  - $\mathrm{wpl}(P, \langle m\rangle\varphi) = \langle m\rangle\mathrm{wpl}(P, \varphi) \quad$ for $m \in \mathrm{Mod}$.
  - $\mathrm{wpl}(P, \mu X\varphi) = \mu X\mathrm{wpl}(P, \varphi)$.
  - $\mathrm{wpl}(P, @y\ \varphi) = @y\ \mathrm{wpl}(P, \varphi) \quad$ for $y \in \mathrm{Nom} \setminus \{x\}$
  - $\mathrm{wpl}(P, @x\ \varphi) = @x\ \mathrm{nn}(x, \varphi)$.

Next we define nn:

- $\mathrm{nn}(x, \varphi)$ is also defined by induction on the construction of φ.
  - $\mathrm{nn}(x, g) = \mathbf{false} \quad$ for $g \in \mathrm{PS}$.
  - $\mathrm{nn}(x, y) = \mathbf{false} \quad$ for $y \in \mathrm{Nom} \setminus \{x\}$.
  - $\mathrm{nn}(x, x) = \mathbf{true}$
  - $\mathrm{nn}(x, X) = X \quad$ for $X \in \mathrm{PV}$.
  - $\mathrm{nn}(x, \neg\varphi) = \neg\mathrm{nn}(x, \varphi)$.
  - $\mathrm{nn}(x, \varphi_1 \vee \varphi_2) = \mathrm{nn}(x, \varphi_1) \vee \mathrm{nn}(x, \varphi_2)$.
  - $\mathrm{nn}(x, \mu X\varphi) = \mu X\mathrm{nn}(P, \varphi)$.
  - $\mathrm{nn}(x, @y\ \varphi) = @y\ \mathrm{wpl}(x\texttt{:=new();}, \varphi) \quad$ for $y \in \mathrm{Nom} \setminus \{x\}$.

– $\mathrm{nn}(x, @x\,\varphi) = \mathrm{nn}(x, \varphi)$.

The following two lemmas formally express the intuition described before the definitions of wpl and nn.

Lemma 4.6 Let P be an atomic program of PML other than x`:=new();`, $\mathcal{S}_1$ and $\mathcal{S}_2$ be pointer structures such that $(\mathcal{S}_1, \mathcal{S}_2) \in [\![P]\!]$. (Thus $N_{\mathcal{S}_1} = N_{\mathcal{S}_2}$ holds and we denote this set by S.) Let v be a valuation for $\mathcal{S}_1$ and $\mathcal{S}_2$, and φ be a formula. Then for any $s \in S$,

$$\mathcal{S}_1, v, s \models \mathrm{wpl}(P, \varphi) \iff \mathcal{S}_2, v, s \models \varphi$$

holds.

Proof The proof is induction on the construction of φ. Most of the cases are trivial and here we only illustrates the proof by picking up a few cases. Omitted cases can be proved similarly and without difficulty. Let $\mathcal{S}_1 = (S, p_1, v_1, \rho_1, \mathrm{nil})$ and $\mathcal{S}_2 = (S, p_2, v_2, \rho_2, \mathrm{nil})$.

Case $P = x_1$`:=`x_2`;`, $\varphi = x_1$. What needs to be proved is: $\mathcal{S}_1, s \models x_1 \iff \mathcal{S}_2, s \models x_2$. The lhs is $s = \rho_1(x)$ and the rhs is $s = \rho_2(x)$. But $\rho_2(x) = \rho_1(x)$ by the definition of $[\![P]\!]$ and we are done.

Case $P = x_1$`:=`$x_2.f$`;`, $\varphi = @x_1\,\psi$. We have $\mathcal{S}_2, s \models @x_1\,\psi \iff \mathcal{S}_2, \rho_2(x_1) \models \psi \iff \mathcal{S}_1, p_1(f, \rho_1(x_2)) \models \mathrm{wpl}(P, \psi) \overset{(*)}{\iff} \mathcal{S}_1, \rho_1(x_2) \models \langle f \rangle \mathrm{wpl}(P, \psi) \iff \mathcal{S}_1, s \models @x_2 \langle f \rangle \mathrm{wpl}(P, \psi) \iff \mathcal{S}_1, s \models \mathrm{wpl}(P, @x_1\,\psi)$. For the direction from right to left marked by (*), note that a Kripke structure that corresponds to a pointer structure is functional.

Case $P = x_1.f$`:=`x_2`;`, $\varphi = \langle f \rangle \psi$. Note that $\rho_1(x_1) = \rho_2(x_1)$ and $\rho_1(x_2) = \rho_2(x_2)$. Let us write s_1 for $\rho_1(x_1)$ and s_2 for $\rho_1(x_2)$. If $s = s_1$, we have $\mathcal{S}_2, s \models \langle f \rangle \psi \iff \mathcal{S}_2, s_2 \models \psi \iff \mathcal{S}_1, s_2 \models \mathrm{wpl}(P, \psi) \iff \mathcal{S}_1, s \models @x_2\,\mathrm{wpl}(P, \psi) \iff \mathcal{S}_1, s \models x_1 \to @x_2\,\mathrm{wpl}(P, \psi)\,;\langle f \rangle \mathrm{wpl}(P, \psi)$. On the other hand if $s \neq s_1$, since $p_1(f, s) = p_2(f, s)$, we have $\mathcal{S}_2, s \models \langle f \rangle \psi \iff \mathcal{S}_2, p_2(f, s) \models \psi \iff \mathcal{S}_1, p_1(f, s) \models \mathrm{wpl}(P, \psi) \iff \mathcal{S}_1, s \models \langle f \rangle \mathrm{wpl}(P, \psi) \iff \mathcal{S}_1, s \models x_1 \to @x_2\,\mathrm{wpl}(P, \psi)\,;\langle f \rangle \mathrm{wpl}(P, \psi)$. Thus in either case we have $\mathcal{S}_2, s \models \varphi \iff \mathcal{S}_1, s \models \mathrm{wpl}(P, \varphi)$ and we are done. ∎

Lemma 4.7 Let P be atomic program x`:=new();`, $\mathcal{S}_1$ and $\mathcal{S}_2$ be pointer structures such that $(\mathcal{S}_1, \mathcal{S}_2) \in [\![P]\!]$, $S_1 = N_{\mathcal{S}_1}$, $S_2 = N_{\mathcal{S}_2}$, $\hat{s}$ be the unique element of $S_2 \setminus S_1$, v_1 and v_2 be valuations for $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively, such that $v_1(X) = v_2(X) \cap S_1$ for any $X \in \mathrm{PV}$. Let φ be an AV-free formula. Then the followings hold:

- Formula $\mathrm{wpl}(P, \varphi)$ is AV-free and for any $s \in S_1$,

$$\mathcal{S}_2, v_2, s \models \varphi \iff \mathcal{S}_1, v_1, s \models \mathrm{wpl}(P, \varphi).$$

- Formula $\mathrm{nn}(x, \varphi)$ is AV-free and

$$\begin{aligned} \mathcal{S}_2, v_2, \hat{s} \models \varphi &\iff \forall s \in S_1 \quad \mathcal{S}_1, v_1', s \models \mathrm{nn}(x, \varphi) \\ &\iff \exists s \in S_1 \quad \mathcal{S}_1, v_1', s \models \mathrm{nn}(x, \varphi) \end{aligned}$$

  where valuation v_1' for $\mathcal{S}_1$ is defined as follows:

$$v_1'(X) = \begin{cases} S_1 & \text{if } \hat{s} \in v_2(X) \\ \varnothing & \text{otherwise} \end{cases}$$

Proof The proof is given by induction on the construction of φ. That $\mathrm{wpl}(P, \varphi)$ and $\mathrm{nn}(x, \varphi)$ are AV-free can be easily checked by noticing that all subformulae of an AV-free formula are AV-free and the set of free variables in φ and that of $\mathrm{wpl}(P, \varphi)$ or $\mathrm{nn}(P, \varphi)$ are identical.

We first check wpl. Checks for $\psi \in \mathrm{PS} \cup \mathrm{Nom} \cup \mathrm{PV}$, $\neg\psi$, $\psi_1 \vee \psi_2$, $@y\ \psi$ $(y \neq x)$ are trivial. The case of $\langle m \rangle \psi$ is also clear by noticing that no transition is added to existing nodes in both direction.

Case $\varphi = \mu X \psi$. We need to show $[\![\mathrm{wpl}(P, \mu X \psi)]\!]^{\mathcal{S}_1, v_1} = [\![\mu X \psi]\!]^{\mathcal{S}_2, v_2} \cap S_1$. The lhs is $[\![\mu X \mathrm{wpl}(P, \psi)]\!]^{\mathcal{S}_1, v_1}$. For natural number i, we define $T_i^1 \subseteq S_1$ and $T_i^2 \subseteq S_2$ by $T_0^1 = \varnothing$, $T_{i+1}^1 = [\![\mathrm{wpl}(P, \psi)]\!]^{\mathcal{S}_1, v_1[X \mapsto T_i^1]}$ and $T_0^2 = \varnothing$, $T_{i+1}^2 = [\![\psi]\!]^{\mathcal{S}_2, v_2[X \mapsto T_i^2]}$. We show by induction on i that $T_i^1 = T_i^2 \cap S_1$. The case $i = 0$ is clear. The case $i+1$: since from the hypothesis of the induction on i, $T_i^1 = T_i^2 \cap S_1$, we have $(v_1[X \mapsto T_i^1])(Y) = v_2(Y) \cap S_1$ for any $Y \in \mathrm{PV}$. Therefore the hypothesis of the induction on φ can be applied and we have $[\![\mathrm{wpl}(P, \psi)]\!]^{\mathcal{S}_1, v_1[X \mapsto T_i^1]} = [\![\psi]\!]^{\mathcal{S}_2, v_2[X \mapsto T_i^2]} \cap S_1$, which means $T_{i+1}^1 = T_{i+1}^2 \cap S_1$ as desired.

Since the underlying set of a pointer structure is finite, we have
$[\![\mu X \mathrm{wpl}(P, \psi)]\!]^{\mathcal{S}_1, v_1} = \bigcup_{i<\omega} T_i^1 = \bigcup_{i<\omega} T_i^2 = [\![\mu X \psi]\!]^{\mathcal{S}_2, v_2} \cap S_1$.

Case $\varphi = @x\ \psi$. Since φ is a subformula of a AV-free formula, it is AV-free, hence there is no free variables in ψ. Therefore we can ignore the valuations. We start with $\mathcal{S}_2, s \models @x\ \psi$. It is equivalent to $\mathcal{S}_2, \hat{s} \models \psi$. By induction hypothesis, $\mathcal{S}_2, \hat{s} \models \psi \implies \mathcal{S}_1, s \models \mathrm{nn}(x, \varphi) \implies \mathcal{S}_2, \hat{s} \models \psi$, so they are equivalent. Finally by definition $\mathcal{S}_1, s \models \mathrm{nn}(x, \varphi) \iff \mathcal{S}_1, s \models \mathrm{wpl}(P, @x\ \psi)$.

Next we check nn. Again almost all cases are straight-forward, except for $\mu X \psi$, $@y\ \psi$ $y \neq x$, and $@x\ \psi$. But the last two do not contain particular difficulty if one

notices that $@y\,\psi$ and $@x\,\psi$ are AV-free and ψ does not contain free variables, just as the case in wpl. So the only remaining case is the following:

Case $\varphi = \mu X\psi$. We need to show $[\![\mu X \mathrm{nn}(x,\psi)]\!]^{S_1,v_1'} = S_1$ if $\hat{s} \in [\![\mu X\psi]\!]^{\mathcal{S}_2,v_2}$ and $[\![\mu X \mathrm{nn}(x,\psi)]\!]^{S_1,v_1'} = \varnothing$ otherwise. For natural number i, we define $T_i^1 \subseteq S_1$ and $T_i^2 \subseteq S_2$ by $T_0^1 = \varnothing$, $T_{i+1}^1 = [\![\mathrm{nn}(x,\psi)]\!]^{\mathcal{S}_1,v_1'[X\mapsto T_i^1]}$ and $T_0^2 = \varnothing$, $T_{i+1}^2 = [\![\psi]\!]^{\mathcal{S}_2,v_2[X\mapsto T_i^2]}$. We show by induction on i that $T_i^1 = S_1$ if $\hat{s} \in T_i^2$ and $T_i^1 = \varnothing$ otherwise. The case $i = 0$ is clear. The case $i+1$: since from the hypothesis of the induction on i, $T_i^1 = S_1$ if $\hat{s} \in T_i^2$ and $T_i^1 = \varnothing$ otherwise, we have $(v_1[X \mapsto T_i^1])(Y) = S_1$ if $\hat{s} \in v_2[X \mapsto T_i^2]$ and $(v_1[X \mapsto T_i^1])(Y) = \varnothing$ otherwise. $v_2(Y) \cap S_1$ for any $Y \in \mathrm{PV}$. Therefore the hypothesis of the induction on φ can be applied and we have $[\![\mathrm{nn}(P,\psi)]\!]^{\mathcal{S}_1,v_1[X\mapsto T_i^1]} = S_1$ if $\hat{s} \in [\![\psi]\!]^{\mathcal{S}_2,v_2[X\mapsto T_i^2]}$ and $[\![\mathrm{nn}(P,\psi)]\!]^{\mathcal{S}_1,v_1[X\mapsto T_i^1]} = \varnothing$ otherwise. By the definition of T_{i+1}^1 and T_{i+1}^2, the induction on i completes.

f If $\hat{s} \in [\![\mu X\psi]\!]^{\mathcal{S}_2,v_2} = \bigcup_{i<\omega} T_i^2$, pick i such that $\hat{s} \in T_i^2$ then $[\![\mu X \mathrm{nn}(x,\psi)]\!]^{\mathcal{S}_1,v_1'} \supseteq T_i^1 = S_1$. Otherwise $T_i^1 = \varnothing$ for all $i < \omega$ and thus $[\![\mu X \mathrm{nn}(x,\psi)]\!]^{\mathcal{S}_1,v_1'} = \varnothing$. ∎

Definition 4.8 For atomic program P of PML and closed formula φ, closed formula $\mathrm{wp}(P,\varphi)$ is defined as follows:

$$\mathrm{wp}(P,\varphi) = \begin{cases} \mathrm{wpl}(P,\hat{\varphi}) \wedge \mathrm{nn}(x,\hat{\varphi}) & \text{if } P \text{ is in the form of } x\texttt{:=new();} \\ \mathrm{wp}(P,\varphi) & \text{otherwise} \end{cases}$$

where $\hat{\varphi}$ is some fixed FA-free formula such that $\varphi \equiv \hat{\varphi}$. ∎

Note that the existence of $\hat{\varphi}$ is guaranteed by Lemma 2.21

Theorem 4.9 Assume P is an atomic program of PML, φ is a closed formula, $\mathcal{S}_1$ and $\mathcal{S}_2$ are pointer structures such that $(\mathcal{S}_1,\mathcal{S}_2) \in [\![P]\!]$. Then,

$$\mathcal{S}_1 \models \mathrm{wp}(P,\varphi) \iff \mathcal{S}_2 \models \varphi$$

holds.

Proof When P is not in the form of $x\texttt{:=new();}$, the conclusion clearly follows from Lemma 4.6.

Assume P is $x\texttt{:=new();}$ and take $\hat{\varphi}$ in Definition 4.8. Since it is closed and FA-free, it is AV-free by Lemma 2.20. Therefore we can apply Lemma 4.7. In the rest of the proof, we use the same notation as in the lemma. Note that both φ and $\mathrm{wp}(P,\varphi)$ are

closed and we can ignore the valuations in the lemma.

$$
\begin{array}{lll}
 & \mathcal{S}_1 \models \mathrm{wp}(P, \varphi) & \\
\iff & \forall s \in S_1 \quad \mathcal{S}_1, s \models \mathrm{wpl}(P, \hat{\varphi}) \wedge \mathrm{nn}(x, \hat{\varphi}) & \\
\iff & \forall s \in S_1 \quad \mathcal{S}_2, s \models \hat{\varphi} \quad \text{and} \quad \mathcal{S}_2, \hat{s} \models \hat{\varphi} & \\
\iff & \forall s \in S_2 \quad \mathcal{S}_2, s \models \hat{\varphi} & \\
\iff & \mathcal{S}_2 \models \varphi & \blacksquare
\end{array}
$$

This result is an extension of our previous results for CTL [68].

4.5 Abstract Structure

4.5.1 Predicate Abstraction

The relation "PML program P satisfies specification formula σ" is defined with transition system $\mathcal{T}(G)$, which is infinite since there are infinitely many pointer structures. Therefore it is impossible to apply the ordinary search method of model checking. We will build a *finite* transition system that abstracts $\mathcal{T}(G)$ using the predicate abstraction technique [55].

In the predicate abstraction technique, for given (usually infinite or very large finite) transition system, a finite set of predicates on the transition system is fixed. The underlying set A of the abstract transition system $\mathcal{A} = (A, \rightarrow_{\mathcal{S}})$ consists of the combinations of truth values (true or false) for these predicates. Abstract transition is usually defined using the weakest precondition wp of the predicates and decision procedure sat for satisfiability: $a_1 \rightarrow_{\mathcal{S}} a_2 \iff \mathrm{sat}(a_1 \wedge \mathrm{wp}(a_2)) =$"yes". Here we overuse notation a little. More accurately, recalling that $a \in A$ is the combination of the truth values of predicates, we can calculate $\mathrm{wp}(P)$ for predicates P which **true** is given for in a_2 and $\mathrm{wp}(\neg P)$ for predicates P which **false** is given for in a_2 and make conjunctions all of them plus predicates P which **true**: is given for in a_1 and $\neg P$ for predicates P which **false** is given for in a_2. Then we pass the conjunction to the decision procedure, if it says the conjunction is satisfiable then we conclude that in the abstract transition system there is a transition from a_1 to a_2; otherwise there is not. Refer to [69] for details.

We apply the technique to the concrete transition system $\mathcal{T}(G)$ and we use p-formulae for predicates. In order for the abstraction to be sound, the *weakest* preconditions and *complete* decision procedure for satisfiability are not necessarily required as we will see below.

Definition 4.10 A function pre : Action $\times$ PForm $\to$ PForm is called a *precondition function*[5] if for each $\mathcal{S}_1, \mathcal{S}_2 \in$ PtrStr, $a \in$ Action and $Q \in$ PForm, $(\mathcal{S}_1, \mathcal{S}_2) \in [\![a]\!]$ and $\mathcal{S}_2 \models Q$ implies $\mathcal{S}_1 \models \mathrm{pre}(a, Q)$. ∎

A trivial example of precondition function is $\mathrm{pre}(a, Q) = \mathrm{true}$. This would lead to a sound abstraction as we will see, but the resulting abstract system is meaningless for verification purposes. More useful precondition function will be described in Section 4.7.1.

Let $\vec{a} = (a_1, \ldots, a_n)$ be a finite sequence of actions. The precondition of the sequence $\vec{a}$ is defined as $\mathrm{pre}(a_1, \ldots, \mathrm{pre}(a_n, Q)\ldots)$ and denoted by $\mathrm{pre}(\vec{a}, Q)$ using the same symbol.

Decision procedures of satisfiability needs to be satisfied the condition in the following definition for a sound predicate abstraction.

Definition 4.11 A function sat : PForm $\to \{\mathbf{true}, \mathbf{false}\}$ *judges the satisfiability* if $\mathrm{sat}(Q) = \mathbf{true}$ whenever there is a pointer structure $\mathcal{S}$ such that $\mathcal{S} \models Q$. ∎

We have a trivial example. The constant function $\mathrm{sat}(Q) = \mathbf{true}$ judges the satisfiability. But the resulting abstract structure is totally meaningless since any pair of nodes are connected by the transition relation. More useful function will be described in Section 4.7.1.

For a finite sequence $\pi = g_0, \ldots, g_n \in |\mathrm{CFG}(P)|$ of nodes of a control flow graph, we denote by $\mathrm{act}(\pi)$ the sequence $l(g_0, g_1), \ldots, l(g_{n-1}, g_n)$ of actions.

Definition 4.12 Let pre be a precondition function, sat a function that judges the satisfiability, $Q_1, \ldots, Q_n$ be p-formulae, P a PML program, G a subset of $|\mathrm{CFG}(P)|$ such that $\mathrm{path}(G)$ is finite. The abstract transition system $\mathcal{T}^{\mathrm{A}} = (A, \to_A)$ generated by pre, sat, $Q_1, \ldots, Q_n$, P and G is as follows: the underlying set A is defined as $A = B \times G$, where $B = B' \cup \{\bot\}$ and $B' = \{b \mid b : \{1, \ldots, n\} \to \{\mathbf{true}, \mathbf{false}\}\}$. The transition relation $\to_A$ is defined so that $(b_1, g_1) \to_A (b_2, g_2)$ if and only if there exists $\pi \in \mathrm{path}(G)$ $\mathrm{sat}(Q(b_1) \wedge \mathrm{pre}(\mathrm{act}(\pi), Q(b_2))) = \mathrm{true}$, where $Q : B \to$ PForm is defined so that $Q(b) = \bigwedge_{b(i)=\mathbf{false}} \neg Q_i \wedge \bigwedge_{b(i)=\mathbf{true}} Q_i$ for $b \in B$ and $Q(\bot) = \mathrm{abort}$. ∎

This abstraction is sound for specification formulae, that is the following theorem holds.

[5] Again, this definition may seem a little different from the natural meaning of "precondition". More natural definition would be "$(\mathcal{S}_1, \mathcal{S}_2) \in [\![a]\!]$ and $\mathcal{S}_1 \models \mathrm{pre}(a, Q)$ implies $\mathcal{S}_2 \models Q$", which can be written as $\neg\mathrm{pre}(a, \neg Q)$ in our notation. But that would be useless for sound abstraction.

Theorem 4.13 Let σ be a specification formula. If $\mathcal{T}^{\mathrm{A}} \models \sigma$, $\mathcal{T}(G) \models \sigma$ holds.

Proof Induction on the construction of the specification formula. ∎

Since $\mathcal{T}^{\mathrm{A}}$ is a finite transition system, we can model-check whether $\mathcal{T}^{\mathrm{A}} \models \sigma$ holds or not. If it holds, the program is guaranteed to satisfy the specification by the theorem.

4.6 Steps of Analysis

When a PML program P is given, we can follow the following steps to verify specifications for the program.

(1) Choose $G \subseteq |\mathrm{CFG}(P)|$ so that $\mathrm{path}(G)$ is finite. For example, it is achieved by including all head points of the while loops in the program. "Interesting" nodes of the control flow graph, i.e. nodes at which some properties need to be checked, should also be included in G.

(2) Describe a specification formula σ so that $\mathcal{T}(G) \models \sigma$ expresses the specification to be verified.

(3) Choose finite p-formulae $Q_1, \ldots, Q_n$ as abstraction predicates. All p-formulae appeared in the specification formula as atomic formulae should be included in them.

(4) Construct the abstract structure $\mathcal{T}^{\mathrm{A}}$ from P, G, $Q_1, \ldots, Q_n$ and model-check the structure to judge whether $\mathcal{T}^{\mathrm{A}} \models \sigma$ holds or not. If it holds, $\mathcal{T}(G) \models \sigma$ holds by Theorem 4.13 and the verification succeeds.

(5) If $\mathcal{T}^{\mathrm{A}} \models \sigma$ does not hold, observe the counterexample from the model checking. If it is a true counterexample, i.e. if a corresponding counterexample exists in the transition system $\mathcal{T}(G)$, it means $\mathcal{T}(G) \not\models \sigma$ and the verification fails.

(6) If it is not a true counter example, add p-formulae so that the counterexample does not occur and go back to Step (4).

The construction of the abstract structure in Step (4) has been automated in our framework. The observation in Step (5) and the addition of p-formulae in Step (6) need to be done manually by the person who conduct the verification.

4.7 Implementation

4.7.1 MLAT

We build a tool called MLAT (Modal Logic Abstraction Tool) which implements the verification method so far, more specifically it automates Steps (1) and (4) in Section 4.6.

There were several points we need to fix as design decision of the tool. First, the logic to use in p-formulae. In MLAT we decided to use the hybrid two-way CTL. This is a sublogic of $\mathrm{L}_0(\mu_{\mathrm{AF}}, @, ^{-})$, and the usage of fixed-point operators are restricted to the following two patterns: $\mathbf{E}_m(\varphi \,\mathbf{U}\, \psi) = \mu X(\psi \vee (\varphi \wedge \langle m \rangle X))$ and $\mathbf{A}_m(\varphi \,\mathbf{U}\, \psi) = \mu X(\psi \vee (\varphi \wedge [m] X))$. Abbreviations such as $\mathbf{E}_m \mathbf{F}\, \varphi = \mathbf{E}_m(\mathbf{true}\, \mathbf{U}\, \psi)$ are also used.

Second, the precondition function. We adapt the definition of the function wp described in Section 4.4 to the hybrid two-way CTL and define a function for the logic. The counterpart of Theorem 4.9 holds for the hybrid two-way CTL [65].

Third, function sat, the function that judges the satisfiability. We take the FUNC decision procedure described in Section 3.2 with a few changes and regard it as the function sat. Since we further restrict ourselves to CTL rather than alternation-free μ-calculus, some simplification is possible. For example the range of the first component x of the pair (x, y) in the node of the tableau can always be $0, \infty$. This is because we can check the loop-freeness in a different way. As we discussed in Chapter 3, although $\mathrm{Sat}(\mu_{\mathrm{AF}}, @, ^{-})$ is not decidable, the decision procedure is still sound (and not complete). Therefore the function sat satisfies the condition of Definition 4.11.

Fourth, we use model checker NuSMV[12] in Step (4).

Figure 4.5 shows an example input file for MLAT. The first section Decl declares variables, fields and so on. In the next section Source, a PML program is to be written. In order to specify the set $G \subseteq |\mathrm{CFG}(P)|$, labels can be placed. In this example two labels `start` and `end` are specified. The labels must be declared in the Decl section. MLAT generates labels automatically at all head points of while loops, unless the user already place one, in order to guarantee that $\mathrm{path}(G)$ is finite. Abstraction predicates are to be placed in the Pred section. The predicate `q1` in this example expresses p-formula $@x\, (\mathbf{E}_{\mathtt{next}}\mathbf{F}\, (u \wedge \neg \mathrm{NULL}))$, which is equivalent to $@x\, \mu X((u \wedge \mathrm{NULL}) \vee \langle \mathtt{next} \rangle X)$. The final section Spec is for specification formulae. The formula `s1` in the example is equivalent to the formula σ_1 in Table 4.3 and `s3` to σ_3.

For given input file, MLAT generates the abstraction transition system described in Definition 4.12. Figure 4.6 illustrates the generated transition system. For example

```
%%Decl
  Var x,y,t,u;
  Field next;
  Label start, end;
%%Source
  start:
    y := NULL;
  //  _auto1: (A label is automatically generated here.)
    while (!(x == NULL)) {
      t := y;
      y:= x;
      x:= x.next;
      y.next:=t;
    }
  end:
%%Pred
  q1 = x ==> E<next>F (u & !NULL);
    //  Node u is reachable from variable x.
  q2 = y ==> E<next>F (u & !NULL);
    //  Node u is reachable from variable y.
%%Spec
  s1 = [](start && q1 -> [](end -> q2));
    //  If q1 holds at start, q2 holds at end.  This spec is satisfied.
  s2 = [](end -> q2);
    //  q2 holds at end (for any input).  This spec is not satisfied.
  s3 = [](start && !abort -> []!abort)
    //  The program does not abort.  This spec is satisfied.
```

Figure 4.5: Example Input for MLAT

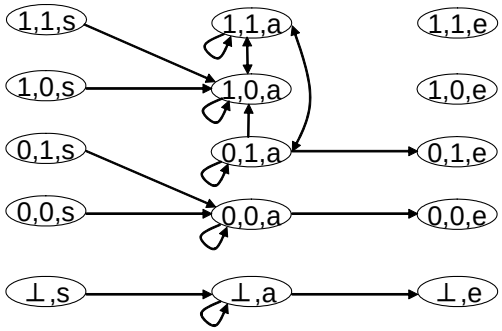


Figure 4.6: Abstraction Transition System MLAT Generates

node $(1, 0, s)$ in the figure expresses that predicate `q1` is true, `q2` is false, and the node in the control flow graph is s. One can model-check for each specification formulae. For example specification `s1` is satisfied because there is no path that starts at $(1, 1, s)$ or $(1, 0, s)$ and ends in $(1, 0, e)$ or $(0, 0, e)$.

4.7.2 Verification Examples

We verify specifications (1), (2), and (3) in Section (4.3.2) using MLAT.

The input file shown in Figure 4.5 is used to verify (1) and (3). The abstraction predicates used in the file are only those which appear in the specification formulae. It takes about 1.2 seconds to generate the abstract transition system and to model-check the resulting transition system by invoking NuSMV. (Pentium 4, 2.4GHz, 512MB memory, Windows XP.)

Spec (2) is not so simple compared with (1) and (3). When we use only those p-formulae appeared in the specification formula as the abstraction predicates, a counterexample is generated in the model checking. By checking against the original transition relation, it turns out to be a false counterexample. In order to remove the counter example, we add more predicates such as $@x\, u_2$, $@y\, u_1$ and $@y\, (\mathbf{E}_n \mathbf{F}\, \neg u_2)$, and MLAT reports that the specification formulae is satisfied in the abstract transition system. The time for MLAT to run is about 37 seconds in the same environment.

Our method cannot verify specifications for liveness properties such as (4). This will be discussed in Section 7.2.2.

4.8 Related Work

There are various researches to analyze programs that manipulate pointers by means of abstraction.

Sagiv et al. conduct one of the most successful research [57]; they also have a tool called TVLA based on their theory. It uses Kleene's three-valued logic for abstraction. The logic has three truth-values: 0 (false), 1 (true), and 1/2 (unknown). This is useful to express abstract relations; if abstract nodes α and β are represents concrete nodes a_1, a_2 and b_1 respectively and there is relation n between a_1 and b_1 but not between a_2, b_1. In such a case the truth value of relation n between α and β can be considered as $1/2$ thanks to the three-valued logic. TVLA calculates an approximation of the concrete heap by using the method of abstract interpretation [18] in the abstract structure with three-valued truth values.

Separation Logic ([53]) is an extension of the Hoare Logic and has operators to handle the status of the heap [23]. It has been used to write Hoare style proofs for pointer manipulating programs. Recently the logic is also used as methods for the automatic analysis of TVLA style.

Our framework described in this chapter is also classified as abstraction. One of the difference is that we use the predicate abstraction technique and TVLA is based on abstract interpretation. Another is we use a restriction of the first-order predicate logic (FOL), while TVLA use an extension. This also comes from the first point: since we use the predicate abstraction, we wish to have a decidable logic, and such a logic cannot contain the FOL.

In that sense, our approach is more closer to Dams and Namjoshi [19] and Balaban *et al.* [3]. Both use the predicate abstraction technique to analyze programs manipulating pointers. In the former, the abstract structure is not computed at once; they need iteration and during the iteration hints from the user is necessary. In the latter approach, the logic is decidable since it has the small model property, and a decision procedure based on this property is used to calculate the abstract transition system. Both of the approach define a logic for expressing predicates and they are closely related to the programming language, while in our approach we use modal logics widely used and general decision procedure is applied to the analysis.

Chapter 5

Verification with Manually Constructed Abstraction

In the previous chapter we propose a method to verify properties of programs that manipulating pointers based on the predicate abstraction framework. It is simple in the sense that the user only needs to find an appropriate set of predicates. Once it is fixed, the abstract transition system is generated automatically.

However, the method is not very efficient when the properties to verify is complex. Since all the selected predicates are globally used, the resulting abstract transition system becomes large more than necessary. In such a case, it is useful for the user to construct the abstract space manually so that redundant combinations of predicates do not appear in the abstract transition system. Once the target transition system is established, the verification can be conducted automatically in the same way as in the previous chapter.

In this chapter we illustrates how the user specifies the abstract transition based on his intuition of the reasoning why the program satisfies the specifications. We also have an implementation that automatically judges whether the abstract transition system constructed by the user is correct. We take the Deutsch-Schorr-Waite (DSW) marking algorithm as a running example and prove its safety properties.

5.1 Deutsch-Schorr-Waite Marking Algorithm

5.1.1 Algorithm

The original DSW algorithm handles arbitrary number of successors per a node, but for simplicity we follow the convention in the literature [43], [36] and assume that

each node has at most two successors, named l (for $l$eft) and r (for $r$ight).

An ordinary marking algorithm based on the depth first search uses a stack to remember the node where we should return after the search from the current node completes. The DSW algorithm does not use a stack. Instead it temporarily alters the pointers of the current node to remember the node to return to, and change them back again when the search from the current node completes. To realize that, each node needs to keep its status, which is one of the following:

(a) The node has not yet been visited.

(b) The node has been visited and the left neighbors are being processed.

(c) The search for the left neighbors of the node has completed and the right neighbors are being processed, or the search from the node has completed.

Since it is a marking algorithm, it is obvious that each node need to have flag to record whether it has been marked or not. We call the flag m. Adding to that, we need another flag for each node to distinguish the statuses (b) and (c). We call the flag s since the operation taken when a node changes its status from (b) to (c) is called $s$wing. Thus at status (a) m is false, at status (b) m is true and s is false, and at status (c) both m and s are true.

Now we explain the DSW algorithm. There are finitely many nodes all of which are unmarked, that is, flag m is false, and a node is designated as root. The objective of the algorithm is to mark all the node which are reachable (through either l or r) from the root. We use two pointer type variables p and t. At the beginning, t points to the root and t points to nothing. In the following we call the node t points to the current node and the node p points to the previous node. Then the loop starts:

- If t points to an unmarked node, do the following operation called *push*. Set the m flag and unset the s flag of the current node. Make the left pointer of the current node points to the previous node.[1] Change t and p so that t points to the left successor of the current node and p the current node.

- If t points to a marked node or nothing and p points to a node whose s flag is unset, do the following operation called *swing*. Set the s flag of the previous node. Make the right pointer of the previous node points to the node its left pointer now points to, and make its left pointer points to the current node. Change t so that t points to the node the right pointer of the previous node pointed to.

[1] It means that if p pointed to nothing then the left pointer of the current node points to nothing. We will omit this kind of notice in the following description.

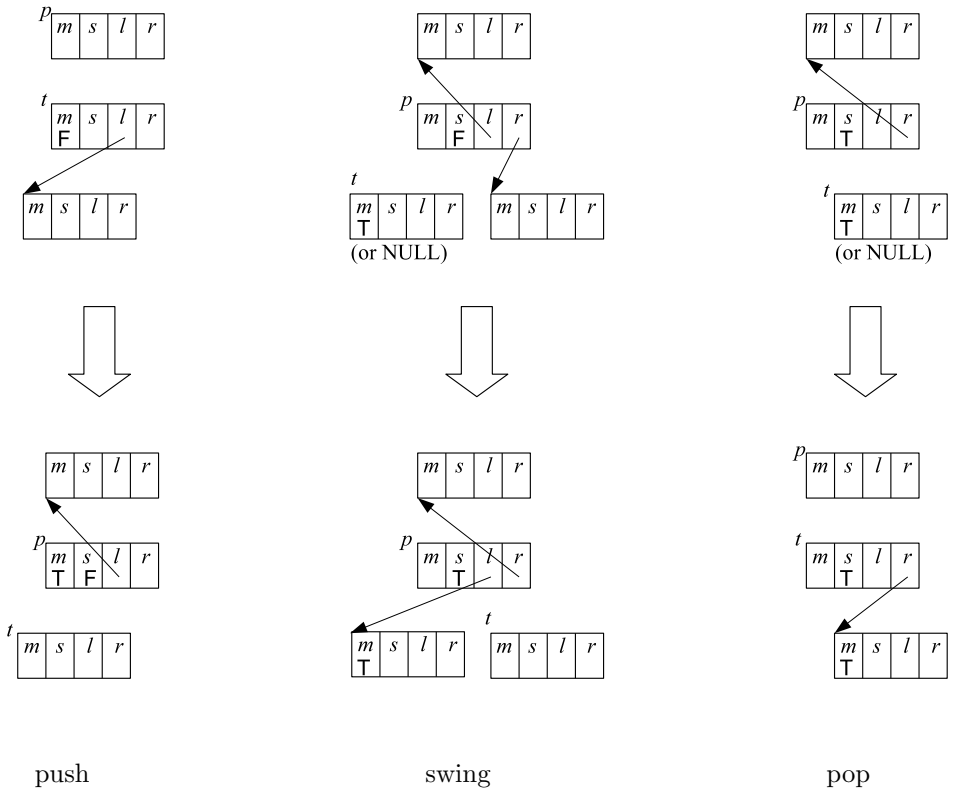


Figure 5.1: Operations in the DSW Algorithm

- If t points to a marked node or nothing and p points to a node whose s flag is set, do the following operation called *pop*. Make the right pointer of the previous node points to the current node. Change t and p so that t points to the previous node and p points to the node the right pointer of the previous node pointed to.
- If t points to a marked node or nothing and p points to nothing, exit the loop.

The operations and their conditions are illustrated in Figure 5.1. Detailed algorithm written in PML is shown in Figure 5.2.

5.1.2 Specifications

The DSW is a marking algorithm and it is supposed to satisfy the following properties:

(1) At the end of the execution, each pointer points to the same node as it pointed to at the beginning of the execution (although they are altered during the exe-

```
t := root;
p := NULL;
while ( !((t == NULL || t.m) || p != NULL) ) {
  if ( !(t == NULL || t.m) ) {
    // push
    x := p;
    p := t;
    t := t.l;
    p.l := x;
    p.m := true;
    p.s := false;
  } else if ((t == NULL || t.m) && p != NULL && !p.s) {
    // swing
    x := t;
    t := p.r;
    y := p.l;
    p.r := y;
    p.l := x;
    p.s := true;
  } else { // (t == NULL || t.m) && p != NULL && p.s
    // pop
    x := t;
    t := p;
    p := p.r;
    t.r := x;
  }
}
```

Figure 5.2: Deutsch-Schorr-Waite Marking Algorithm

cution).

(2) All nodes that are reachable from the root are marked during the loop.

(3) No node that is not reachable from the root is not marked.

We will verify the specs (1) and (2) in this chapter. The spec (3) can be shown in similar way to the spec (2) so we omit it.

5.1.3 Informal Correctness

It is not very clear that the DSW algorithm satisfies the specifications listed above. In this subsection we give an intuitive explanation why the DSW algorithm satisfies specs (1) and (2). It is the basis of the construction of our abstract space which will be described in the next section.

In the following discussion we often identify a variable and the node which it points to. For example we say t = root at the beginning. Also for simplicity we introduce an imaginary node called nil and when variables t and p and left and right pointers of the nodes point to nothing we regard they point to nil. We also regard that the m flag of nil is set and the left and right pointer of nil point to itself. Then the condition "t points to nothing or a marked node" can be replaced with simpler condition "t is marked".

The following properties for nodes x and y will be used.

- $\mathrm{rum}(x, y)$: node y is Reachable from node x by following pointers l and r and all nodes on one of such paths, including x and y, are $U\mathrm{n}M$arked.
- $\mathrm{rs}(x, y)$: node y is Reachable from node x by following pointer l or r depending on whether it has been Swung – pointer r is to be followed if flag s of the node is set[2] and l otherwise.
- $\mathrm{rsum}(x, y)$: there is some node z such that $\mathrm{rum}(x, z)$ and $\mathrm{rs}(z_{\mathrm{R}}, y)$ holds where z_{R} is the right successor of z.

First as preparation we claim that $\mathrm{rs}(p, \mathrm{nil})$ holds and all the nodes on the path between p and nil which guarantees $\mathrm{rs}(p, \mathrm{nil})$ are marked. This can easily been shown on the induction: at the beginning the claim holds and all the operations push, swing and pop does not break it. From the claim we know that p is always marked.

[2]Flag s shows whether the node has been swung only if it is marked, but there will be no problem about that. In the following discussion whenever $\mathrm{rs}(x, y)$ is used all nodes on the path from x and y are marked. Anyway we can define $\mathrm{rs}(x, y)$ for any x and y by referring flag s.

Next we show that the spec (1) is satisfied. We pick up an arbitrary node a and show that this node is marked when the algorithm terminates. We claim that while a is unmarked either $\mathrm{rum}(t, a)$ or $\mathrm{rsum}(p, a)$ holds. Note that when $\mathrm{rum}(t, a)$ holds the algorithm cannot terminate since t is unmarked and the same for $\mathrm{rsum}(p, a)$ since p is not nil. Therefore once the claim is established we conclude that a should be marked when the algorithm terminates.

At the beginning $\mathrm{rum}(t, a)$ holds since $t = \mathrm{root}$ and no node is marked.

Assume $\mathrm{rum}(t, a)$. Since t is unmarked, push is the only possible operation. Let t_{L} (and t_{R} respectively) be the left (and right respectively) successor of t. Since $\mathrm{rum}(t, a)$ holds either $\mathrm{rum}(t_{\mathrm{L}}, a)$ or $\mathrm{rum}(t_{\mathrm{R}}, a)$ (or both) must hold. In the first case, $\mathrm{rum}(t, a)$ still holds after the operation pop since t_{L} becomes t. In the second case, property $\mathrm{rum}(t, a)$ can break after the operation. But we still have $\mathrm{rum}(p, a)$, since t becomes p and the right pointer remains intact. By taking $z = p$, we have $\mathrm{rsum}(p, a)$.

Next assume $\mathrm{rsum}(p, a)$. Take node z such that $\mathrm{rs}(p, z)$ and $\mathrm{rum}(z, a)$. and path π from z to a such that no nodes on π are marked. If t is on π we also have $\mathrm{rum}(t, a)$ and the case has already been checked, so we assume that t does not appear on π.

Any of the operation push, swing and pop can occur. When the operation is push, $\mathrm{rs}(p, z)$ does not break with the operation by the definition and $\mathrm{rsum}(z, a)$ still holds since t does not appear on π. When the operation is pop, $\mathrm{rs}(p, z)$ does not break and $\mathrm{rsum}(z, a)$ still holds since p does not appear on π because p is marked. When the operation is swing, there are two cases. If $p = z$, $\mathrm{rum}(t, a)$ now holds since z_{R} was the right successor of p. If $p \neq z$, $\mathrm{rs}(p, z)$ does not break and $\mathrm{rsum}(z, a)$ still holds since p does not appear on π because p is marked.

Thus the spec (1) holds. The spec (2) can be shown in a similar manner and we omit details.

5.2 Abstract Space

We use the language $\mathrm{L}_0(\mu_{\mathrm{AF}}, @)$ to express the properties of the heap.

5.2.1 Construction of Abstract Space

We could verify specs (1) and (2) independently, but in order to avoid to keep two similar but different transition systems, we will verify them simultaneously. To verify them, we introduce nominals a, b and c. Intuitively, a represents arbitrary node, b is the left successor of a, and c is the right successor of a.

Recall $\varphi_1 \rightarrow \varphi_2 \,;\varphi_3$ is an abbreviation for $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \varphi_3)$. Let us define three formulae corresponding to the properties used in the previous section :

$$\begin{aligned} \mathrm{RUM}(x) &\stackrel{\mathrm{def}}{=} \mu X(\neg m \wedge (x \vee \langle l\rangle X \vee \langle r\rangle X)) \\ \mathrm{RS}(x) &\stackrel{\mathrm{def}}{=} \mu X(x \vee (s \rightarrow \langle r\rangle X \,;\langle l\rangle X)) \\ \mathrm{RSUM}(x) &\stackrel{\mathrm{def}}{=} \mathrm{RS}(\langle r\rangle \mathrm{RUM}(x)) \end{aligned}$$

where x is a place holder for a formula. For example, for formula φ, $\mathcal{K}, s \models \mathrm{RUM}(\varphi)$ means that there is a node s' such that $\mathcal{K}, s' \models \varphi'$ and $\mathrm{rum}(s, s')$ holds. Therefore for nominals x and y, formula $@x\,\mathrm{RUM}(y)$ means $\mathrm{rum}(\hat{\lambda}(x), \hat{\lambda}(y))$. Similar for the other two formulae.

We convert the informal reasoning described in Section 5.1.3 into an abstract transition system. Refer to Figure 5.3 for the result. Each abstract state, shown by a box with numbers, such as "01", "02", *etc.*, in the figure, represents the nodes of the pointer structure that satisfy all the formulae in the box. In order to save space, we omit the following two formulae that should be added to all the boxes in the figure:

- $@a\,\neg\mathrm{nil}$
- $@p\,\mu X(\mathrm{nil} \vee (m \wedge (s \rightarrow \langle r\rangle X \,;\langle l\rangle X)))$

States 01 and 03 are designated as the initial states.

The structure of the program is quite simple; it consists of a single while loop. At the top of the loop, due to the condition, there are four possibilities; (1) executes push, (2) executes swing, (3) executes pop, or (4) exits from the loop and terminates. The program location of all the abstract states shown in the figure is the top of the loop. Arrows between the boxes show possible transitions. Thus, for example, at the state 03, push is the only possible execution and the resulting state is either 11 or 14. We also claim that exiting from the loop only occurs at the state 41, although it is not explicitly written in the figure.

We explain how the abstract transition system in the figure is constructed.

Recall that when we prove the spec (1), we prove and use the fact that either $\mathrm{rum}(t, a)$ or $\mathrm{rsum}(p, a)$ is satisfied until a gets marked. So we prepare two abstract states, numbered 01 and 02 in the figure, one is for the status in which $\mathrm{rum}(t, a)$ holds therefore we put formula $@t\,\mathrm{RUM}(a)$ in it, and the other is for $\mathrm{rsum}(p, a)$ and has $@p\mathrm{RSUM}(a)$. Since both are of interest when a has not yet marked, we also put $@a\neg m$ in them. Transitions between abstract states 01 and 02 should be as in illustrated in the figure (we will formally prove it later) from the discussion in Section 5.1.3.

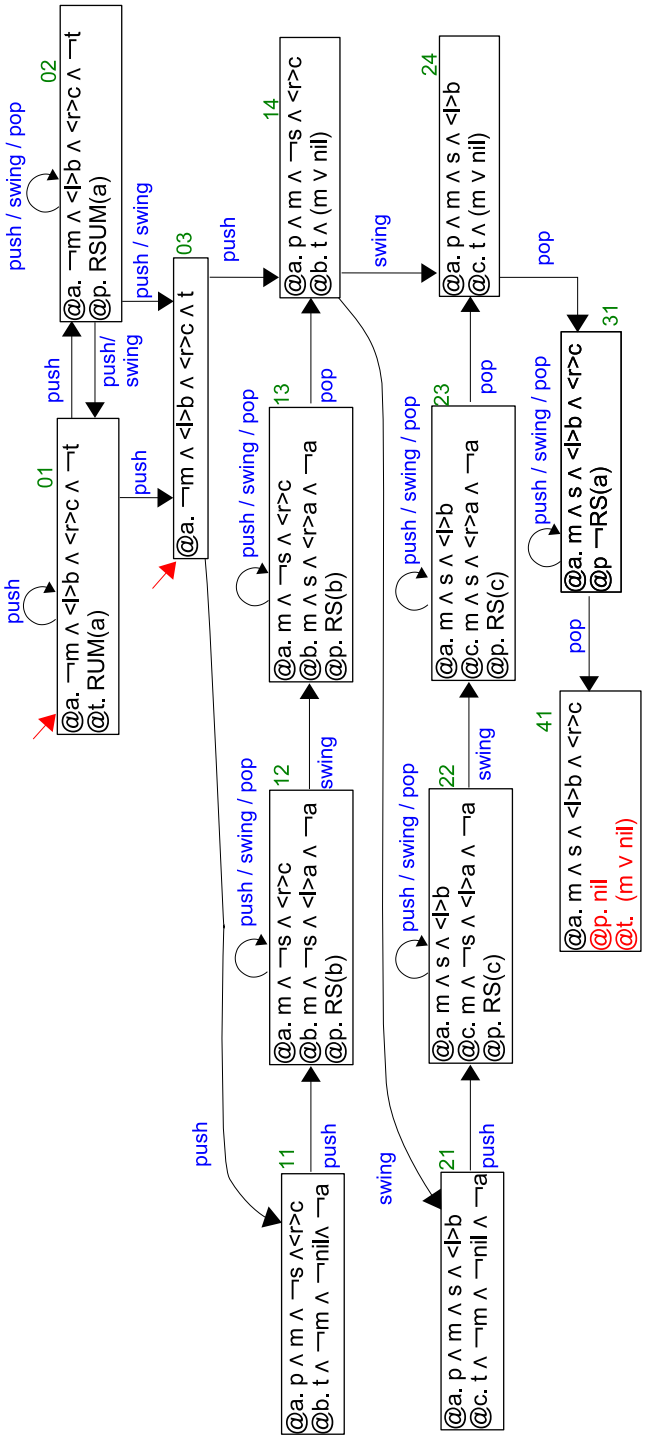


Figure 5.3: Abstract Transition System for DSW

Then a will be marked. To express the event, we create a new abstract state, numbered "03" in the figure. Just before a node gets marked, it is pointed to by variable t. So formulae $@a\, t$ and $@a\, \neg m$ are put in the state. Also to distinguish states 01 and 02 from state 03, $@a\, \neg t$ are added to them.

In the states described so far a has not yet been visited therefore pointers l and r are not altered. Therefore $@a\, \langle l \rangle b$ and $@a\, \langle r \rangle c$ can be put in them and they are necessary for the spec (2).

Thus states 01, 02 and 03 are constructed from the informal reasoning of the spec (1). The rest of the states are constructed by considering the informal reasoning of the spec (2). We omit the details.

5.2.2 Verification of the Transition System

For a state a in the abstract transition system, we denote by $g(a)$ the corresponding node in the CFG; and by $\varphi(a)$ the conjunction of the formulae associated with a. For a node g in the CFG, we denote by $A(g)$ the set of the corresponding states in the abstract transition system. For an edge σ in the CFG starting from $g(a)$, we denote by $\mathrm{Succ}(\sigma, a)$ the set of abstract nodes that are connected with a by transitions corresponding to σ.

An abstract state a is said to be *correct* if for each edge σ starting from $g(a)$, the following formula is valid, i.e., its negation is unsatisfiable:

$$\varphi(a) \to \bigvee_{a' \in \mathrm{Succ}(\sigma, a)} \mathrm{wp}(\sigma, \varphi(a'))$$

where the weakest precondition $\mathrm{wp}(\cdot, \cdot)$ is defined in Definition 4.8. We say that the abstraction system is correct if all the abstract state is correct.

Assume the abstract transition system is correct. Let g_0 be the initial node of the CFG and assume for any initial pointer structure $\mathcal{S}_0$, $\mathcal{S}_0 \models \bigvee_{a \in A(g_0)} \varphi(a)$ holds. Then it is clear from Theorem 4.9 that for any node g in the CFG, if $(\mathcal{S}, g)$ is reachable from a concrete initial state, $\mathcal{S} \models \bigvee_{a \in A(g)} \varphi(a)$ holds.

Let us apply the above discussion to the abstract transition system we constructed for the DSW algorithm. It is clear that the initial pointer structure satisfies the conjunction of the formulae either in state 01 or 03. Once we establish that the transition system is correct, we conclude that at the end of the program the formulae associated to state 41 must be satisfied, since this is the only node where the condition of the while loop can fail. Among the formulae associated to state 41, we find formula $@a\, m$. It expresses that the node a is marked and that is what we need to show for

Table 5.1: Experimental Results

state	01	02	03	11	12	13	14
push	5458	12240	2067	1973	2633	2463	1598
swing	1829	29826	1620	1623	6519	4791	2119
pop	1867	14618	1645	1587	2647	3081	1613
terminate	1903	2057	1634	1622	1935	1877	1588

state	21	22	23	24	31	41
push	1988	2703	2657	1624	2687	1578
swing	1623	4720	3450	1596	3647	1604
pop	1685	2914	2838	2436	2778	1627
terminate	1672	1877	1841	1662	1674	1895

spec (1). Similarly, formula $@a\ \langle l\rangle b \wedge \langle r\rangle c$ ensures that the algorithm satisfies spec (2).

5.3 Implementation

We build a tool which implements the method described in the previous section and verifies given abstract transition system. The logic for describing abstract states is $\mathrm{L}_0(\mu_{\mathrm{AF}}, @)$. For this implementation we use postconditions instead of preconditions. Postconditions can be calculated by using the weakest precondition and we can avoid reverse modalities, although we may lose accuracy in some cases. For the satisfiability judgment, we take the FUNC decision procedure described in Section 3.2 with a few changes for the performance improvement.

Figure 5.4 shows one of the input files for the verification of the DSW algorithm. This file is for the correctness of state 01 for the operation push. The last line of the input file claims that the successor of state 01 is either state 01, 02 or 03.

Table 5.1 shows the running time in milliseconds required to check that each abstract node is correct for each transition. The computer used for the the measurement is Xeon 3GHz*2 CPU, 4GB memory running Redhat Enterprise Linux.

5.4 Related Work

In this section, we discuss related work on the verification of the DSW marking algorithm.

```
Var             a,b,c,p,t,x;
BoolFld         marked,swung;
PointerFld      left,right;

Source push {
  assume(t != NULL && !t.marked);
  x := p;
  p := t;
  t := p.left;
  p.left := x;
  p.marked := true;
  p.swung := false;
}

Formula
  always = @a !nil &
           @p mu X0 ( nil | (marked & (swung ? <right>X0 : <left>X0))),
  rum_a  = mu XC (!marked & (a | <left>XC | <right>XC)),
  rsum_a = mu XE (<right>rum_a | (swung ? <right>XE : <left>XE)),
  a_init  = @a (!marked & <left>b & <right>c),
  state01 = a_init & @a !t & @t rum_a,
  state02 = a_init & @a !t & @p rsum_a,
  state03 = a_init & @a t,
  pre = always & state01,
  post = always & (state01 | state02 | state03);

ht(pre, push, post);  // Check that post holds after push is executed
                      // if pre holds beforehand.
```

Figure 5.4: Example of Input File

The algorithm has been very often used as a case study of the analyzing method for pointer manipulating programs. For example Bornat wrote in [9], "The Schorr-Waite algorithm is the first mountain that any formalism for pointer analysis should climb." This algorithm is appropriate for such case studies as it is fairly complex, although not very large in size, and it is not obvious whether the algorithm has the desired properties.

The DSW algorithm in its original form was provided in [60] and its correctness is discussed in it.

One of the earliest research to give a formal proof that can be checked by computer is [9] by Bornat. The proof assistant Jape was used.

Mehta and Nipkow used higher-order system Isabelle/HOL in [47]. and Yang wrote a machine checkable proof in the logic called Bunched Implications [78], which becomes the base of Separation Logic.

In the abovementioned research, a proof is constructed manually and then a proof assistant on the computer checks it. The research by Hubert and March'e is the same from the aspect, but unlike the others they verified not only safety properties but also the termination of the algorithm [37]. They use a tool called CADUCEUS that handles C source code directly, and embed invariants into the code as comments in a special form, which the tool checks whether they are really invariants or not.

Loginov *et al.* automated the verification. They use the tool TVLA [57], which performs abstract interpretation [18] on pointer manipulating programs. Since the default predicates that TVLA has are not enough for verifying the properties of the algorithm, they add predicates for this verification. TVLA completes the verification in several hours. They verify both safety properties and the termination, but the input data is restricted to the shape of a tree.

Our approach is in a sense intermediate of the two directions. It is not fully automatic because the user need to build the abstract transition system manually. However, the running time is much shorter. On the other hand, the burden of the user is lighter compared to write entire proof. For example it is not necessary to write down a complete invariant; what is needed is to divide the situation into some specific cases and specify how each case changes by the execution of the code fragment.

Chapter 6

Application to Cellular Automata

In this chapter we apply satisfiability checking of modal logics to analyze a graph transition system called Linear Cellular Automata. Researches exist in this area and some of them already applied modal logics for abstraction [33, 63]. We improve those existing results and propose a simple analyzing method.

The contents of this chapter has been published as a part of [64].

6.1 Cellular Automata

In this chapter we analyze one-dimensional cellular automata [77], which have the following properties.

- A cellular automaton consists of an array of cells that are unbounded in both direction.
- Each cell has the status of 0 or 1.
- The status of the cell changes per a unit of time.
- The status change occur simultaneously for all the cells. We say a generation proceeds when the unit of time passes and the status of the cells changes.
- The change of the status of a cell depends on the current status of the cell itself and that of some neighbors of the cell. It does not depend on previous statuses.
- A same rule is applied to the all the cells.

An in finite sequence of 0 and 1 that represents the status of the cells at a time is called a *configuration*.

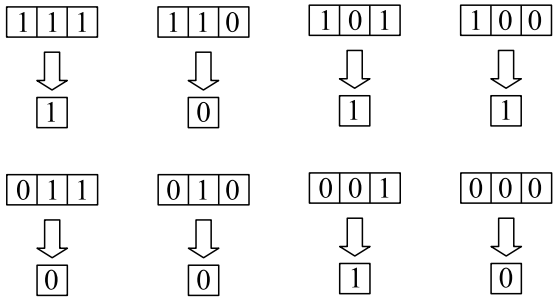


Figure 6.1: The rule 178

Example 6.1 Consider a rule illustrated in Figure 6.1. Each of the eight block in the figure represents the condition and the result for the change of the status. For example The upper-right corner expresses that "if the current status of the immediate-left, itself, and immediate-right, respectively, are 1, 0, and 0, respectively, then the status of the next generation is 1." The rule is called the rule 178, which is the binary number 10110010_2 obtained by sorting the values of the new generations as in the figure.

If in the initial configuration only one cell has status 1 and all the others have status 0, the subsequent generations can be decided using the rule, as shown in the Configuration column in Figure 6.2.

In this chapter we take the set PS of propositional symbols to be $\{1\}$, that is the only propositional symbol is "1". Only one modality symbol m is considered. A configuration C can be considered as a total Kripke structure $\mathcal{K}(C) = (S, R, \lambda)$: the underlying set S consists of the cells of the automaton, $(c_1, c_2) \in R(m)$ if and only if c_2 is the immediate right of c_1, and $\lambda(1)$ is the set of the cells whose status is 1.

Our analyze method that will be described in the next section constructs a sequence of finite total Kripke structures that approximates configurations at each generation. We have shown the resulted Kripke structures for each generation for the rule 178 in the Approximation column of Figure 6.2 as an example. In the figure the transition relation $R(m)$ is shown as arrows and the label "1" on a node shows that the node is an element of set $\lambda(1)$ and the label "0" on a node shows that the node is not an element of the set. For example in Generation 0, a simulation can be defined from the configuration to the Kripke structure so that the center cell (the cell with status 1) relates to the center node (the node labeled "1"), the cells located in the left (or right, respectively) side of the center cell relate to the node left (or right, respectively) to the

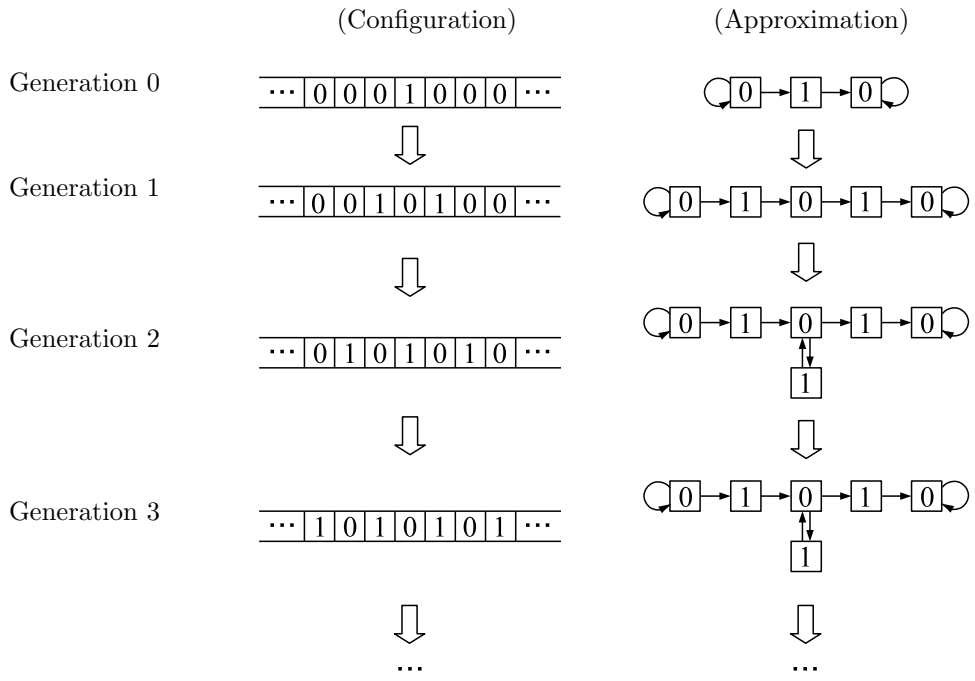


Figure 6.2: Approximation Sequence of Kripke Structures

center node. Simulations for other generations can be defined similarly. We observe that the approximation Kripke structures for Generation 2 and after are identical. From the observation we can conclude, for example, that a configuration that have two adjacent cells both with status 1 cannot occur.

6.2 Abstraction by Modal Logic Formulae

6.2.1 2LTL

We define formulae φ in *2LTL*, or *Two-way Linear Temporal Logic* with the following BNF:

$$\varphi ::= \mathbf{1} \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\,\varphi \mid \overline{\mathbf{X}}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi \mid \varphi\,\overline{\mathbf{U}}\,\varphi$$

Abbreviations such as $\mathbf{0} \stackrel{\text{def}}{=} \neg\mathbf{1}$, $\mathbf{true} \stackrel{\text{def}}{=} \mathbf{0} \vee \mathbf{1}$, $\mathbf{false} \stackrel{\text{def}}{=} \mathbf{0} \wedge \mathbf{1}$, $\varphi_1 \wedge \varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\varphi_1\,\mathbf{R}\,\varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1\,\mathbf{U}\,\neg\varphi_2)$, $\varphi_1\,\overline{\mathbf{R}}\,\varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1\,\overline{\mathbf{U}}\,\neg\varphi_2)$, $\mathbf{F}\,\varphi \stackrel{\text{def}}{=} \mathbf{true}\,\mathbf{U}\,\varphi$, $\overline{\mathbf{F}}\,\varphi \stackrel{\text{def}}{=} \mathbf{true}\,\overline{\mathbf{U}}\,\varphi$, $\mathbf{G}\,\varphi \stackrel{\text{def}}{=} \mathbf{false}\,\mathbf{R}\,\varphi$, and $\overline{\mathbf{G}}\,\varphi \stackrel{\text{def}}{=} \mathbf{false}\,\overline{\mathbf{R}}\,\varphi$ will also be used.

Let ω be the set of natural numbers and $\mathbf{Z}$ the set of integers. Let $\pi : \mathbf{Z} \to \{0, 1\}$ be a function from the set of integers to $\{0, 1\}$ and φ be a 2LTL formula. We define

the relation "π satisfies φ" or $\pi \models \varphi$ by induction on the construction of π as follows:

$$
\begin{array}{lll}
\pi \models \mathbf{1} & \text{iff} & \pi(0) = 1 \\
\pi \models \neg\varphi & \text{iff} & \pi \not\models \varphi \\
\pi \models \varphi_1 \vee \varphi_2 & \text{iff} & \pi \models \varphi_1 \text{ or } \pi \models \varphi_2 \\
\pi \models \mathbf{X}\,\varphi & \text{iff} & \pi^1 \models \varphi \\
\pi \models \overline{\mathbf{X}}\,\varphi & \text{iff} & \pi^{-1} \models \varphi \\
\pi \models \varphi_1\,\mathbf{U}\,\varphi_2 & \text{iff} & \exists j \in \omega.\, \pi^j \models \varphi_2 \text{ and } \forall i \in \omega.\, i < j \implies \pi^i \models \varphi_1 \\
\pi \models \varphi_1\,\overline{\mathbf{U}}\,\varphi_2 & \text{iff} & \exists j \in \omega.\, \pi^{-j} \models \varphi_2 \text{ and } \forall i \in \omega.\, i < j \implies \pi^{-i} \models \varphi_1
\end{array}
$$

where for $i \in \mathbf{Z}$, $\pi^i : \mathbf{Z} \to \{0,1\}$ is defined as $\pi^i(x) = \pi(i + x)$.

Let $\mathcal{K} = (S, R, \lambda)$ be a total Kripke structure. Function $\pi : \mathbf{Z} \to S$ is called a *path* if $(\pi(x), \pi(x+1)) \in R(m)$ for all $x \in \mathbf{Z}$. The relation "π satisfies φ" or $\mathcal{K}, \pi \models \varphi$, for path π and 2LTL formula φ, is defined as $\pi' \models \varphi$ where function $\pi' : \mathbf{Z} \to \{0,1\}$ is defined so that $\pi'(x) = 1 \iff \pi(x) \in \lambda(1)$ holds for $x \in \mathbf{Z}$. For $s \in S$, we write $\mathcal{K}, s \models \mathbf{A}\,\varphi$ ($\mathcal{K}, s \models \mathbf{E}\,\varphi$, respectively) if $\mathcal{K}, \pi \models \varphi$ for any (some, respectively) path π such that $\pi(0) = s$.

6.2.2 Abstract Cells

Let $\Phi = (\varphi_1, \ldots, \varphi_n)$ be a sequence of 2LTL formulae. We call a sequence of symbols T and F of length n an *abstract cell*. There are 2^n abstract cells. Cells of which configurations consist are called *concrete cells* to distinguish them from abstract cells.

Let C be a configuration and c be a cell of C. The abstract cell $(b_1, \ldots, b_n)$ induced by C and c, which is denoted by $\alpha(c, C)$, is defined as follows:

$$
b_i = \begin{cases} \mathrm{T} & \text{if } \mathcal{K}(C), c \models \mathbf{E}\,\varphi_i \\ \mathrm{F} & \text{otherwise} \end{cases}
$$

Note that there is essentially only one path in $\mathcal{K}(C)$, or more precisely, if π_1 and π_2 are paths in $\mathcal{K}(C)$ then there is an integer i such that $\pi_1 = (\pi_2)^i$.

Example 6.2 Let $\Phi = (\overline{\mathbf{X}}\,\mathbf{1}, \mathbf{1}, \mathbf{X}\,\mathbf{1}, \mathbf{F}\,\mathbf{1})$. Abstract cells induced by the concrete cells of the initial configuration in Figure 6.2 can be illustrated as follows:

$\cdots$	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	$\boxed{1}$	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	$\cdots$
$\cdots$	(FFFT)	(FFFT)	(FFTT)	(FTFT)	(TFFF)	(FFFF)	(FFFF)	$\cdots$

All abstract cells located in the left (right, respectively) hand side to the cells shown above are (FFFT) ((FFFF), respectively). ∎

For abstract cell $a = (b_1, \ldots, b_n)$, we define formula φ_i^a by

$$\varphi_i^a = \begin{cases} \varphi_i & \text{if } b_i = \mathrm{T} \\ \neg\varphi_i & \text{if } b_i = \mathrm{F} \end{cases}$$

and we identify the abstract cell a and formula $\bigwedge_{i=1}^n \varphi_i^a$. For example the center abstract cell (FTFT) in the previous example is also written as $\neg\overline{\mathbf{X}}\,\mathbf{1} \wedge \mathbf{1} \wedge \neg\mathbf{X}\,\mathbf{1} \wedge \mathbf{F}\,\mathbf{1}$. With this notation, we have $\mathcal{K}(C), c \models \alpha(c, C)$.

6.2.3 Covering Set

Let C be a configuration. A set A of abstract cells *covers* C if contains the abstract cell $\alpha(c, C)$ for all the concrete cell c in C. This relation is denoted by $C \sqsubset A$. It is not possible to calculate the least covering set of given configuration C based on the definition since there are infinitely many concrete cells. We give several methods to judge whether given abstract cell a is induced by a concrete cell c in C, that is $a = \alpha(c, C)$ from a Kripke structure $\mathcal{K} = (S, R, \lambda)$ that simulates $\mathcal{K}(C)$. We call such a judgment a *existence judgment*. As we will see below, the judgments are not complete but sound. Since the number of abstract cells is finite (2^n), we can construct a covering set A as the set of abstract cells that are judged to be induced by concrete cells.

Existence Judgment A An abstract cell a is judged to be induced by a concrete cell if there is $s \in S$ such that $\mathcal{K}, s \models \mathbf{E}\,a$. ∎

Generally, for LTL and its variant logics, to decide whether $\mathcal{K}, s \models \mathbf{E}\,\varphi$ is satisfied for given Kripke structure $\mathcal{K}$, its state s and formula φ is called model checking [15]. Judgment A can be implemented using a 2LTL model checker.

A 2LTL formula φ is called *one-way* if it does not have both of past operators ($\overline{\mathbf{X}}$ and $\overline{\mathbf{U}}$) and future operators($\mathbf{X}$ and $\mathbf{U}$). For example formulae $\mathbf{X}\,\mathbf{1}$, $\mathbf{1}\,\overline{\mathbf{U}}\,\mathbf{0}$, and $\mathbf{1}$ are one-way, while formula $\mathbf{X}\,(\mathbf{1}\,\overline{\mathbf{U}}\,\mathbf{0})$ is not. When all formulae in Φ are one-way, for an abstract cell a, we denote by $\overrightarrow{a}$ the formula obtained from a by removing formulae in Φ or its negation that have past operators. Similarly for $\overleftarrow{a}$, but in this case not only removing formulae in Φ or its negation that have future operators, but also replacing remaining past operators with corresponding future operators. For example if $\Phi = \{\overline{\mathbf{X}}\,\mathbf{1}, \mathbf{0}, \mathbf{X}\,\mathbf{F}\,\mathbf{1}\}$ and $a = \overline{\mathbf{X}}\,\mathbf{1} \wedge \mathbf{0} \wedge \mathbf{X}\,\mathbf{F}\,\mathbf{1}$, then $\overrightarrow{a} = \mathbf{0} \wedge \mathbf{X}\,\mathbf{F}\,\mathbf{1}$ and $\overleftarrow{a} = \mathbf{X}\,\mathbf{1} \wedge \mathbf{0}$.

For Kripke structure $\mathcal{K} = (S, R, \lambda)$, we denote by $\overleftarrow{\mathcal{K}}$ the Kripke structure obtained from $\mathcal{K}$ by reversing the transition relations, namely (S, R', λ) where $R'(m) = R^{-1}(m)$ for $m \in \mathrm{MS}$.

Existence Judgment B This judgment can be applied when all formulae in Φ are one-way. An abstract cell a is judged to be induced by a concrete cell if there is $s \in S$ such that $\mathcal{K}, s \models \mathbf{E}\,\overrightarrow{a}$ and $\overleftarrow{\mathcal{K}}, s \models \mathbf{E}\,\overleftarrow{a}$. ∎

Judgment B is weaker than Judgment A, that is, it may judge more abstract cells to be induced by concrete cells. But it can be implemented using a LTL model checker rather than a 2LTL model checker.

Existence Judgment C An abstract cell a is judged to be induced by a concrete cell if there is $s \in S$ such that for all $i = 1, \ldots, n$, $\mathcal{K}, s \models \mathbf{E}\,\varphi_i^a$. ∎

Judgment C is also weaker than Judgment A, but while Judgment A requires $2^n|S|$ times 2LTL model checking, Judgment C requires $2n|S|$ times. If all formulae in Φ are one-way, LTL model checking is enough for Judgment C.

Example 6.3 Suppose $\Phi = \{\overline{\mathbf{X}}\,\mathbf{1}, \mathbf{1}, \mathbf{X}\,\mathbf{1}, \mathbf{F}\,\mathbf{1}\}$. We take the upper-right Kripke structure in Figure 6.2 as $\mathcal{K}$.

First we try Judgment A. The formulae in the form of $\mathbf{E}\,a$, where a is an abstract cell, that are satisfied at each node of $\mathcal{K}$ are as follows.

- At the left node, $\mathbf{E}\,(\neg\overline{\mathbf{X}}\,\mathbf{1} \wedge \neg\mathbf{1} \wedge \neg\mathbf{X}\,\mathbf{1} \wedge \mathbf{F}\,\mathbf{1})$, $\mathbf{E}\,(\neg\overline{\mathbf{X}}\,\mathbf{1} \wedge \neg\mathbf{1} \wedge \mathbf{X}\,\mathbf{1} \wedge \mathbf{F}\,\mathbf{1})$, and $\mathbf{E}\,(\neg\overline{\mathbf{X}}\,\mathbf{1} \wedge \neg\mathbf{1} \wedge \neg\mathbf{X}\,\mathbf{1} \wedge \neg\mathbf{F}\,\mathbf{1})$.
- At the center node, $\mathbf{E}\,(\neg\overline{\mathbf{X}}\,\mathbf{1} \wedge \mathbf{1} \wedge \neg\mathbf{X}\,\mathbf{1} \wedge \mathbf{F}\,\mathbf{1})$.
- At the right node, $\mathbf{E}\,(\overline{\mathbf{X}}\,\mathbf{1} \wedge \neg\mathbf{1} \wedge \neg\mathbf{X}\,\mathbf{1} \wedge \neg\mathbf{F}\,\mathbf{1})$, and $\mathbf{E}\,(\neg\overline{\mathbf{X}}\,\mathbf{1} \wedge \neg\mathbf{1} \wedge \neg\mathbf{X}\,\mathbf{1} \wedge \neg\mathbf{F}\,\mathbf{1})$.

Therefore the covering set obtained by Judgment A is {(FFFT), (FFTT), (FTFT), (TFFF), (FFFF)}.

Judgment B gives the same covering set as Judgment A. We omit details.

For Judgment C, we list the formulae in the form of $\mathbf{E}\,\varphi_i^a$ that are satisfied at each node of $\mathcal{K}$:

- At the left node, $\mathbf{E}\,(\neg\overline{\mathbf{X}}\,\mathbf{1})$, $\mathbf{E}\,(\neg\mathbf{1})$, $\mathbf{E}\,(\mathbf{X}\mathbf{1})$, $\mathbf{E}\,(\neg\mathbf{X}\mathbf{1})$, $\mathbf{E}\,(\mathbf{F}\,\mathbf{1})$, and $\mathbf{E}\,(\neg\mathbf{F}\,\mathbf{1})$.
- At the center node, $\mathbf{E}\,(\neg\overline{\mathbf{X}}\,\mathbf{1})$, $\mathbf{E}\,(\mathbf{1})$, $\mathbf{E}\,(\neg\mathbf{X}\,\mathbf{1})$, and $\mathbf{E}\,(\mathbf{F}\,\mathbf{1})$.
- At the right node, $\mathbf{E}\,(\overline{\mathbf{X}}\,\mathbf{1})$, $\mathbf{E}\,(\neg\overline{\mathbf{X}}\,\mathbf{1})$, $\mathbf{E}\,(\neg\mathbf{1})$, $\mathbf{E}\,(\neg\mathbf{X}\,\mathbf{1})$, and $\mathbf{E}\,(\neg\mathbf{F}\,\mathbf{1})$.

Thus the covering set obtained is {(FFFF), (FFFT), (FFTF), (FFTT), (FTFT), (TFFF)}. This set has an extra element (FFTF) compared with that in Judgment A. ∎

Lemma 6.4 Let C be a configuration. If Kripke structure $\mathcal{K}$ simulates $\mathcal{K}(C)$, the set A of abstract cells that are judged to be induced by concrete cells by Existence Judgment A covers C. The same holds for Judgments B and C.

Proof The second half is clear from the first half since Judgments B and C are weaker than Judgment A. To prove the first part, take a concrete cell c. We will show $\alpha(c, C) \in A$.

Let π be the unique path in $\mathcal{K}(C)$ such that $\pi(0) = c$. Let h be a simulation from $\mathcal{K}(C)$ to $\mathcal{K} = (S, R, \lambda)$ and take $s_0 \in S$ such that $(\pi(0), s_0) \in S$. Since h is a simulation we can take by induction on $i \in \omega$ $s_i \in S$ and $s_{-i} \in S$ such that $(\pi(i), s_i) \in h$, $(\pi(-i), s_{-i}) \in h$, $(s_i, s_{i+1}) \in R$, and $(s_{-(i+1)}, s_{-i}) \in R$. Let π' be a path in $\mathcal{K}$ defined by $\pi'(x) = s_x$ for $x \in \mathbf{Z}$. Then by induction on the construction of 2LTL formulae φ, we can show $\mathcal{K}(C), \pi \models \varphi \iff \mathcal{K}, \pi' \models \varphi$ for $x \in \mathbf{Z}$. The base case is guaranteed by the fact h is a simulation and each induction step is almost obvious.

Since $\mathcal{K}(C), c \models \mathbf{E}\,\alpha(c, C)$, we have $\mathcal{K}(c), \pi \models \alpha(c, C)$ and therefore $\mathcal{K}, \pi' \models \alpha(c, C)$, which means $\mathcal{K}, s_0 \models \mathbf{E}\,\alpha(c, C)$. Therefore Existence Judgment A concludes that $\alpha(c, C)$ is induced by a concrete cell and we are done. ∎

6.2.4 Constructing a Kripke Structure

In this subsection we describe methods to construct a finite Kripke structure $\mathcal{K}(A)$ from given set A of abstract cells. First we give methods to judge whether two abstract cells are adjacent.

Adjacency Judgment D Abstract cell a_1 is judged to be a left-hand neighbor of abstract cell a_2 if formula $a_1 \wedge \mathbf{X}\, a_2$ is satisfiable. ∎

Judgment D can be implemented using a 2LTL satisfiability checker.

Adjacency Judgment E This judgment can be applied when all formulae in Φ are one-way. Abstract cell a_1 is judged to be a left-hand neighbor of abstract cell a_2 if formulae $\overrightarrow{a_1} \wedge \mathbf{X}\,\overrightarrow{a_2}$ and $\mathbf{X}\,\overleftarrow{a_1} \wedge \overleftarrow{a_2}$ are satisfiable. ∎

Judgment E is weaker than Judgment D, that is, If Judgment D judges a_1 is a left-hand neighbor of a_2, so is Judgment E. However, Judgment E can be implemented using an LTL satisfiability checker rather than a 2LTL satisfiability checker.

Using an adjacency judgment method, we construct Kripke structure $\mathcal{K}(A) = (A, R, \lambda)$ from a set A of abstract cells as follows. We further assume throughout this chapter that formula $\mathbf{1}$ is contained in Φ. We denote by I the index of the formula $\mathbf{1}$ in Φ, that is $\varphi_I = \mathbf{1}$. The underlying set of $\mathcal{K}(A)$ is A. The transition relation $R(m)$ is defined so that $(a_1, a_2) \in R(m)$ if the method judges a_1 is a left-hand neighbor of a_2. The labeling function is defined as $\lambda(\mathbf{1}) = \{a = (b_1, \ldots, b_n) \in A \mid b_I = \mathrm{T}\}$.

Lemma 6.5 If a set A of abstract cells covers a configuration C, the Kripke structure $K(A)$ constructed above using Adjacency Judgment D simulates $\mathcal{K}(C)$. The same

holds for Adjacency Judgment E.

Proof The second half is clear from the first half since Judgment E is weaker than Judgment D.

Let $\mathcal{K}(C) = (C, R^{\mathcal{K}(C)}, \lambda^{\mathcal{K}(C)})$, $\mathcal{K}(A) = (A, R^{\mathcal{K}(A)}, \lambda^{\mathcal{K}(A)})$, and $h = \{(c, \alpha(c, C)) \mid c \in C\}$. Then $h \subseteq C \times A$ is a relation between C and A. We check that h is a simulation from C to A.

The first condition in Definition 2.5 is clearly satisfied. For the second condition, take $c_1, c_2 \in C$ such that c_1 is the left neighbor of c_2. Let $a_1 = \alpha(c_1, C)$ and $a_2 = \alpha(c_2, C)$. Since $\mathcal{K}(C), c_1 \models a_1$ and $\mathcal{K}(C), c_2 \models a_2$, $\mathcal{K}(C), c_1 \models a_1 \wedge \mathbf{X}\, a_2$, which means that the formula $a_1 \wedge \mathbf{X}\, a_2$ is satisfiable, therefore Judgment D judges a_1 is a left-hand neighbor of a_2, that is, $(a_1, a_2) \in R^{\mathcal{K}(A)}(m)$. The third condition can be checked similarly.

For the fourth condition, when we write $\alpha(c, C) = (b_1, \ldots, b_n)$, we have $c \in \lambda^{\mathcal{K}(C)}(\mathbf{1}) \iff b_I = \mathrm{T} \iff \alpha(c, C) \in \lambda^{\mathcal{K}(A)}(\mathbf{1})$. ∎

Example 6.6 Let us construct a Kripke structure from the set of abstract cells shown in the example of Section 6.2.3. For example we can judge if (FFFT) is a left neighbor of (FTFT) by checking whether formula

$$(\neg\overline{\mathbf{X}}\,\mathbf{1} \wedge \neg\mathbf{1} \wedge \neg\mathbf{X}\,\mathbf{1} \wedge \mathbf{F}\,\mathbf{1}) \quad \wedge \quad \mathbf{X}\,(\neg\overline{\mathbf{X}}\,\mathbf{1} \wedge \mathbf{1} \wedge \neg\mathbf{X}\,\mathbf{1} \wedge \mathbf{F}\,\mathbf{1})$$

is satisfiable. It is unsatisfiable since it implies two incompatible formulae $\mathbf{X}\,\mathbf{1}$ and $\neg\mathbf{X}\,\mathbf{1}$. By checking satisfiability for all pairs of the formulae, we have the Kripke structure illustrated below:

Recall that if we use Existence Judgment C, there is an extra abstract cell (FFTF). Since the cell itself is unsatisfiable, it becomes an isolated point in the resulting Kripke structure. ∎

Even if Kripke structure $\mathcal{K}$ simulates $\mathcal{K}(C)$, it is not necessarily the case that $\mathcal{K}$ coincides with the image of an simulation. A node s in $\mathcal{K}$ is called a *spurious node* if for any simulation h there is no concrete cell c such that $(c, s) \in h$. A transition $(s, s') \in R^{\mathcal{K}}$ is called a *spurious transition* if for any simulation h there is no adjacent concrete cells c and c' such that $(c, s) \in h$ and $(c', s') \in h$. The less spurious nodes and transitions exist in $\mathcal{K}$ the more accurately it approximates $\mathcal{K}(C)$. Let φ_0 be formula $\mathbf{G}\,\mathbf{X}\,\mathbf{true} \wedge \overline{\mathbf{G}\,\mathbf{X}}\,\mathbf{true}$. Since for any configuration C and concrete cell c $\mathcal{K}(C), c \models \mathbf{E}\,\varphi_0$

is satisfied, if node s is not spurious $\mathcal{K}, s \models \mathbf{E}\,\varphi_0$ also holds. Therefore if a node does not have an infinite path for both direction, such as an isolated point, it is spurious. A Kripke structure that approximates a configuration still approximates it if all spurious nodes are removed from the structure.

6.2.5 Kripke Structure for Next Generation

In this subsection we describe a method to construct the Kripke structure for the next generation from given Kripke structure $\mathcal{K}(A)$.

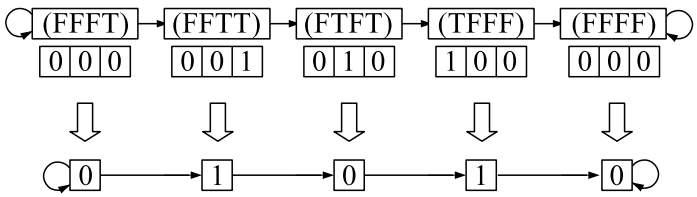
Construction of Kripke Structure for the Next Generation We assume that formulae $\overline{\mathbf{X}}\,\mathbf{1}$, $\mathbf{1}$, and $\mathbf{X}\,\mathbf{1}$ are included in Φ. Let $\mathcal{K}(A) = (A, R, \lambda)$ be given Kripke structure. The Kripke structure for the next generation $\mathcal{K}' = (A, R, \lambda)$ is defined as follows. The underlying set and the transition relation is the same as $\mathcal{K}(A)$. Let b^{-1}, b^0, b^1 be the value (T or F) for $\overline{\mathbf{X}}\,\mathbf{1}$, $\mathbf{1}$, and $\mathbf{X}\,\mathbf{1}$, respectively, of $a \in A$. Let $v(a)$ be the value (T or F) obtained by applying the rule of the cellular automaton to the triple (b^{-1}, b^0, b^1). The labeling function is defined as $\lambda(\mathbf{1}) = \{a \in A \mid v(a) = \mathrm{T}\}$. ∎

Lemma 6.7 If Kripke structure $\mathcal{K}(A)$ of abstract cells simulates configuration C, the Kripke structure $\mathcal{K}'$ constructed by the above method simulates the configuration C' in the next generation.

Proof Let h be a simulation from $\mathcal{K}(C)$ to $\mathcal{K}(A)$. We show that h is also a simulation from $\mathcal{K}(C')$ to K' by identifying corresponding concrete cells in C and C'.

The first three conditions in Definition 2.5 are trivially satisfied since we identify C and C' and the transition relation of $\mathcal{K}(A)$ is same as that of $\mathcal{K}'$. For the fourth condition, take a concrete cell c and abstract cell a such that $(c, a) \in h$. We have $\mathcal{K}(A), a \models \mathbf{E}\,\alpha(c, C)$ as in the proof of Lemma 6.4. As three formulae $\overline{\mathbf{X}}\,\mathbf{1}$, $\mathbf{1}$, and $\mathbf{X}\,\mathbf{1}$ are included in Φ, the status $\{\mathrm{T}, \mathrm{F}\}$ of c in C' is calculated with the same rule as in the calculation of the labeling function of $\mathcal{K}'$. Therefore $c \in \lambda^{\mathcal{K}(C')}(\mathbf{1}) \iff a \in \lambda^{\mathcal{K}'}(\mathbf{1})$. ∎

Example 6.8 The next generation Kripke structure for the Kripke structure of the example in the previous subsection can be calculated as illustrated below:



Thus the Kripke structure for Generation 1 is calculated by applying Existence Judgment, Adjacent Judgment and Construction of Kripke Structure to the Kripke structure for Generation 0. Structures for Generation 2 and after can be calculated in a similar way. ∎

6.2.6 Analysis

The overall analyzing method, which combines the elements described so far, is as follows. We fix an existence judgment, and an adjacent judgment; and for given initial configuration C_0, choose a finite set Φ of 2LTL formulae and a finite total Kripke structure $\mathcal{K}_0$ which simulates $\mathcal{K}(C_0)$. Then starting with given initial configuration C_0, we construct for $i = 0, 1, \ldots$:

(1) A set of abstract cells A_i from $\mathcal{K}_i$ and Φ using the existence judgment.

(2) A Kripke structure $\mathcal{K}(A_i)$ that simulates $\mathcal{K}_i$ using the adjacent judgment.

(3) A Kripke structure $\mathcal{K}_{i+1}$ using the construction of Kripke structure for the next generation.

The method terminates after finitely many steps: because candidates for A_i and $\mathcal{K}_i$ are finite since the set of all abstract cells is finite, therefore there is i and j with $i < j$ such that $\mathcal{K}_i = \mathcal{K}_j$. The following theorem describes the basic property of the structures thus constructed.

Theorem 6.9 For any i, $\mathcal{K}_i$ simulates $\mathcal{K}(C_j)$.

Proof Induction on i. The case $i = 0$ is clear. The case $i + 1$ is: A_i covers C_i by Lemma 6.4. Then $\mathcal{K}(A_i)$ simulates $\mathcal{K}(C_i)$ by Lemma 6.5. Finally $\mathcal{K}(C_i)$ simulates C_i by Lemma 6.7. ∎

Theorem 6.9 guarantees that "patterns" which do not appear in the sequence of simulated Kripke structures $\mathcal{K}_{ii}$ do not appear in the concrete configurations either. Since there are finite number of structures this can be checked by observing all the structure. Refer Section 6.3 for examples of such analyses.

The accuracy of the resulted sequence of Kripke structures depends on the choice of the judgment methods, the set Φ of 2LTL formulae, and the initial Kripke structure $\mathcal{K}_0$. A weaker judgment method may lose the accuracy. More appropriate formulae in Φ increase the accuracy, they may require, however, more computation time and the resulting structures may have complex ones with large number of nodes.

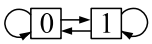
6.3 Experimental Results

We implemented a tool for the analysis described in the previous section for Existence Judgment B and Adjacent Judgment E. Those judgment methods require LTL model checking and LTL satisfiability judgment, and we use Maude [16] for them. The tool is invoked with the rule number, the initial finite Kripke structure $\mathcal{K}_0$, and the set Φ of one-way 2LTL formulae. Three formulae $\mathbf{1}$, $\mathbf{X}\,\mathbf{1}$, and $\overline{\mathbf{X}}\,\mathbf{1}$ are always regarded as elements of Φ. The tool reports the resulted Kripke structure $\mathcal{K}_i$ as its output.

The following examples are some of the results of our experimental analysis. In all of the examples the computation time of the tool is a few seconds. The longest one is rule 1 and it takes 6.4 seconds for a PC with Pentium M CPU 1.2 GHz, 512MB memory, and Windows XP. We denote Φ_1 by $\Phi \setminus \{\mathbf{1}, \mathbf{X}\,\mathbf{1}, \overline{\mathbf{X}}\,\mathbf{1}\}$.

Example 6.10 Rule 178. We set $\Phi_1 = \{\mathbf{F}\,\mathbf{1}\}$ and $\mathcal{K}_0$ as in the top-right corner in Figure 6.2, and the tool generates a sequence of Kripke structure shown in the figure. Note that the structures for Generations 2 and 3 are identical. We can for example conclude that one of the statuses of any two adjacent cells in configurations C_i must be zero, since we do not see pattern $\boxed{1}\to\boxed{1}$ in the approximation structures. If we set $\Phi_1 = \varnothing$, then a Kripke structure with the above pattern does appear in the approximation sequence, though we can still conclude that one of the statuses of any *three* adjacent cells must be zero, since pattern $\boxed{1}\to\boxed{1}\to\boxed{1}$ does not appear. ∎

Example 6.11 Rule 0. We set $\phi_1 = \varnothing$ and use the following Kripke structure as $\mathcal{K}_0$:



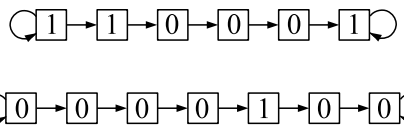
Note that the structure $\mathcal{K}_0$ simulates *any* Kripke structure $\mathcal{K} = (S, R, \lambda)$ since $h = \{(s, s_0) \mid s \in S \setminus \lambda(\mathbf{1})\} \cup \{(s, s_1) \mid s \in \lambda(\mathbf{1})\}$ is a simulation from $\mathcal{K}$ to $\mathcal{K}_0$ where s_0 and s_1 are the left and right element in the figure, respectively.

The tool generates the following Kripke structure as the approximation structure for Generation 1.

We conclude that all status of any configuration for and after Generation 1 must be zero regardless of the initial configuration. ∎

Example 6.12 Rule 1. Take $\Phi_1 = \{\mathbf{F}\,\mathbf{1}, \mathbf{G}\,\mathbf{1}, \mathbf{X}\,\mathbf{X}\,\mathbf{1}\}$ and the same $\mathcal{K}_0$ as in Figure 6.2.

The sequence that the tool generates ends in a loop which consists of the following two Kripke structures:



What we can say from the result is that two types of configurations appear alternatively, one has all but three adjacent cells with status one, in the other all cells except for exactly one cell have status one. ∎

6.4 Related Work

Cellular automata are proposed by von Neumann [75] for studying self-copying machines. Although the original cellular automata were two-dimensional, variants have been proposed in various researches. One-dimensional cellular automata were studied extensively by Wolfram [77]. For example he conjectured that the one-dimensional cellular automata with rule 110 are capable of universal computation, and it was later proved by Cook [17].

Since cellular automata can be regarded as a graph by regarding the adjacent cells are connected with edges, techniques of abstraction with modal logics have been applied to the analysis of cellular automata. The method proposed by Hagiya *et al.* in [33] falls into this category. The study in this chapter is an improvement of the method. In the rest of this section we compare our work with [33].

The basic idea is common; abstract cells are defined by using modal logic formulae and finite graphs constructed with abstract cells are regarded as approximations of concrete configurations.

One of the major differences is the logic. The existing study uses 2CTL and when analyzing one-dimensional automata, an axiom $\mathbf{AX}\varphi \leftrightarrow \mathbf{EX}\varphi$ is added. Our method does not need such an axiom since we use 2LTL or its restriction. The other difference is that the existing study only uses satisfiability judgment, while ours uses both satisfiability judgment and model checking.

An advantage of our method is its simplicity. First, in the existing method, when there are multiple candidates for the next generation of an abstract cell, the complete graph of all the candidates are generated and it is inserted into the Kripke structure as its substructure. This operation is called 'split'. Although inconsistent cells are subsequently removed, a same abstract cell may appear as the next generation of different abstract cells. In such cases they are identified. This operation is called 'merge'. In our method we calculate the set of abstraction cells as the first step, and

the Kripke structure is built in the subsequent step. That avoids the operations split and merge. Second, in the existing method, inconsistent adjacent relations are found by judging the satisfiability of formulae in the form of $\varphi \rightarrow \mathbf{EX}\psi$. However, this type of judgment does not always successfully rule out the inconsistency when a formula related reachability (such as $\mathbf{EF}\,\varphi$) is included. To avoid the problem, it is required to judge the satisfiability of formulae in the form of $\varphi \rightarrow \mathbf{EF}\,\psi$ as well. In our method, we do not need the process since we model check the Kripke structure to rule out such inconsistency. Because of the simpleness, we are able to implement a system to analyze the system without difficulties.

In exchange for the simplicity our method loses some area for applications. For example existing study proposes a direction for extending its method for the analysis of two-dimensional cellular automata. Our method is not capable of analyzing them since LTL (including 2LTL) is specialized for the linear structure.

Chapter 7

Conclusion

7.1 Summary

In the first half of the thesis implementable decision procedures for modal logics were studied. We considered four extensions for the minimal modal logic **K**; these extensions play important roles when we apply modal logics to shape analysis. The extensions were (a) alternation-free fixed-point operators, (b) nominals, (c) reverse modalities, and (d) functional Kripke structures. We presented decision procedures for the logics that had the combinations (a)+(c)+(d), (a)+(b)+(d), and (a)+(c)+(d) and proved their correctness.

In the second half of the thesis, we applied the decision procedures to verification problems based on the predicate abstraction technique. The first application was for shape analysis. We defined a small programming language to manipulate pointers, provided a method to calculate the weakest preconditions for each basic statement of the language, and built two systems to verify the properties of programs written in the language. The decision procedures in the first half of the thesis were used in the system. One of the systems was for automatic verification and the other was for manual verification. We showed the safety properties of the DSW marking algorithms by using the latter system. The second application was for the analysis of one-dimensional cellular automata on the basis of similar techniques.

7.2 Future Work

7.2.1 Decision Procedures

The first direction in future work is to strengthen our decision procedures for application to more powerful logics. There is a decidable sublogic of the first-order

predicate logic called Loosely Guarded Fragment (LGF), and the extension of the LGF with fixed-point operators is called μ-LGF, which can be regarded as a natural extension of the two-way modal μ-calculus. Although a general decision procedure is already known for μ-LGF [31], it is a complex procedure involving the use of alternating tree automata as in the full modal μ-calculus. Therefore, it would be meaningful to find an appropriate subsystem of μ-LGF to which our procedure can be applied. We successfully adapted our procedures to the LGF (without fixed-point operators) and built an experimental implementation [58]; however, fixed-point operators must be included for applications. An apparent attempt would be to define an alternation-free μ-LGF in which we do not permit the fixed operators to alternate. However, a simple adaptation of our procedure to the logic is not effective, and further investigations are required.

The second direction is the improvement of the decision procedures in terms of computation time. Although our implementations show sufficient performance to verify the properties described in the thesis, a more powerful procedure is always desirable, for example, in order to make the DSW properties automatically verifiable. The current procedure first creates all possible nodes in the tableau and removes inconsistent nodes. If we did not use the BDDs, this would be a critical overhead as compared with the approach in which only necessary nodes are created at the first stage, such as the well-known decision procedure for (one-way) CTL [27]. Although we reduce the overhead by using BDDs in the current implementation, it would be better if we could completely avoid the overhead. One of the ideas we would like to pursue for achieving this is to the utilization of Boolean SAT solvers [25, 44]. It does not seem possible to introduce a Boolean SAT solver simply to implement our decision procedures, since there is no apparent method to estimate the number of nodes. However, it may be possible to incorporate search techniques for Boolean SAT solvers that have been developed in recent years that dramatically improve their performance.

7.2.2 Applications

The first point is the automatization in the MLAT system described in Chapter 4. Although the system automatically creates the abstract transition system, the user must manually perform some work with regard to the following functions:

(1) Inspection of the counterexample: When the model checker produces a counterexample, the user needs to ascertain whether it is a real counterexample or not. The user has to extract the execution path from the counterexample and the

truth values for the abstraction predicates at each time and to ascertain whether corresponding transitions are possible in the concrete transition system.

(2) Choice of the predicates: The user has to define the set of predicates before starting verification and after ascertaining that the counterexample is not a true one by model checking.

When applying the predicate abstraction technique, a method called counterexample guided abstraction refinement [14] is widely used in order to automate the two abovementioned functions. Starting from the endpoint of the given counterexample, this method follows it reversely, calculating the weakest preconditions. If the condition becomes false, it is ascertained to be a false counterexample. In such a case, predicates can be generated to remove the counterexample. However, by applying this technique to our method, the loop of the counterexample analysis and the predicate generation does not terminate in most cases. We have to investigate further to seek a method to avoid the infinite loop, for example, by using an appropriate approximation.

The second issue is the performance improvement of the system. As the bottleneck for the performance is the decision procedure, the improvement described in the previous subsection is helpful. Moreover, a reduction in the size of the abstract transition system is also required. In the current framework, all the predicates specified by the user are used at every position of the source code. An apparent improvement can be achieved by introducing a feature to restrict the usage of each predicate to certain positions in the source. However, this does not help in the case of the DSW algorithm since some predicates are needed only at some states in the abstract transition system among those states that correspond to the same position of the source code. Further research is required to handle these cases.

For the system discussed in Chapter 5, we plan to integrate it with the interactive proof editor Agda [1]. Since the approach discussed in the chapter requires user interaction, it is desirable to have a system in which the user completes the verification. The integrated system will have a library for Agda with which the user will build proofs of the properties to be verified in Hoare logic, and the decision procedure is called from Agda to validate Hoare triples corresponding to atomic programs. A prototype version based on a less powerful logic (two-way CTL) has been built [79], and we will enhance it for the logics discussed in the thesis.

The final point concerns liveness properties. The predicate abstraction technique is suitable for safety properties but not always for liveness properties. Alternative techniques for liveness properties, such as transition predicate abstraction [52, 29], are

emerging. One of the issues in the application of such techniques to shape analysis is the manner of defining and evaluating appropriate well-founded relations. Let us confine ourselves to the most simple liveness property — termination of a loop. An idea we would like to develop is the utilization of min-plus algebra [2]. By extending the definition of the Kripke structure so that its truth values range in min-plus algebra $\mathbf{N}_\infty$ rather than $\{\mathbf{true}, \mathbf{false}\}$, a formula in the modal logic defines a value in $\mathbf{N}_\infty$ according to the heap status. If we show that the value after the execution of the loop body is strictly less than that before the execution, we can conclude that the loop must terminate. We observe that the relation "is strictly less than" can also be expressed in a modal logic (under the $\mathbf{N}_\infty$-valued semantics). Therefore, what we need is a decision procedure for ascertaining the satisfiability in the $\mathbf{N}_\infty$-valued semantics (i.e., having the value zero). We have started investigations by searching for model checking algorithms under this semantics and have obtained some results [38].

References

[1] Agda. `http://unit.aist.go.jp/cvs/Agda/`.

[2] Francois Louis Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and Linearity : An Algebra for Discrete Event Systems*. Wiley, 1992.

[3] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis by predicate abstraction. In Radhia Cousot, editor, *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pages 164–180. Springer, 2005.

[4] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122. Springer-Verlag New York, Inc., 2001.

[5] Patrick Blackburn. Nominal tense logic. *Notre Dame Journal of Formal Logic*, 34:56–83, 1993.

[6] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, 2001.

[7] Piero A. Bonatti, Carsten Lutz, Aniello Murano, and Moshe Y. Vardi. The complexity of enriched μ-calculi. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP (2)*, volume 4052 of *Lecture Notes in Computer Science*, pages 540–551. Springer, 2006.

[8] Piero A. Bonatti and A. Peron. On the undecidability of logics with converse, nominals, recursion and counting. *Artificial Intelligence*, 158:75–96, 2004.

[9] Richard Bornat. Proving pointer programs in Hoare logic. In Roland Carl Backhouse and José Nuno Oliveira, editors, *MPC*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer, 2000.

[10] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[11] BuDDy. `http://sourceforge.net/projects/buddy`.

[12] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.

[13] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), 1992.

[14] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169, London, UK, 2000. Springer-Verlag.

[15] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[16] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[17] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1):1–40, 2004.

[18] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[19] Dennis Dams and Kedar S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI03)*, pages 310–324, 2003.

[20] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Nicolas Halbwachs and Doron Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 1999.

[21] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[22] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[23] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.

[24] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of International Conference on Software Engineering, ICSE 2001*, 2001.

[25] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[26] E. A. Emerson and E. M. Clarke. Using branching-time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[27] E. Allen Emerson. Temporal and modal logic. In *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 995–1072. Elsevier Science Publishers B.V., 1990.

[28] E. Allen Emerson and Charanjit S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *FOCS*, pages 368–377. IEEE, 1991.

[29] Carl Christian Frederiksen and Masami Hagiya. Sub-computation based transition predicate abstraction. *IPSJ Transactions on Programming*, 48, SIG10 (PRO33):114–137, 2007.

[30] Pierre Genevès and Nabil Layaïda. Comparing XML path expressions. In *DocEng '06: Proceedings of the 2006 ACM symposium on Document engineering*, pages 65–74, New York, NY, USA, 2006. ACM.

[31] Erich Grädel. Guarded fixed point logics and the monadic theory of countable trees. *Theoretical Computer Science*, 288:129–152, 2002.

[32] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[33] Masami Hagiya, Koichi Takahashi, Mitsuharu Yamamoto, and Takahiro Sato. Analysis of synchronous and asynchronous cellular automata using abstraction by temporal logic. In *FLOPS2004: The Seventh Functional and Logic Programming Symposium*, volume 2998 of *Lecture Notes in Computer Science*, pages 7–21, 2004.

[34] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110. Springer-Verlag, 1995.

[35] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, pages pp. 58–70. ACM Press, 2002.

[36] Thierry Hubert and Claude Marche. A case study of C source code verification: the Schorr-Waite algorithm. In *3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'05), Koblenz, Germany*. IEEE Comp. Soc. Press, September 2005.

[37] Thierry Hubert and Claude Marché. A case study of C source code verification: the Schorr-Waite algorithm. In Bernhard K. Aichernig and Bernhard Beckert, editors, *SEFM*, pages 190–199. IEEE Computer Society, 2005.

[38] Dai Ikarashi, Yoshinori Tanabe, Koki Nishizawa, and Masami Hagiya. The modal *mu*-calculus on min-plus algebra $\mathbf{N}_\infty$ and its application (in Japanese). In *24th Conference of Japan Society for Software Science and Technology*, 2007.

[39] JavaBDD. `http://javabdd.sourceforge.net/`.

[40] Yoshiki Kinoshita and Koki Nishizawa. An algebraic semantics of predicate abstraction for PML. In *24th Conference of Japan Society for Software Science and Technology*, 2007.

[41] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[42] Ralf Küsters. Memoryless determinacy of parity games. In Grädel et al. [32], pages 95–106.

[43] Alexey Loginov, Thomas W. Reps, and Mooly Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In Kwangkeun Yi, editor, *SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 261–279. Springer, 2006.

[44] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient SAT solver. In Holger H. Hoos and David G. Mitchell, editors, *SAT (Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 360–375. Springer, 2004.

[45] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, 1984.

[46] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[47] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In Franz Baader, editor, *CADE*, volume 2741 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2003.

[48] Andrzej Wlodzimierz Mostowski. Regular expressions for infinite trees and a standard form of automata. In *5th Symposium on Computation Theory*, volume 208 of *Lecture Notes in Computer Science*, pages 157–168. Springer-Verlag, 1984.

[49] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[50] Guoqiang Pan, Ulrike Sattler, and Moshe Y. Vardi. BDD-based decision procedures for K. In *Proceedings of the 18th International Conference on Automated Deduction*, pages 16–30. Springer-Verlag, 2002.

[51] Guoqiang Pan and Moshe Y. Vardi. Optimizing a BDD-based modal solvar. In *19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Computer Science*, pages 75–89, 2003.

[52] Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[53] John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[54] Alexandre Riazanov and Andrei Voronkov. Vampire 1.1 (system description). In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *IJCAR*, volume 2083 of *Lecture Notes in Computer Science*, pages 376–380. Springer, 2001.

[55] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer Verlag, 1997.

[56] Shumuel Safra. On the complexity of omega-automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science, FoCS '88*, pages 319–327. IEEE Computer Society Press, 1988.

[57] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[58] Takahiro Sato, Yoshinori Tanabe, and Masami Hagiya. Decision procedures for guarded fragments using BDD (in Japanese). In *21th Conference of Japan Society for Software Science and Technology*, 2004.

[59] Ulrike Sattler and Moshe Y. Vardi. The hybrid μ-calculus. In *Proceedings of the International Joint Conference on Automated Reasoning*, volume 2083 of *LNCS*, pages 76–91. Springer Verlag, 2001.

[60] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.

[61] Toshifusa Sekizawa, Toshinori Takai, Yoshinori Tanabe, and Koichi Takahashi. A method to generate formulas for temporal logic satisfiability checkers. *Electronics and Communications in Japan, Part II*, 90(11):98–108, 2007.

[62] Toshifusa Sekizawa, Yoshinori Tanabe, Yoshifumi Yuasa, and Koichi Takahashi. Mlat: A tool for heap analysis based on predicate abstraction by modal logic. In *The IASTED International Conference on Software Engineering (SE 2008)*, 2008. (to appear).

[63] Koichi Takahashi and Masami Hagiya. Abstraction of graph transformation using temporal formulas. In *Supplemental Volume of the 2003 International Conference on Dependable Systems and Networks (DNS-2003)*, pages W–65 to W–66, 2003.

[64] Koichi Takahashi, Yoshinori Tanabe, and Toshifusa Sekizawa. Finite approximation analysis of one dimensional cellular automata (in Japanese). *Computer Software*, 23(3):147–157, 2006.

[65] Yoshinori Tanabe, Toshifusa Sekizawa, Yoshifumi Yuasa, and Koichi Takahashi. A heap verification method using a modal logic (in Japanese). In *3rd Dependable Software Workshop (DSW'06)*, pages 39–50, 2006.

[66] Yoshinori Tanabe, Koichi Takahashi, Mitsuharu Yamamoto, Takahiro Sato, and Masami Hagiya. An implementation of a decision procedure for satisfiability of two-way CTL formulae using BDD (in Japanese). *Computer Software*, 22(3):154–166, 2005.

[67] Yoshinori Tanabe, Koichi Takahashi, Mitsuharu Yamamoto, Akihiko Tozawa, and Masami Hagiya. A decision procedure for the alternation-free two-way modal μ-calculus. In *TABLEAUX 2005*, volume 3702 of *Lecture Notes in Artificial Intelligence*, pages 277–291, 2005.

[68] Yoshinori Tanabe, Toshinori Takai, Toshifusa Sekizawa, and Koichi Takahashi. Preconditions of properties described in ctl for statements manipulating pointers. In *Supplemental Volume of the 2005 International Conference on Dependable Systems and Networks (DSN-2005)*, pages 228–234, 2005.

[69] Yoshinori Tanabe, Toshinori Takai, and Koichi Takahashi. Verification tools using abstraction (in Japanese). *Computer Software*, 22(1):2–44, 2005.

[70] Stephan Tobies. The complexity of reasoning with cardinality restrictions and nominals in expressive description logics. *J. Artif. Intell. Res. (JAIR)*, 12:199–217, 2000.

[71] Akihiko Tozawa. On binary tree logic for XML and its satisifiability test. In *Sixth Workshop on Programming and Programming Language (PPL2004)*, 2004.

[72] Jan van Eijck. HyLoTab — tableau-based theorem proving for hybrid logics, 2002. Manuscript, CWI, available from `http://www.cwi.nl/~jve/hylotab`.

[73] Moshe Y. Vardi. Why is modal logic so robustly decidable? In Neil Immerman and Phokion G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 149–184. American Mathematical Society, 1996.

[74] Moshe Y. Vardi. Reasoning about the past with two-way automata. In *Automata, Languages and Programming, 25th International Colloquium, ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 628–641, 1998.

[75] John von Neumann. *Theory of Self-Reproducing Automata*. Univ. of Illinois Press, 1966.

[76] Igor Walukiewicz. Completeness of Kozens axiomatisation of the propositional mu-calculus. *Information and Computation*, 157:142–182, 2000.

[77] Stephen Wolfram. *Cellular Automata and Complexity*. Addison-Wesley, 1994.

[78] Hongseok Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *Proceedings of the 1st Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, January 2001.

[79] Yoshifumi Yuasa, Makoto Takeyama, Toshifusa Sekizawa, Yoshinori Tanabe, and Koichi Takahashi. On verification of programs manipulating pointers by coordination of an automatic prover and a theorem proving supporting system (in Japanese). In *23rd Conference of Japan Society for Software Science and Technology*, 2006.

[80] Júlia Zappe. Modal μ-calculus and alternating tree automata. In Grädel et al. [32], pages 171–184.