NAIST-IS-DT9961204

Doctor's Thesis

Security Verification of Programs with Stack Inspection

Naoya Nitta

February 5, 2002

Department of Information Processing
Graduate School of Information Science
Nara Institute of Science and Technology

Doctor's Thesis
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
DOCTOR of ENGINEERING

Naoya Nitta

Thesis committee: Hiroyuki Seki, Professor
Katsumasa Watanabe, Professor
Minoru Ito, Professor

Security Verification of Programs with Stack Inspection*

Naoya Nitta

Abstract

Recently, with rapidly growth of open network environment, a well-defined access control mechanism becomes necessary. Java development kit 1.2 provides a runtime access control mechanism which inspects a control stack to examine whether the program has appropriate access permissions. Jensen et al. introduced a verification problem of deciding for a given program P with stack inspection and a given security property ψ written in a temporal logic formula, whether every reachable state of P satisfies ψ. They showed that the problem is decidable for the class of programs which do not contain mutual recursion. In this thesis, we show that the set of state sequences of a program is always an indexed language and consequently the verification problem is decidable. Our result is stronger than Jensen's in that a security property can be specified by a regular language, whose expressive power is stronger than temporal logic, and in that a program can contain mutual recursion. We also investigated the computational complexity of the problem. Since the result implies the problem is computationally intractable in general, we introduce a practically important subclass of programs which exactly model programs containing stack inspection of Java development kit 1.2. We present an algorithm which can solve the problem for this subclass in linear time in the size of a program. Furthermore, we implemented a verification system based on the proposed algorithm. Experimental results suggest that the proposed algorithm can be efficiently executed for real-world programs.

Keywords:

access control, security verification, stack inspection, Java, indexed language

*Doctor's Thesis, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DT9961204, February 5, 2002.

スタック検査機能を持つプログラムに対するセキュリティ検証*

新田 直也

内容梗概

近年コンピュータネットワーク環境の急速な発展に伴い, 適切なアクセス制御法の確立が, ますます必要となってきている. Java development kit 1.2 は, プログラム実行時に制御スタックを検査することでアクセス制御を行うプログラム環境である. Jensen らは, このようなスタック検査機能を持つプログラム P および時相論理式を用いて記述された検証条件 ψ を与えたときに, P の到達可能な状態全てが ψ を満たすかどうかを決定する問題として検証問題を定義し, 相互再帰を含まないプログラムのクラスに対して検証問題が決定可能となることを示した.

本論文では, 時相論理式よりも真に表現能力の大きい正規言語を用いて検証問題を定義する. そして, プログラムの実行系列の集合がインデックス言語となることを示し, その系としてプログラムが相互再帰を含む場合も含めて検証問題が一般に決定可能となることを示す.

また検証問題の計算複雑さについて解析を行ない, 一般に検証問題が計算量的に手におえない問題のクラスに属することを示す.

現実的な計算時間で検証問題を解くには, 問題のクラスを制限する必要がある. そこで, Java development kit 1.2 のスタック検査機構をモデル化したプログラムからなる部分クラスを考え, そのクラスに対して検証問題をプログラムサイズの線形時間で解く効率のよい検証アルゴリズムを提案する. さらに, 同アルゴリズムを実装した検証システムについて実験を行ない, その結果を基に本検証法の実用性について議論する.

キーワード

アクセス制御, セキュリティ検証, スタック検査, Java, インデックス言語

*奈良先端科学技術大学院大学 情報科学研究科 情報処理学専攻 博士論文, NAIST-IS-DT9961204, 2002 年 2 月 5 日.

List of Publications

1 Publications Related to the Thesis

1.1 Journal Papers

(1) N. Nitta, Y. Takata and H. Seki: Decidability of the Security Verification Problem for Programs with Stack Inspection, IEICE Transactions on Information and Systems, to appear (in Japanese).

(2) N. Nitta, Y. Takata and H. Seki: An Efficient Security Verification Method for Programs with Stack Inspection, JSSST Computer Software, to appear.

1.2 International Conferences (Reviewed)

(3) N. Nitta, Y. Takata and H. Seki: Security Verification of Programs with Stack Inspection, Proceedings of 6th ACM Symp. on Access Control Models and Technologies (ACM SACMAT 2001), pp.31–40, Chantilly, Virginia, May 2001.

(4) N. Nitta, Y. Takata and H. Seki: An Efficient Security Verification Method for Programs with Stack Inspection, Proceedings of the 8th ACM Conference on Computer and Communication Security (ACM CCS-8), pp.68–77, Philadelphia, Pennsylvania, Nov 2001.

1.3 Workshops

(5) S. Ikada, N. Nitta, Y. Takata and H. Seki: A Security Verification Method for Programs with Stack Inspection, Technical Report of IEICE, ISEC2000-78, Sep 2000 (in Japanese).

(6) N. Nitta, Y. Takata and H. Seki: Complexity of the Security Verification Problem for Programs with Stack Inspection, 3rd JSSST Workshop on Programming and Programming Languages (PPL2001), pp.53–60, Mar 2001.

(7) N. Nitta, Y. Takata and H. Seki: An Efficient Security Verification Method for Programs with Stack Inspection, Technical Report of IEICE, SS2001-7, pp.9–16, May 2001.

1.4 Technical Report

(8) N. Nitta, S. Ikada, Y. Takata and H. Seki: Decidability and Complexity of the Security Verification Problem for Programs with Stack Inspection, Technical Report NAIST-IS-TR2001003, Nara Institute of Science and Technology, 2001.

2 Other Publications

2.1 Workshops

(9) N. Nitta and H. Seki: Dependence Logic: A Logic for Software Design Modification, Technical Report of IEICE, SS99-1, May 1999 (in Japanese).

(10) N. Nitta and H. Seki: Dependence Logic and Its Application to Database Design Modification, 16th JSSST Conference Proceedings, D1-2, pp.37–40, Sep 1999 (in Japanese).

Acknowledgements

First, and foremost, I would like to thank Professor Hiroyuki Seki for his continuous support and encouragement of the work. He suggested an idea of this research in early discussion, and he helped through the research.

I would like to thank to Professor Katumasa Watanabe for providing me with beneficial comments to improve this research.

I am grateful to Professor Minoru Ito for his valuable suggestion in this research.

I would like to express my sincere gratitude to Associate Professor Yuichi Kaji for his support and advice throughout the research.

I wish to thank Assistant Professor Yoshiaki Takata for his help of the work. Most of the analyses of the computational complexity are based on his idea.

I sincerely thank Associate Professor Seiji Hamaguchi of Osaka university for the explanation of the relation between regular languages and temporal logic.

I also thank Assistant Professor Yasunori Ishihara of Osaka university for his valuable comments.

Finally, I wish to express my gratitude to all members of Seki Laboratory for discussions and help.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As a world wide computer network grows rapidly, providing a well-defined access control mechanism for network application systems becomes more important. For example, consider an electronic commerce application. Since a number of anonymous external mobile processes as well as local ones can be executed in a user's site, an appropriate access control is needed to prevent a malicious external process from accessing secret local resources. Java^{TM} sandbox model provides a security protection mechanism for such a distributed computational environment. However, the sandbox model lacks flexibility since it imposes too strong restriction on the behavior of an external process which may access local resources.

For this reason, Java development kit 1.2 (JDK1.2) provides a simple but sufficiently practical access control technology, *stack inspection* [13]. In the JDK1.2 environment, stack inspection mechanism can be incorporated into a user's system by placing invocations of check method *checkPermission* in the system code and by defining a security policy. In a security policy, every method belongs to one of the protection domains, and each protection domain is granted several *permissions*. If a method m belongs to a protection domain d and d is granted a permission p, then we simply say m has permission p. If *checkPermission*(p) with a permission p as an actual argument is invoked from a method m, then JDK1.2 examines not only whether the method m has p but also whether every ancestor method which directly or indirectly invokes m have p. If all those methods have p, then the execution continues. Otherwise, the execution is aborted. As is the case with other programming languages, the Java execution environment has a runtime control stack (or simply, stack), which consists of frames

for the active method and its ancestor methods. A frame for a method *m* contains the protection domain which *m* belongs to as well as actual arguments, local variables and the return address for *m*. The stack is inspected by *checkPermission*(p) from the top (the active method) to the bottom to examine whether the above mentioned condition is met; if a method which does not have p is encountered, then the execution is aborted. If the stack bottom or a method with a particular mode (called *privileged*) is encountered, then *checkPermission*(p) terminates successfully and returns to the active method.

Appropriate invocations of check method should be carefully placed in the local methods which directly or indirectly access secret local resources. Consider the following simple example. Let *write* be a local method which directly updates a customer's bank account and *credit* be a method which is called by a customer's method and updates the customer's bank account by calling *write* method. Suppose the following situation. Both *write* and *credit* methods have write permission p_{write} and every method of a valid customer has permission $p_{customer}$. The system manager places *checkPermission*(p_{write}) at the beginning of *write* method. The system manager makes *credit* method privileged, but there is no check statement in *credit* method. If a malicious user's method which does not have $p_{customer}$ calls *credit*, then *credit* successively calls *write*. Since *credit* is privileged, *checkPermission*(p_{write}) in *write* method succeeds and an illegal update may occur. Let ψ be the property that 'if the control reaches *write* method, then the control has passed through only methods which have $p_{customer}$ or p_{write},' which the system is expected to satisfy. Let us call such a property ψ as a global security property. The above mentioned execution does not satisfy ψ. In this particular example, if *checkPermission*($p_{customer}$) is placed in *credit* method, then every execution satisfies ψ. However, ensuring that a program satisfies such a global security property by hand becomes difficult when the whole program is large and complicated. Therefore, an automatic verification method is needed which verifies that every execution of P satisfies ψ for a given program P and a global security property ψ.

In [18], the pioneering paper in the formal verification of a program with stack inspection, the verification problem is defined as follows:

- A program is modeled as a directed graph called a flow graph. Since a flow graph does not have a data part, the contents of a control stack can be represented as a sequence of nodes, each of which is a program point where a method invocation

has occurred. A trace is a finite sequence $s_0, s_1, \ldots, s_k$ of stacks (also called states), where s_0 is the initial state and $s_{i+1} (0 \leq i < k)$ is a state reachable from s_i by a unit step execution.

- A local security check statement has the form of *check*(ϕ) where ϕ is an LTL (linear temporal logic) formula [7]. The execution of *check*(ϕ) at a state s succeeds if and only if s, interpreted as a Kripke structure, satisfies ϕ. A global security property ψ to be verified is also represented by an LTL formula.

- The verification problem for a given program (a flow graph) P and a global security property ψ is to decide whether every state in every trace in P (every reachable state of P) satisfies ψ.

Based on this formulation, a verification method is presented in [18] by using model checking [7] of LTL formulas. In [18], it is also shown that if a given program does not contain mutual recursion, then the verification problem is decidable.

In this thesis, we define a program model and a verification problem using regular languages instead of LTL formulas for specifying both a local check statement and a global security property. Since the class of regular languages is known to properly include the class of languages represented by LTL formulas [10], this formulation is an extension of the one in [18]. Furthermore, we show that the verification problem is decidable in general even if a program *does* contain mutual recursion, which is a proper improvement of the result of [18]. The outline of the proof of the decidability is as follows:

(i) For every program P, the set of traces in P is shown to be an indexed language. An indexed language is a language which can be generated by an indexed grammar [3], which is an extension of a context-free grammar.

(ii) The decidability of the verification problem follows from the fact that the class of indexed languages is closed under intersection with regular languages and the emptiness problem for the class of indexed languages is decidable.

We also analyze how the complexity of the verification problem alters when the representation of regular languages to specify a check statement and a global security property is changed (deterministic finite automaton, nondeterministic finite automaton

and regular expression). Since the complexity results imply that the problem is generally intractable, we introduce a subclass of programs which model programs containing *checkPermission* in JDK1.2. This subclass is called $\Pi_{\text{JDK1.2}}$. Next, we show that the verification problem for $\Pi_{\text{JDK1.2}}$ is efficiently solvable for the size of a given program.

The rest of the paper is organized as follows. In chapter 2, we define the program model, its operational semantics and the verification problem with a brief example. The decidability of the problem is shown in chapter 3, and the computational complexity of the problem is investigated in chapter 4. In chapter 5, time complexity of the verification problem for the subclass $\Pi_{\text{JDK1.2}}$ is shown to be linear in the program size. We implemented a verification system for the subclass $\Pi_{\text{JDK1.2}}$ based on the algorithm proposed in chapter 5. Implementation issues and experimental results on verifying some programs are described in chapter 6. Finally, we draw some concluding remarks in chapter 7.

1.1. Related Works

Our program model and the definition of the verification problem are based on the model introduced in [18], the pioneering paper which discussed the security verification of programs with stack inspection. The difference between the model in [18] and ours is that linear temporal logic (LTL) formulas [7] is used in [18] to describe both the property in a check statement and a global security property while we use regular languages, whose expressive power to represent a set of finite sequences is properly stronger than that of LTL formulas [10]. Also, the verification algorithm in [18] is based on model checking [7] and works only for mutual recursion-free programs, while we showed that the problem is generally decidable even for programs which contain mutual recursion. Recently, a more general verification method of programs with stack inspection is proposed in [11]. In their method, not only global security properties discussed in both this thesis and [18] but also other properties (*e.g.*, liveness property) can be verified. The verification method in [11] also uses an LTL formula to specify a property of a program. However, what an LTL formula represents is different between [11] and [18]. In [11], an LTL formula is used to represent a set of traces (state sequences) of a program. On the other hand, in [18], an LTL formula is used to represent a set of states (node sequences) of a program. A global security property

in [18] and this thesis can be represented as an LTL formula with regular valuation in [11]. More precisely, a global security property ψ is equivalent to an LTL formula $\mathbf{G}p$ with regular valuation ν, where p is an atomic proposition and $\nu(p)$ is a regular language determined by ψ. On the other hand, an arbitrary LTL formula can be used to represent a property to be verified in [11]. Therefore, their verification method is more general than ours and Jensen's [18] (Table 1.1). However, in their paper, the complexity of the verification problem is not analyzed in detail. Furthermore, a practical verification method and its implementation are presented in this thesis. In [33], a more general notion of stack inspection is proposed by using ABLP logic [1], which is a kind of belief logic, and a sufficient condition for a check statement to succeed is shown. However, the verification of a global security property is not discussed in [33] in contrast with [18], [11] and our papers.

Table 1.1. Representation of a property

	states (node sequences)	traces(state sequences)
[18]	LTL formula	LTL formula $\mathbf{G}p$
this thesis	Regular language †	
[11]		(arbitrary) LTL formula

†Expressive power: LTL formula $\subset$ Regular language

In [8] and [9], which is the pioneering work of information flow, a static analysis based on a lattice model of security classes is proposed. Denning's analysis method has been formalized and extended in various ways by abstract interpretation [26], type theory [32, 14] and process algebra [2]. For example, in the type theoretical approach, a type system is defined so that if a given program is well-typed then the program has a *noninterference* property such that it does not cause undesirable information flow. A structure of security classes modeled as a finite lattice is usually a simple one such as $\{$topsecret, trusted, untrusted$\}$. In [21], a fine grained model of security classes called decentralized labels is proposed. Based on this model, Myers [20] proposes a programming language called JFLOW, for which a static type system for information flow analysis as well as a simple but flexible mechanism for dynamically controlling the privileges (declassification) is provided. However, the correctness of their type system has not been formally verified. All of the above mentioned studies are basically concerned with information flow analysis to ensure that high level secret information

does not flow into an insecret storage. In contrast, [18] and this paper discuss the problem of deciding whether a program possesses an arbitrarily given global security property provided that the program passes local security checks.

Security verification in a distributed system has been extensively studied by using a process algebra called spi calculus and its extensions in [2] and its companion papers. In [30], it is shown that the type system in [32] is no longer correct in a distributed environment and presented a new type system for a multi-threaded language.

In the field of data engineering, access control and information flow control issues have been also extensively studied for a distributed and object-oriented environment (see [5]). For example, Samarati et al. [28] presents an information flow control algorithm which blocks illegal information flows in object-oriented databases. However, their algorithm does not perform semantic analysis inside a method. Semantic analysis of security flows or security verification against inference attacks in object-oriented databases are discussed in [31, 16].

Chapter 2

Preliminaries

2.1. Program Model

Following [18], we model a program with stack inspection as a directed graph called a *flow graph*. Each node of a flow graph corresponds to a location in the program (*program point*). A statement which performs runtime check of access permission such as *checkPermission* in JDK1.2 is called a check statement and is incorporated into the model. A check statement examines whether the current state of the executed program satisfies the property specified in the statement. If the property is satisfied, the program continues its execution. Otherwise, the execution is aborted.

A flow graph has two kinds of edges. The first one is a *transfer edge* (tg), which represents a control flow within a method. For example, if there is a tg from n_1 to n_2 (denoted as $n_1 \stackrel{TG}{\rightarrow} n_2$), then the control can move to n_2 just after the execution of n_1. The second type of edge is a *call edge* (cg), which represents a method invocation. For example, suppose that there is a cg from n_1 to n_2 (denoted as $n_1 \stackrel{CG}{\rightarrow} n_2$). If the control reaches n_1, then the control can further be passed to n_2.

Let ε denote the empty sequence. For a finite set Σ of symbols, let Σ^* denote the set of all finite sequences on Σ including ε. Also let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. Formally, a program

P is a directed graph represented as a 5-tuple $P = (NO, IS, IT, TG, CG)$ such that:

$$\begin{array}{lcl} IS & : & NO \to \{call, return, check(L_\phi)\} \\ IT & \in & NO \\ TG & \subseteq & NO \times NO \\ CG & \subseteq & NO \times NO. \end{array}$$

NO is a set of nodes representing program points. IT is the entry point of the entire program called the *initial node*. TG and CG are sets of transfer edges and call edges, respectively.

The set of nodes is divided into the following three subsets by IS. Let $n \in NO$.

- $IS(n) = call$. n is a *call node* which represents a method call.
- $IS(n) = return$. n is a *return node* which represents the return from a callee method.
- $IS(n) = check(L_\phi)$. n is a *check node* which represents a check statement. If the current state of the program satisfies the property represented by the language L_ϕ, then the execution is continued. Otherwise, the execution is aborted. The syntax and semantics of the language L_ϕ are defined in section 2.2.3.

Conditionals such as *if* statements and *while* statements substitute for nondeterministic statements. For example, consider the following sequence of statements: $m_1()$; if ... then $m_2()$ else $m_3()$. In a flow graph, there will be two tgs $n_1 \stackrel{TG}{\to} n_2$ and $n_1 \stackrel{TG}{\to} n_3$, where n_1, n_2, and n_3 represent $m_1()$, $m_2()$, and $m_3()$, respectively. Transformation methods from an object-oriented program into a flow graph using data flow analysis or type inference have been studied (*e.g.*, [27]).

Note. A flow graph does not always represent the exact behaviors of an original program. More specifically, if an ordinal imperative program P_0 is modeled as a flow graph P, then every execution sequence (trace) of P_0 is also a trace of P, but not vice versa. The reasons why we do such an "approximation" are as follows:

- Most of decision problems for imperative programs which contain either conditional statements and procedure calls or *while* statements are undecidable. Therefore, static program analyses such as type inference and abstract interpretation use an approximation such as the one described here.

- If a flow graph P of an original program P_0 satisfies a safety property discussed in this paper, then it is guaranteed that P_0 also satisfies the property. (This is not true for liveness properties.)

2.2. Operational Semantics

2.2.1 State

Each state of an ordinary imperative program can be represented by a tuple of a current program point (or a continuation), contents of global variables, and a runtime control stack (shortly, stack).

For each invocation to a method m, a frame f_m is allocated and pushed onto the stack. A frame f_m contains actual values of the input arguments of m, values of local variables, the return address, and other information on access permissions which m has. In the flow graph model, however, values of variables and arguments are abstracted away. Hence, a frame degenerates into a return address, i.e, a node. If, in addition, the current program point is also kept on top of the stack, then each state of a program can be represented as a stack, that is, a sequence of nodes.

A *state* of a program $P = (NO, IS, IT, TG, CG)$ is a finite sequence of nodes, which is also called a *stack*. The initial state of P is the stack which contains only the initial node IT. The state immediately after a method call is the state (stack) obtained by pushing the node of the callee onto the current state. If the top element (current program point) of the stack is a node n_1 and $n_1 \stackrel{TG}{\rightarrow} n_2$, then the state just after the execution of n_1 can be the state obtained by replacing the top element n_1 of the stack with n_2. The concatenation of sequences s_1 and s_2 of nodes is represented as $s_1 : s_2$. The sequence which consists of only one node n is denoted by $\langle n \rangle$. We may write n instead of $\langle n \rangle$ if no ambiguity occurs. For example, a state $n_1 : n_2 : n_3$ indicates that the method including the program point n_2 has been called from n_1, the control has reached n_2, the method including program point n_3 has been called from n_2, and the current program point is n_3. If n_3 is a return statement (*i.e.*, $IS(n_1) = return$), then n_3 is popped from the stack and the next state becomes $n_1 : n_4$ where $n_4 \in \{n \mid n_2 \stackrel{TG}{\rightarrow} n\}$.

2.2.2 Trace

The semantics of a program is defined by a *transition relation* $\triangleright$ on the set of states. For states s_1 and s_2, $s_1 \triangleright s_2$ means that the transition from s_1 to s_2 is possible by a unit execution step of the program.

Definition 2.2.1 (transition relation) For a given program $P = (NO, IS, IT, TG, CG)$, the relation $\triangleright$ is the least relation which satisfies the following three rules, where s is a state ($\in NO^*$) and n, m, n_i, n_j are nodes ($\in NO$).

$$\frac{IS(n) = call,\ n \overset{CG}{\rightarrow} m}{s : n \triangleright s : n : m}$$

$$\frac{IS(m) = return,\ n_i \overset{TG}{\rightarrow} n_j}{s : n_i : m \triangleright s : n_j}$$

$$\frac{IS(n) = check(L_\phi),\ s : n \in L_\phi,\ n \overset{TG}{\rightarrow} n_j}{s : n \triangleright s : n_j}$$

□

For a program P, a *trace* of P is a finite sequence of states in which the first state is the initial state and every pair of adjacent states satisfies the transition relation $\triangleright$. The concatenation of states is denoted as $\triangleright$ by slightly abusing the notation. The *set of traces* is defined as follows.

Definition 2.2.2 (set of traces) For a given program $P = (NO, IS, IT, TG, CG)$, the set $[\![P]\!]$ of traces of P is:

$$\begin{aligned}[\![P]\!] \quad = \quad & \{s_1 \triangleright \cdots \triangleright s_k \mid s_1 = \langle IT \rangle, s_1, \ldots, s_k \in NO^*, \\ & \forall i < k, s_i \triangleright s_{i+1}\}.\end{aligned}$$

□

A language L_ϕ in a node $check(L_\phi)$ is a regular language over NO (thus, $L_\phi \subseteq NO^*$). Recall that every state s is a sequence of nodes, that is, $s \in NO^*$. As defined in the third rule of Definition 2.2.1, if the control reaches a node $check(L_\phi)$, then the execution is continued if and only if the current state belongs to L_ϕ.

2.2.3 Security Property in Check Node

The model itself does not assume any particular representation (*e.g.*, regular expression, finite automaton) to denote a regular language L_ϕ although we will use regular expression in this chapter.

Example 2.2.1 (Java stack inspection in JDK1.2) A method invocation *checkPermission*(p) succeeds if

(α) every frame of the stack has permission p, or

(β) a frame (say f) is privileged and every later frame (including f) in the stack has p.

Let $N(p)$ be the set of nodes which have permission p and let PRV be the set of privileged nodes. We can represent *checkPermission*(p) as the check node *check*($JDK(p)$) where:

$$JDK(p) = (\underbrace{NO^*}_{(a)}\underbrace{(PRV \cap N(p))}_{(b)} \cup \underbrace{\varepsilon}_{(c)})\underbrace{(N(p))^*}_{(d)}. \tag{2.1}$$

The concatenation of (c) and (d) represents the set of node sequences which satisfy the condition (α). Note that (b) represents the set of nodes which are privileged and have p as well. Hence, the concatenation of (a), (b) and (d) represents the set of node sequences which satisfy the condition (β). Remember that for a state $n_1 n_2 \cdots n_k$, the leftmost symbol n_1 represents the node at the bottom of the stack, the rightmost symbol n_k represents the node at the top of the stack, and the other nodes $n_2 \cdots n_{k-1}$ are arranged from bottom to top. □

Example 2.2.2 Figure 2.1 shows a program $P = (NO, IS, IT, TG, CG)$ which models a part of an on-line banking system, which serves its clients with a method for withdrawing money. There are four protection domains called *System*, *Provider*, *Client* and *Unknown*. A reliable provider is supplied with *read* and *write* methods and is privileged by the system. All users including clients and unknowns can invoke a *debit* method, which invokes *read* and *write* methods. In the figure, a solid arrow represents a call edge and a dotted arrow represents a transfer edge. Let $NO = \{n_i \mid 1 \leq i \leq 14\}$. Permissions granted to each protection domain as well as the protection domain which

each node belongs to are also shown in figure 2.1. For example, the set of permissions granted to *System* is $\{p_{debit}, p_{read}, p_{write}\}$. Also, the set $N(p)$ of nodes which have permission p is: $N(p_{debit}) = \{n_1, n_2, n_3, n_4, n_7, n_8, \cdots, n_{14}\}$ and $N(p_{read}) = N(p_{write}) = \{n_1, n_2, n_7, n_8, \cdots, n_{14}\}$. Let $PRV = \{n_8, n_9\}$ be the set of nodes which are privileged. The property $JDK(p_{debit})$ specified in node n_7 is represented by the following regular expression:

$$JDK(p_{debit}) = (NO^* \ (PRV \cap N(p_{debit})) \cup \varepsilon)(N(p_{debit}))^*$$

by (2.1) in Example 2.2.1. The properties $JDK(p_{read})$ in n_{11} and $JDK(p_{write})$ in n_{13} are represented in the same way. Consider the following two sequences:

$$\begin{aligned}
\alpha_1 &= n_1 \triangleright n_1 n_3 \triangleright \underline{n_1 n_3 n_7} \triangleright n_1 n_3 n_8 \triangleright \underline{n_1 n_3 n_8 n_{11}} \\
&\quad \triangleright n_1 n_3 n_8 n_{12} \triangleright n_1 n_3 n_9 \triangleright \underline{n_1 n_3 n_9 n_{13}} \\
&\quad \triangleright n_1 n_3 n_9 n_{14} \triangleright n_1 n_3 n_{10} \triangleright n_1 n_4, \\
\alpha_2 &= n_1 \triangleright n_1 n_5 \triangleright \underline{n_1 n_5 n_7} \triangleright n_1 n_5 n_8.
\end{aligned}$$

For the state sequence α_1, check nodes are executed three times, at the underlined states. In each case, the state satisfies the property (belongs to the language) specified in the check node. Therefore, $\alpha_1 \in [\![P]\!]$. For the sequence α_2, $n_1 n_5 n_7 \notin JDK(p_{debit})$ since $n_5 \notin N(p_{debit})$, and hence $\alpha_2 \notin [\![P]\!]$. □

2.3. The Verification Problem

2.3.1 Definition of the Verification Problem

In this chapter, we define a verification problem, which is a generalization of the one in [18]. Each program is required to satisfy a certain global security property such as 'local resource r never be read out by any (malicious) method which does not have permission p.' Intuitively, the verification problem is to verify whether every state in every trace in $[\![P]\!]$ of a given program P satisfies a given global security property. A global security property is expressed as a regular language L_ψ, which is called a *verification property*.

Let $L_{\text{safe}}[\psi] = \{\alpha \mid \alpha \text{ is a state sequence such that every state in } \alpha \text{ belongs to } L_\psi\}$. $L_{\text{safe}}[\psi]$ can be represented as $L_{\text{safe}}[\psi] = (L_\psi \triangleright)^* L_\psi$. A program P *satisfies* a verification property L_ψ if and only if every state in every trace in $[\![P]\!]$ belongs to L_ψ. Formally, we define the **verification problem** as follows:

Instance: A program P and a verification property L_ψ.

Question: $[\![P]\!] \subseteq L_{\text{safe}}[\psi]$?

2.3.2 An Example

Example 2.3.1 Consider Example 2.2.2 again. Let $L_\psi = ((\overline{E_{RW}})^* \cup (N\ (p_{debit}))^* E_{RW} (\overline{E_{RW}})^*)$ be the verification property where $E_{RW} = \{n_{11}, n_{12}, n_{13}, n_{14}\}$. $L_{\text{safe}}[\psi] = (L_\psi \triangleright)^* L_\psi$ means that if the control reaches either the *read* or *write* method successfully, then the control has passed through only nodes which have p_{debit}. In this particular example, $[\![P]\!] \subseteq L_{\text{safe}}[\psi]$ holds, that is, program P satisfies L_ψ. □

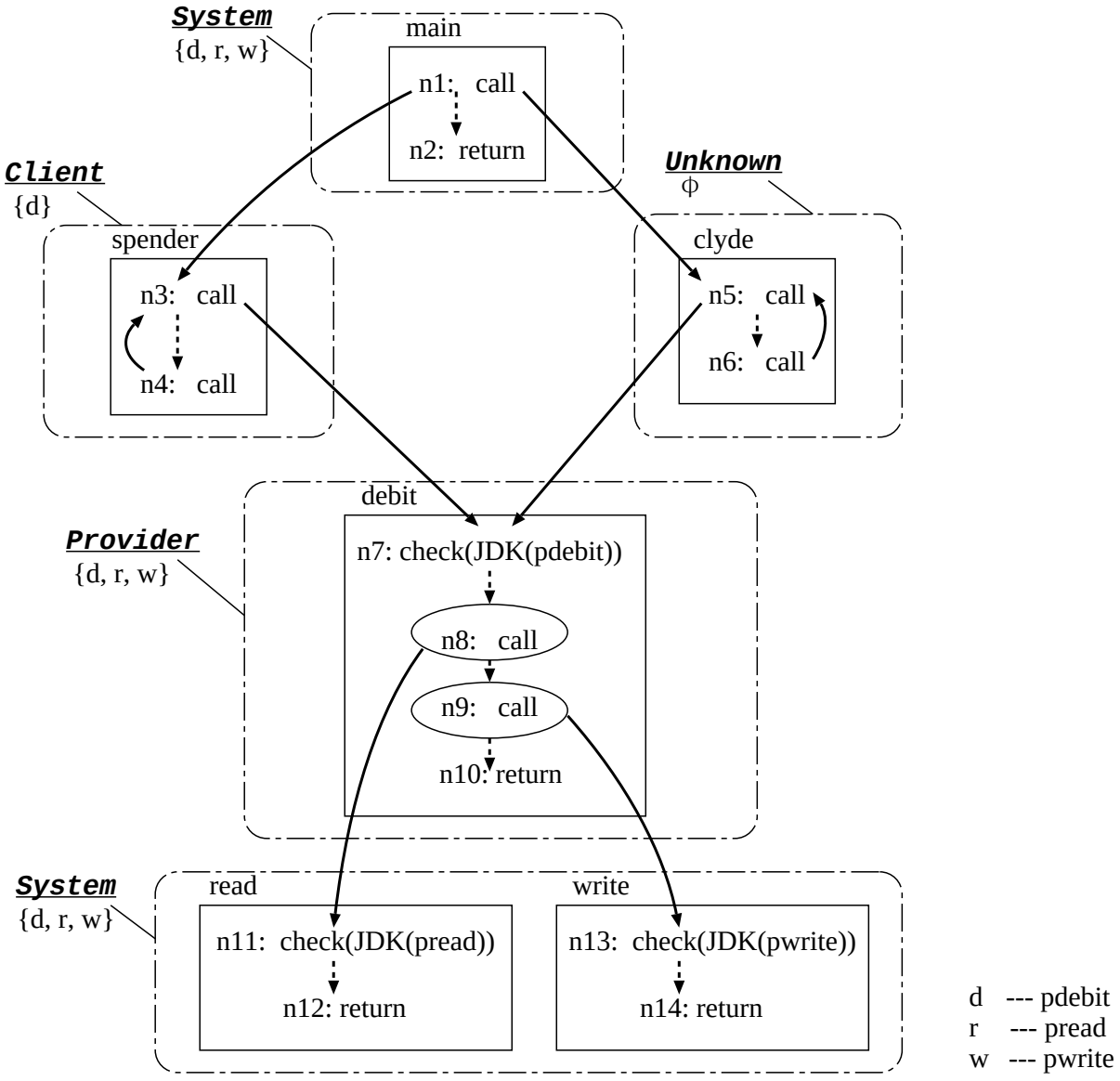


Figure 2.1. A Sample Program

Chapter 3

Decidability of the Verification Problem

3.1. Overview

In this chapter, we present our approach to the verification problem. We will show that for each program P, the set $[\![P]\!]$ of traces of P can be generated by an indexed grammar. The generative capacity of indexed grammars (IGs)[3] is stronger than that of context-free grammars (CFGs) while IGs inherit good mathematical properties from CFGs. Using these properties, we prove the decidability of the verification problem.

As stated in the following lemma, the set of traces can not always be generated by a CFG.

Lemma 3.1.1 *There exists a program P such that the set $[\![P]\!]$ of traces is not a context-free language (CFL).*

Proof. Let $P = (NO, IS, IT, TG, CG)$ be the program where $NO = \{n\}$, $IT = n$, there is no transfer edge in P, and there is only one call edge $n \overset{CG}{\to} n$. The set of traces in P is as follows.

$$[\![P]\!] = \{n,\ n \triangleright nn,\ n \triangleright nn \triangleright nnn, \cdots\}.$$

Let h be the homomorphism defined by $h(n) = n$ and $h(\triangleright) = \varepsilon$. Then we obtain:

$$h([\![P]\!]) = \{n,\ nnn,\ nnnnnn,\ nnnnnnnnnn, \cdots\} = \{n^{i(i+1)/2} \mid i \geq 1\}.$$

It is easy to prove that $h([\![P]\!])$ is not a CFL using the pumping lemma for CFL. Since the class of CFL is closed under homomorphism, $[\![P]\!]$ is not a CFL. □

The above lemma implies that a class of grammars of which generative capacity is stronger than CFG is needed to generate the set of traces. The outline of this chapter is as follows.

(a) A trace in a program is a sequence of stacks, which are sequences of nodes. A state transition does not alter the nodes other than the top and the next to the top node in a stack (see section 2.2) at a time. That is, if $s_i \triangleright s_{i+1}$, then there are $\alpha, \beta, \gamma \in NO^*$ with $|\beta| \leq 2$, $|\gamma| \leq 2$ such that s and s' can be written as $s = \alpha\beta$ and $s' = \alpha\gamma$, respectively. As explained in section 3.2.1, an indexed grammar can generate a sequence which is controlled by such a stack-like data structure.

A sequence of states is called an unchecked trace if the sequence becomes a trace by assuming that every possible local check node succeeds. More precisely, a sequence α of states is an *unchecked trace* in a program P if α is a trace in P when the third inference rule in Definition 2.2.1 is replaced with

$$\frac{IS(n) = check(L_\phi),\ n \overset{TG}{\rightarrow} n_j}{s : n \triangleright s : n_j}.$$

For a given program P, we will define an indexed grammar $G_{P,\mathrm{T}}$ which generates the set of unchecked traces in P in section 3.2.2.

(b) An unchecked trace α is also a trace if α satisfies the properties specified in check nodes in α. That is, if the property specified in a check statement holds at the current state, then the execution continues; otherwise, the execution should be aborted at the state. For this reason, in section 3.2.3, we will also define the regular language $L_{P,\mathrm{C}}$ of state sequences in which any pair of states can be adjacent to each other as long as those states satisfy the property (belong to the language) specified in check nodes. Finally, we will show $[\![P]\!] = L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}}$ in chapter 3.2.4.

3.2. Decidability

3.2.1 Indexed Grammar

Indexed grammar[3] is an extension of CFG. An index grammar (IG) is a 5-tuple $G = (N, T, I, R, S)$ where:

(a) N is a finite set of *nonterminal symbols*,

(b) T is a finite set of *terminal symbols*,

(c) I is a finite set of *index symbols*,

(d) $S \in N$ is the *start symbol*, and

(e) R is a finite set of *productions* of one of the following forms:

(Type 1) $A \to \alpha$

(Type 2) $A \to Bf$

(Type 3) $Af \to \alpha$,

where $A, B \in N$, $f \in I$, and $\alpha \in (N \cup T)^*$.

A nonterminal symbol, a terminal symbol and an index symbol are abbreviated as a nonterminal, a terminal, and an index, respectively. A derivation in IG is similar to a derivation in CFG except that IG has operations on index sequences. The derivation relation $\underset{G}{\to}$ on $(NI^* \cup T)^*$ is defined as the least relation which satisfies the following conditions (1), (2) and (3). In the following, let $\beta, \gamma \in (NI^* \cup T)^*$, $\xi \in I^*$ and $X_i \in N \cup T$.

(1) Let $A \to X_1 X_2 \cdots X_k \in R$ be a Type 1 production. Then,

$$\beta A \xi \gamma \underset{G}{\to} \beta X_1 \xi_1 X_2 \xi_2 \cdots X_k \xi_k \gamma$$

where if $X_i \in N$ then $\xi_i = \xi$, and if $X_i \in T$ then $\xi_i = \varepsilon$.

(2) Let $A \to Bf \in R$ be a Type 2 production. Then,

$$\beta A \xi \gamma \underset{G}{\to} \beta B f \xi \gamma.$$

(3) Let $Af \rightarrow X_1X_2\cdots X_k \in R$ be a Type 3 production. Then,

$$\beta Af\xi\gamma \underset{G}{\rightarrow} \beta X_1\xi_1X_2\xi_2\cdots X_k\xi_k\gamma$$

where if $X_i \in N$ then $\xi_i = \xi$, and if $X_i \in T$ then $\xi_i = \varepsilon$.

A Type 1 production distributes index sequences associated with the nonterminal to which the production is applied to all nonterminals on the right-hand side. A Type 2 production adds an index f to the left-end of the index sequences associated with the left-hand side (and provides the nonterminal on the right-hand side with the resultant index sequence). A Type 3 production deletes the leftmost index of the index sequence and distributes the remaining sequence to all nonterminals on the right-hand side. The reflexive and transitive closure of $\underset{G}{\rightarrow}$ is denoted by $\overset{*}{\underset{G}{\rightarrow}}$. We will simply write $\rightarrow$ for the relation $\underset{G}{\rightarrow}$ if G is clear from the context. The language generated by an IG $G = (N,T,I,R,S)$ is defined as $L(G) = \{w \in T^* \mid S \overset{*}{\underset{G}{\rightarrow}} w\}$.

3.2.2 Set of Unchecked Traces

We will omit the concatenation symbol : of node sequences in the following. For a given program $P = (NO,IS,IT,TG,CG)$, the index grammar $G_{P,\mathrm{T}} = (N,T,I,R,S)$ is constructed as follows.

(a) $N = \{S,W,A,B,C\} \cup \{N_i,N_i',N_i'' \mid n_i \in NO\}$.

(b) $T = NO \cup \{\triangleright\}$.

(c) $I = \{\dot{n}_i \mid n_i \in NO\} \cup \{\$\}$.

(d) Let R be the set of productions which consists of:

$$S \rightarrow W\$ \tag{3.1}$$
$$W \rightarrow A\dot{n}_1 \text{ for } n_1 = IT \tag{3.2}$$
$$A \rightarrow B\triangleright C \tag{3.3}$$
$$A \rightarrow B \tag{3.4}$$
$$B\dot{n}_i \rightarrow Bn_i \text{ for } \forall n_i \in NO \tag{3.5}$$
$$B\$ \rightarrow \varepsilon \tag{3.6}$$

$$C\dot{n}_i \rightarrow N_i \text{ for } \forall n_i \in NO \tag{3.7}$$

$$N''_j \rightarrow A\dot{n}_k \text{ for } \forall n_j, n_k \text{ such that } n_j \overset{TG}{\rightarrow} n_k \tag{3.8}$$

For each node n_i, one of the following sets (i), (ii) and (iii) of productions is added to R according to $IS(n_i)$.

(i) $IS(n_i) = call$:

$$N_i \rightarrow N'_i\dot{n}_i \tag{3.9}$$

$$N'_i \rightarrow A\dot{n}_j \text{ for } \forall n_j \text{ such that } n_i \overset{CG}{\rightarrow} n_j \tag{3.10}$$

(ii) $IS(n_i) = return$:

$$N_i\dot{n}_j \rightarrow N''_j \text{ for } \forall n_j \in NO \tag{3.11}$$

(iii) $IS(n_i) = check(L_\phi)$:

$$N_i \rightarrow A\dot{n}_j \text{ for } \forall n_j \text{ such that } n_i \overset{TG}{\rightarrow} n_j \tag{3.12}$$

It is clear from production (3.12) that $L(G_{P,\mathrm{T}})$ is the set of unchecked traces in P.

3.2.3 Set of Sequences Satisfying Local Checks

The execution which has resulted in a state s continues if and only if s satisfies the following condition.

> If the top element of the state s is n_i and $IS(n_i) = check(L_{\phi_i})$, then $s \in L_{\phi_i}$ holds.

For a check node n_i, let the language $L^{(i)}_{P,C}$ consisting of state sequences which satisfy the above condition for this particular node n_i is represented by the following regular expression.

$$L^{(i)}_{P,\mathrm{C}} = NO^*(NO - \{n_i\}) \cup \varepsilon \cup L_{\phi_i}.$$

We can define the language $L_{P,\mathrm{C}}$ consisting of state sequences which satisfy the above condition for all check nodes as follows:

$$L_{P,\mathrm{C}} = (X\triangleright)^* NO^* \tag{3.13}$$

where $\{n_1, \ldots, n_l\}$ is the set of check nodes in P and $X := L^{(1)}_{P,C} \cap L^{(2)}_{P,C} \cap \cdots \cap L^{(l)}_{P,C}$.

3.2.4 Proofs

First, we will show that the language $L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}}$ coincides with the set $[\![P]\!]$ of traces of P.

Lemma 3.2.1 *Every $\alpha \in L(G_{P,\mathrm{T}})$ can be written as $\alpha = s_1 \triangleright \cdots \triangleright s_n$ where $s_i \in NO^*$ $(1 \leq i \leq n)$ and $n \geq 1$, and there is a derivation of the following form resulting in α. The number at the right-end of each line is the number of the applied production in section 3.2.2, and $\delta_n \in (I - \{\$\})^+$ $(1 \leq i \leq n)$.*

$$\begin{array}{rcll} S & \stackrel{*}{\rightarrow} & A\delta_1\$ & (1),\,(2) \\ & \rightarrow & B\delta_1\$ \triangleright C\delta_1\$ & (3) \\ & \stackrel{*}{\rightarrow} & B\delta_1\$ \triangleright A\delta_2\$ & (7)-(10) \\ & \rightarrow & B\delta_1\$ \triangleright B\delta_2 \triangleright C\delta_2\$ & (3) \\ & \stackrel{*}{\rightarrow} & B\delta_1\$ \triangleright \cdots \triangleright B\delta_{n-1}\$ \triangleright A\delta_n\$ & \\ & \rightarrow & B\delta_1\$ \triangleright \cdots \triangleright B\delta_{n-1}\$ \triangleright B\delta_n\$ & (4) \\ & \stackrel{*}{\rightarrow} & s_1 \triangleright \cdots \triangleright s_n. & (5),\,(6) \end{array}$$

Proof. For an arbitrary derivation $S \stackrel{*}{\rightarrow} s_1 \triangleright \cdots \triangleright s_n$, the only production which can directly generate a terminal symbol is (3.5). The derivation steps using productions (3.6) and (3.7) can be moved to the end of the derivation. It is easy to see that the other steps in this derivation is exactly those stated in the lemma by the definition of $G_{P,\mathrm{T}}$. □

Let $\sigma : NO^* \rightarrow (I - \{\$\})^*$ be the mapping defined by $\sigma(n_{i_1} \cdots n_{i_n}) := \dot{n_{i_n}} \cdots \dot{n_{i_1}}$.

Lemma 3.2.2 *For each program P, $\delta \in (I - \{\$\})^*$ and $s \in T^*$,*

$$B\delta\$ \underset{G_{P,\mathrm{T}}}{\stackrel{*}{\rightarrow}} s \quad \textit{if and only if} \quad s \in NO^* \textit{ and } \delta = \sigma(s).$$

□

Lemma 3.2.3 *For each program P, $s_1 \triangleright \cdots \triangleright s_n \in [\![P]\!]$ if and only if $S \underset{G_{P,\mathrm{T}}}{\stackrel{*}{\rightarrow}} B\delta_1\$ \triangleright \cdots B\delta_{n-1}\$ \triangleright A\delta_n\$$ and $s_1 \triangleright \cdots \triangleright s_n \in L_{P,\mathrm{C}}$, where $\delta_i = \sigma(s_i)$ $(1 \leq i \leq n)$.*

Proof. The *only if* part is shown by induction on n. (The proof of the *if* part is similar.)
(basis) Assume $s_1 \in [\![P]\!]$. By the definition of a trace, $s_1 = \langle IT \rangle = \langle n_1 \rangle$. By productions (3.1) and (3.2),

$$S \rightarrow W\$ \rightarrow A\dot{n_1}\$$$

holds. Also, $\dot{n_1} = \sigma(n_1)$ and $s_1 \in NO \subseteq (X\triangleright)^* NO^* = L_{P,\mathrm{C}}$.

(inductive step) For an integer $n\,(\geq 1)$, assume an inductive hypothesis if $s_1 \triangleright \cdots \triangleright s_n \in [\![P]\!]$, then $S \xrightarrow{*} B\delta_1\$ \triangleright \cdots \triangleright B\delta_{n-1}\$ \triangleright A\delta_n\$$, $\delta_i = \sigma(s_i)$ $(1 \leq i \leq n)$, and $s_1 \triangleright \cdots \triangleright s_n \in L_{P,\mathrm{C}}$. Further assume that $s_1 \triangleright \cdots \triangleright s_n \triangleright s_{n+1} \in [\![P]\!]$. Since $s_n \neq \varepsilon$ we can write $s_n = s'n_v$ for some $s' \in NO^*$ and $n_v \in NO$. By production (3.3),

$$A\delta_n\$ \rightarrow B\delta_n\$ \triangleright C\delta_n\$. \tag{3.14}$$

One of the following three cases holds according to $IS(n_v)$.

(1) $IS(n_v) = call$. Since $s_n \triangleright s_{n+1}$ and $s_n = s'n_v$, there is a node n_j such that $n_v \xrightarrow{CG} n_j$ and $s_{n+1} = s'n_v n_j$. Productions (3.7), (3.9) and (3.10) can be applied in this order to $C\delta_n\$$ in derivation (3.14), and it follows from $\delta_n = \sigma(s_n)$, $s_n = s'n_v$ and $s_{n+1} = s'n_v n_j$ that

$$C\delta_n\$ = C\dot{n_v}\sigma(s')\$ \rightarrow N_v\sigma(s')\$ \rightarrow N'_v\dot{n_v}\sigma(s')\$ \rightarrow A\dot{n_j}\dot{n_v}\sigma(s')\$ = A\sigma(s_{n+1})\$.$$

Hence, $S \xrightarrow{*} B\delta_1\$ \triangleright B\delta_2\$ \triangleright \cdots \triangleright B\delta_n\$ \triangleright A\delta_{n+1}\$$ and $\delta_i = \sigma(s_i)$ $(1 \leq i \leq n+1)$.

Next, we will show $s_1 \triangleright s_2 \triangleright \cdots \triangleright s_n \triangleright s_{n+1} \in L_{P,\mathrm{C}}$. From the inductive hypothesis $s_1 \triangleright s_2 \triangleright \cdots \triangleright s_n \in L_{P,\mathrm{C}}$, we can see $s_i \in X$ $(1 \leq i < n)$ by (3.13). Hence, it suffices to show that $s_n \in X$ and $s_{n+1} \in NO^*$. The latter is trivial. It follows from $s_n = s'n_v$ and $IS(n_v) = call$ that $s_n \in X$ holds by the definition of $L_{P,\mathrm{C}}$.

(2) $IS(n_v) = return$.
Since $s_n \triangleright s_{n+1}$, we can write s_n as $s_n = sn_u n_v$ for some $n_j \in NO$ and $s \in NO^*$ such that $n_u \xrightarrow{TG} n_j$ and $s_{n+1} = sn_j$. Productions (3.7), (3.11) and (3.8) can be applied in this order to $C\delta_n\$$ in derivation (3.14), and

$$\begin{aligned} C\delta_n\$ &= C\dot{n_v}\dot{n_u}\sigma(s)\$ \rightarrow N_v\dot{n_u}\sigma(s)\$ \\ &\rightarrow \quad N''_u\sigma(s)\$ \rightarrow A\dot{n_j}\sigma(s)\$ = A\sigma(s_{n+1})\$. \end{aligned}$$

Hence, $S \xrightarrow{*} B\delta_1\$ \triangleright B\delta_2\$ \triangleright \cdots \triangleright B\delta_n\$ \triangleright A\delta_{n+1}\$$ and $\delta_i = \sigma(s_i)$ $(1 \leq i \leq n+1)$.

It can be shown that $s_1 \triangleright s_2 \triangleright \cdots \triangleright s_n \triangleright s_{n+1} \in L_{P,\mathrm{C}}$ in a similar way to the case of $IS(n_v) = call$.

(3) $IS(n_v) = check(L_{\phi_v})$. Since $s_n \triangleright s_{n+1}$, there is a node n_j such that $n_v \xrightarrow{TG} n_j$ and $s_n \in L_{\phi_v}$. Also, $s_{n+1} = s'n_j$ follows.

Productions (3.7) and (3.12) can be applied in this order to $C\delta_n\$$ in derivation (3.14), and

$$\begin{aligned} C\delta_n\$ &= Cn_v\sigma(s')\$ \\ &\rightarrow \quad N_v\sigma(s')\$ \rightarrow An_j\sigma(s')\$ = A\sigma(s_{n+1})\$. \end{aligned}$$

Hence, $S \xrightarrow{*} B\delta_1\$ \triangleright B\delta_2\$ \triangleright \cdots \triangleright B\delta_n\$ \triangleright A\delta_{n+1}\$$ and $\delta_i = \sigma(s_i)$ $(1 \leq i \leq n+1)$.

In a similar way to the other cases, it suffices to show $s_n \in X$ in order to prove that $s_1 \triangleright s_2 \triangleright \cdots \triangleright s_n \triangleright s_{n+1} \in L_{P,\mathrm{C}}$. It follows from $s_n = s'n_v$ and $s_n \in L_{\phi_v}$ that $s_n \in L_{P,C}^{(v)}$. Therefore, $s_n \in X$.

□

Theorem 3.2.4 *For each program P,* $[\![P]\!] = L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}}$.

Proof. We will show $[\![P]\!] \subseteq L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}}$. Assume that $s_1 \triangleright s_2 \triangleright \cdots \triangleright s_n \in [\![P]\!]$. By Lemma 3.2.3, $G_{P,\mathrm{T}}$ satisfies $S \xrightarrow{*} B\delta_1\$ \triangleright \cdots \triangleright B\delta_{n-1}\$ \triangleright A\delta_n\$$, where $\delta_i = \sigma(s_i)$ $(1 \leq i \leq n)$. By applying Production (3.4) to $A\delta_n$ in the above derivation, we obtain

$$B\delta_1\$ \triangleright \cdots \triangleright B\delta_{n-1}\$ \triangleright A\delta_n\$ \rightarrow B\delta_1\$ \triangleright \cdots \triangleright B\delta_{n-1}\$ \triangleright B\delta_n\$.$$

Since $\delta_i = \sigma(s_i)$ $(1 \leq i \leq n)$, by applying Lemma 3.2.2 to each $B\delta_i\$$, $B\delta_1\$ \triangleright \cdots \triangleright B\delta_n\$ \xrightarrow{*} s_1 \triangleright \cdots \triangleright s_n$ holds, from which $s_1 \triangleright \cdots \triangleright s_n \in L(G_{P,\mathrm{T}})$ follows. By Lemma 3.2.3, $s_1 \triangleright \cdots \triangleright s_n \in L_{P,\mathrm{C}}$ holds, and hence $s_1 \triangleright s_2 \triangleright \cdots \triangleright s_n \in L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}}$. Thus, $[\![P]\!] \subseteq L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}}$ holds.

$L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}} \subseteq [\![P]\!]$ can be shown in a similar way. □

Lemma 3.2.5 [3] *The class of indexed languages is closed under intersection with regular languages. The emptiness problem for indexed languages is decidable.* □

Theorem 3.2.6 *For a given program P and a given verification property* L_ψ*, the verification problem* $[\![P]\!] \subseteq L_{\mathrm{safe}}[\psi]$ *is decidable.*

Proof. The theorem follows from the fact $[\![P]\!] \subseteq L_{\mathrm{safe}}[\psi] \Leftrightarrow [\![P]\!] \cap \overline{L_{\mathrm{safe}}[\psi]} = \emptyset$ (the empty set), Theorem 3.2.4, and Lemma 3.2.5. □

3.3. An Alternative Method

The set $[P]$ of all states which are reachable from the initial state of a program P is defined as follows:

$$[P] = \{s \in NO^* \mid \exists s_1 \triangleright s_2 \triangleright \cdots \triangleright s_i \in [\![P]\!],\ s = s_i\}.$$

It is easy to see that for a program P and a verification property L_ψ, $[P] \subseteq L_\psi$ if and only if $[\![P]\!] \subseteq (L_\psi \triangleright)^* L_\psi$. Hence, we can solve the verification problem by deciding $[P] \subseteq L_\psi$ instead of deciding $[\![P]\!] \subseteq (L_\psi \triangleright)^* L_\psi$. If $[P]$ belongs to a class of languages which is closed under intersection with regular languages and for which the emptiness problem is decidable, then we can obtain a decision algorithm for the verification problem since $[P] \cap \overline{L_\psi} = \emptyset$ if and only if $[P] \subseteq L_\psi$. Let $L(G)$ denote the language generated by a grammar G and let $||G||$ denote the size of G. It is known that a context-free grammar (CFG) G' can be constructed from a CFG G such that $L(G') = L(G) \cap L_\psi$ and $||G'||$ is $O(||G||)$, and also the emptiness problem for context-free language is solvable in linear time in the size of CFG [15]. Hence, if $[P]$ is generated by a CFG G such that $||G||$ is $O(||P||)$, then the verification problem is solvable in polynomial time in $||P||$. However, it is open at the current time whether $[P]$ is a context-free language for an arbitrary program P. If P contains no check node, then $[P]$ is a regular language and hence we can decide whether $[P] \subseteq L_\psi$ (see section 5.1).

Chapter 4

Complexity of the Verification Problem

4.1. Representations of Inputs

The complexity of the verification problem can depend on the representation of regular languages specified in check nodes and a verification property. We mainly use a finite automaton (FA) as the representation of a regular language. A deterministic FA and a nondeterministic FA are denoted by a DFA and an NFA, respectively. Also, a regular expression is denoted by an RE. Let DEXP-POLY time denote the class of decision problems solvable in deterministic $O(c^{p(n)})$ time for a constant c (> 1) and a polynomial p.

In the following sections, we will show that the verification problem is DEXP-POLY time-complete if a verification property is specified by a DFA. For a set A, let $|A|$ denote the cardinality of A. The number of states of an FA M is denoted as $\#M$. Let $P = (NO, IS, IT, TG, CG)$ be a program where P contains $check(L_{\phi_i})$ $(1 \leq i \leq l)$ and each L_{ϕ_i} is specified by an FA M_{ϕ_i}. The size of P is defined as $\|P\| = |NO| + |TG| + |CG| + \max\{\#M_{\phi_1}, \ldots, \#M_{\phi_l}\}$. For an IG $G = (N, T, I, R, S)$, the size of G is defined as $\|G\| = |N| + |T| + |I| + \|R\|$ where $\|R\|$ is the description length of R.

4.2. Upper Bounds

Lemma 3.2.5 can be refined by analyzing the proofs of Lemma 3.2 and Theorem 4.1 of [3] as follows. Let $L(M)$ denote the language accepted by an FA M.

Lemma 4.2.1[3] *(a) For an IG G and an NFA M, an IG G' can be constructed such that $L(G') = L(G) \cap L(M)$ and $\|G'\|$ is $O(\|G\|(\#M)^3)$. (b) The emptiness problem for the language generated by an IG G is solvable in deterministic $O(2^{p(\|G\|)})$ time for a polynomial p.* □

Lemma 4.2.2 *Let $P = (NO, IS, IT, TG, CG)$ be a program. Assume that P contains $check(L_{\phi_i})$ $(1 \leq i \leq k)$ where each L_{ϕ_i} is specified by an NFA M_{ϕ_i}. Also let L_ψ be a verification property specified by a DFA M_ψ. The verification problem for P and L_ψ is solvable in DEXP-POLY time.*

Proof. By Theorems 3.2.4 and 3.2.6, the problem is equivalent to deciding whether

$$L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}} \cap \overline{L_{\mathrm{safe}}[\psi]} = \emptyset \tag{4.1}$$

where $G_{P,\mathrm{T}}$ is the indexed grammar constructed in section 3.2.2, $L_{P,\mathrm{C}}$ is the regular language defined from L_{ϕ_i} $(1 \leq i \leq k)$ in section 3.2.3 and $L_{\mathrm{safe}}[\psi] = (L_\psi \triangleright)^* L_\psi$. We can show the following properties.

(i) $\|G_{P,\mathrm{T}}\|$ is $O(|NO|^2)$.

(ii) Let $n_1 = \max\{\#M_{\phi_1}, \ldots, \#M_{\phi_k}\}$. An NFA $M_{P,\mathrm{C}}$ can be constructed as follows such that $L_{P,\mathrm{C}} = L(M_{P,\mathrm{C}})$ and $\#M_{P,\mathrm{C}}$ is $O(|NO| \cdot n_1)$.

Let n_{ϕ_i} be the node $check(L_{\phi_i})$. In section 3.2.3, $L_{P,\mathrm{C}}$ is defined as $(X \triangleright)^* NO^*$ where

$$\begin{aligned} X &= \bigcap_{i=1}^{k} (NO^*(NO - \{n_{\phi_i}\}) \cup \varepsilon \cup L_{\phi_i}) \\ &= \varepsilon \cup NO^*(NO - \{n_{\phi_1}, \ldots, n_{\phi_k}\}) \cup \bigcup_{i=1}^{k} (NO^* n_{\phi_i} \cap L_{\phi_i}). \end{aligned}$$

(These equations follow the fact that for any sets $A_1,\ldots,A_k$, $B_1,\ldots,B_k$,

$$\begin{aligned}(A_1 \cup B_1) \cap \cdots \cap (A_k \cup B_k) = \quad & (A_1 \cap \cdots \cap A_{k-2} \cap A_{k-1} \cap A_k) \\ & \cup (A_1 \cap \cdots \cap A_{k-2} \cap A_{k-1} \cap B_k) \\ & \cup (A_1 \cap \cdots \cap A_{k-2} \cap B_{k-1} \cap A_k) \\ & \cup (A_1 \cap \cdots \cap A_{k-2} \cap B_{k-1} \cap B_k) \\ & \cup \cdots \\ & \cup (B_1 \cap \cdots \cap B_{k-2} \cap B_{k-1} \cap B_k).\end{aligned}$$

Note that $NO^*(NO - \{n_{\phi_i}\}) \cup \varepsilon \cup L_{\phi_i} = \varepsilon \cup NO^*(NO - \{n_{\phi_i}\}) \cup (NO^* n_{\phi_i} \cap L_{\phi_i})$. When we let $A_1 = \cdots = A_k = \{\varepsilon\}$ and $B_i = NO^*(NO - \{n_{\phi_i}\}) \cup (NO^* n_{\phi_i} \cap L_{\phi_i})$, the right-hand side equals $\{\varepsilon\} \cup \bigcap_{i=1}^{k} B_i$. When we let $A_i = NO^*(NO - \{n_{\phi_i}\})$ and $B_i = NO^* n_{\phi_i} \cap L_{\phi_i}$ for $1 \leq i \leq k$, the right-hand side equals $\bigcap_{i=1}^{k} A_i \cup \bigcup_{i=1}^{k} B_i$ since $A_i \cap B_j = B_j$ and $B_i \cap B_j = \emptyset$ for any distinct i and j.)

Without loss of generality, we assume that each $M_{\phi_i} (1 \leq i \leq k)$ has no ε-move. (If M_{ϕ_i} has an ε-move, then we can construct an NFA equivalent to M_{ϕ_i} which has the same number of states as M_{ϕ_i} in $O((\#M_{\phi_i})^2)$ time.)

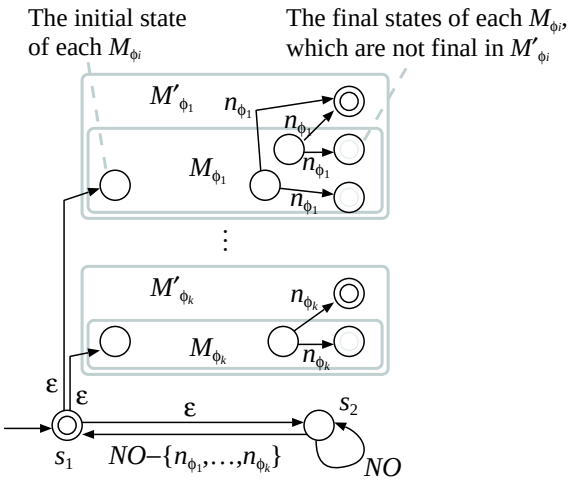
We can construct an NFA M_X such that $X = L(M_X)$ and $\#M_X = \sum_{i=1}^{k}(\#M_{\phi_i} + 1) + 2$, as shown in figure 4.1 where M'_{ϕ_i} is an NFA such that $L(M'_{\phi_i}) = NO^* n_{\phi_i} \cap L_{\phi_i}$. M'_{ϕ_i} can be constructed from M_{ϕ_i} by adding a single new state t' and a transition $s \xrightarrow{n_{\phi_i}} t'$ for each state s such that there is a transition $s \xrightarrow{n_{\phi_i}} t$ for a final state t of M_{ϕ_i}. The initial state of M'_{ϕ_i} is the one of M_{ϕ_i}, and the final state is t'.

Thus, an NFA M_X can be obtained such that $X = L(M_X)$ and $\#M_X = \#M_X^{\mathrm{R}}$.

Using M_X, we can obtain $M_{P,\mathrm{C}}$ as figure 4.2 and $\#M_{P,\mathrm{C}} = \#M_X + 1$, that is, $O(|NO| \cdot n_1)$.

(iii) Let $n_2 = \#M_\psi$. A FA $M_{\overline{\mathrm{safe}}}$ can be constructed such that $\overline{L_{\mathrm{safe}}[\psi]} = L(M_{\overline{\mathrm{safe}}})$ and $\#M_{\overline{\mathrm{safe}}} = n_2$ by defining the set of final states of $M_{\overline{\mathrm{safe}}}$ as the set of non-final states of M_ψ.

Hence, by Lemma 4.2.1 (a), an indexed grammar G_P can be constructed such that $L(G_P) = L(G_{P,\mathrm{T}}) \cap L(M_{P,\mathrm{C}}) \cap L(M_{\overline{\mathrm{safe}}}) = L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}} \cap \overline{L_{\mathrm{safe}}[\psi]}$ and $\|G_P\|$ is $O(|NO|^2(|NO| \cdot n_1 \cdot n_2)^3)$, which is a polynomial order of $\|P\|$ and n_2. By (4.1), the verification problem is equivalent to deciding whether $L(G_P) = \emptyset$. By Lemma 4.2.1 (b) and the above



The initial states of M_X is s_1. The final states of M_X are the ones of any M'_{ϕ_i} and s_1.

Figure 4.1. NFA M_X

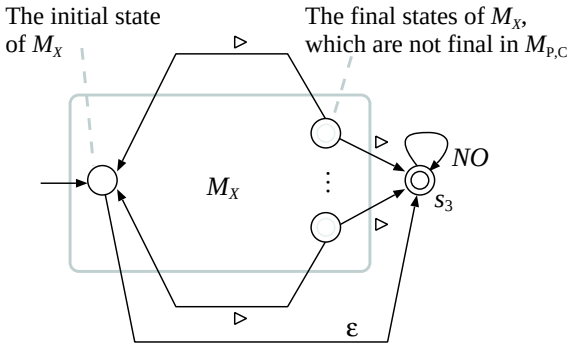
facts, (4.1) can be done in deterministic $O(2^{p(\|P\|+n_2)})$ time for some polynomial p. □

Note. If a verification property L_ψ is specified by an NFA instead of a DFA in Lemma 4.2.2, then the complexity of the problem in this case becomes a double exponential time.

4.3. Lower Bounds

Theorem 4.3.1 *Let $P = (NO, IS, IT, TG, CG)$ be a program and L_ψ a verification property which satisfies the assumption stated in Lemma 4.2.2 except that the language L_{ϕ_i} in each check(L_{ϕ_i}) is specified by a DFA M_{ϕ_i}. The verification problem for P and L_ψ is DEXP-POLY time-complete.*

Proof. By Lemma 4.2.2, it suffices to show that the problem is DEXP-POLY time-hard. It is known that a language L belongs to DEXP-POLY time if and only if L is accepted by a polynomial space-bounded alternating Turing machine (ATM) [6]. For any given polynomial space-bounded ATM M and any input x of M, we can transform



The initial state of $M_{P,C}$ are the one of M_X. The final state of $M_{P,C}$ is s_3.

Figure 4.2. NFA $M_{P,C}$

M and x into a program $P_{M,x}$ and a verification property L_ψ within polynomial time such that

$$[\![P_{M,x}]\!] \not\subseteq L_{\text{safe}}[\psi] \;\Leftrightarrow\; M \text{ accepts } x.$$

Therefore the verification problem is DEXP-POLY time-hard.

Below we show that a transformation which satisfies the above condition exists. Assume that for any input x whose length equals n, M uses not more than $p(n)$ space for a polynomial p. Let $\Gamma = \{\gamma_1, \ldots, \gamma_{|\Gamma|}\}$ be the set of tape symbols of M and γ_1 be the blank symbol. Let δ be the transition function of M.

Consider a program P_1 shown in figure 4.3. A node with $check(NO^*)$ is the one with no operation since every state satisfies NO^*. When the control reaches the node n_1 for the first time, the state (stack) of the program P_1 is a sequence of nodes whose length equals $p(n)+1$. Considering that each $\gamma_{i,j}$ corresponds to the tape symbol γ_i, we can regard this state as one of the $|\Gamma|^{p(n)}$ possible strings contained by the $p(n)$ tape squares of M, which we will refer to as an instantaneous description (ID) as usual. In general, the node n_1 has been visited more than once (by recursive calls), and thus the state of P_1 can be regarded as a sequence of IDs separated by n_1. However, this sequence of IDs may not represent a valid computation of M because any two IDs may be adjacent. To simulate a computation of M, the program should guarantee that the tape square scanned at the last computation step is rewritten to a specified symbol and the other tape squares are preserved.

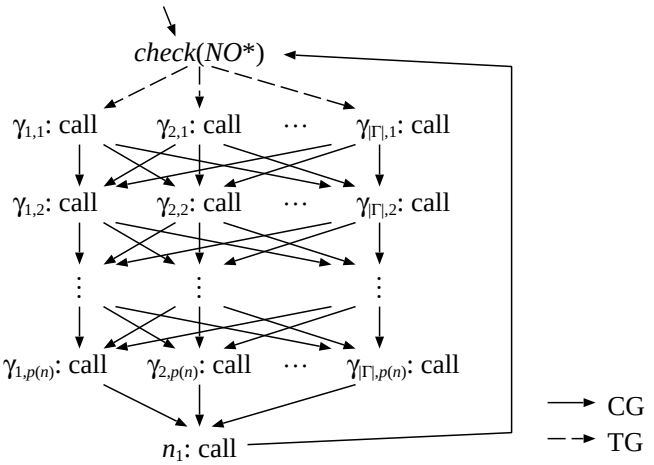


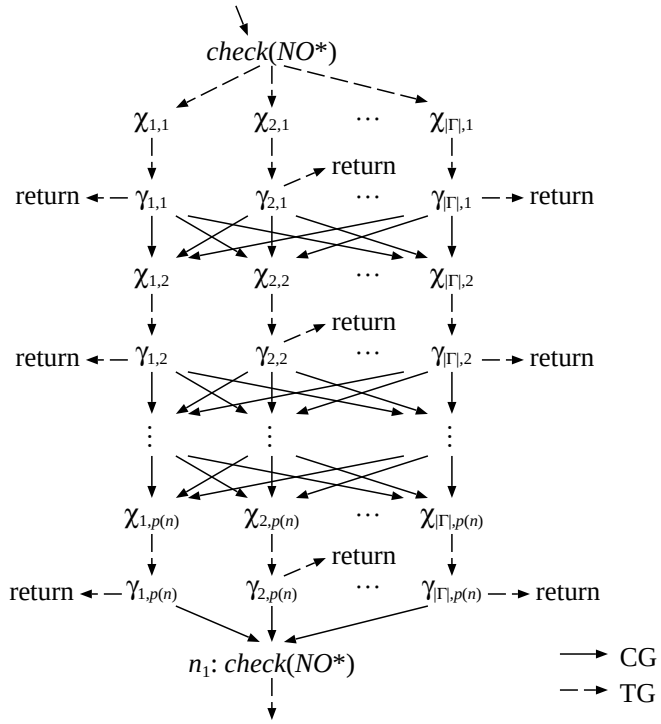
Figure 4.3. Program P_1

For an ID $\alpha = \sigma_1\sigma_2\cdots\sigma_{p(n)}$ (where $\sigma_i = \gamma_{u,i}$ with $1 \le u \le |\Gamma|$ and $1 \le i \le p(n)$), let

$$\alpha[\sigma/k] = \sigma_1\cdots\sigma_{k-1}\sigma\sigma_{k+1}\cdots\sigma_{p(n)},$$

where k corresponds to the position of the tape head of M at the last computation step and σ is the tape symbol which the tape square k is rewritten to. By modifying P_1, we can construct a program $P_2[k,\sigma]$ (figure 4.4) which pushes $\alpha[\sigma/k]$ onto the stack where α is the topmost ID of the stack. More precisely, let $s = s_0\alpha n_0$ (for $s_0 \in NO^*$, α an ID, and $n_0 \in NO$) be a stack. When $P_2[k,\sigma]$ is called with stack s and the control reaches the node n_1 at the bottom of figure 4.4, the stack becomes $s\alpha[\sigma/k]n_1$ (cf. figure 4.7). $P_2[k,\sigma]$ is obtained by attaching a check node $\chi_{i,j}$ and a return node to each $\gamma_{i,j}$ in P_1. $\chi_{i,j}$ is $check(NO^*\gamma_{i,j}NO^{p(n)+1})$ for all $j \neq k$, $check(NO^*)$ for $j = k$ and i such that $\gamma_i = \sigma$, and $check(\emptyset)$ otherwise. Figure 4.5 shows the state of $P_2[k,\sigma]$ when the control is at $\chi_{i,j}$. These check nodes obstruct a sequence of IDs which does not represent a valid computation of M.

Let $\alpha = \sigma_1\sigma_2\cdots\sigma_{p(n)}$ be an ID with state q and head position k. Assume that $(q',\sigma,\Delta) \in \delta(q,\sigma_k)$, that is, a possible move at α is to rewrite k-th tape square from σ_k to σ, change the state from q to q', and change the head position from k to $k' = k+\Delta$. A program $P_3[k,\sigma,q',k']$ in figure 4.6 with topmost ID α on the stack first pushes the ID α' obtained from α by the above move $(q',\sigma,\Delta) \in \delta(q,\sigma_k)$, and simulates further moves from α' recursively (figure 4.7). $P_3[k,\sigma,q',k']$ first executes $P_2[k,\sigma]$ to write the



$\chi_{i,j}$ is $check(L_{i,j})$ where

$$L_{i,j} = \begin{cases} NO^*\gamma_{i,j}NO^{p(n)+1} & \text{if } j \neq k, \\ NO^* & \text{if } j = k \text{ and } \gamma_i = \sigma, \\ \emptyset & \text{otherwise.} \end{cases}$$

Figure 4.4. Program $P_2[k,\sigma]$

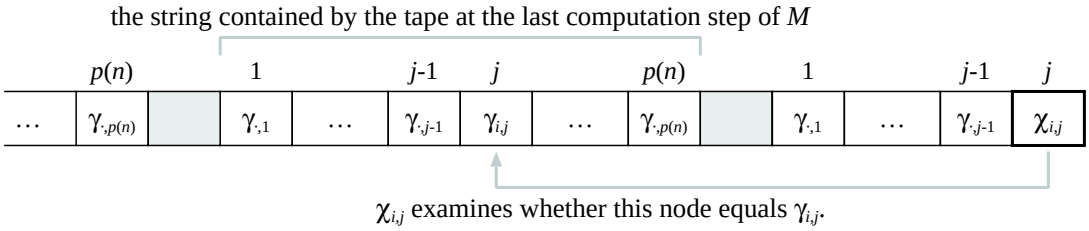
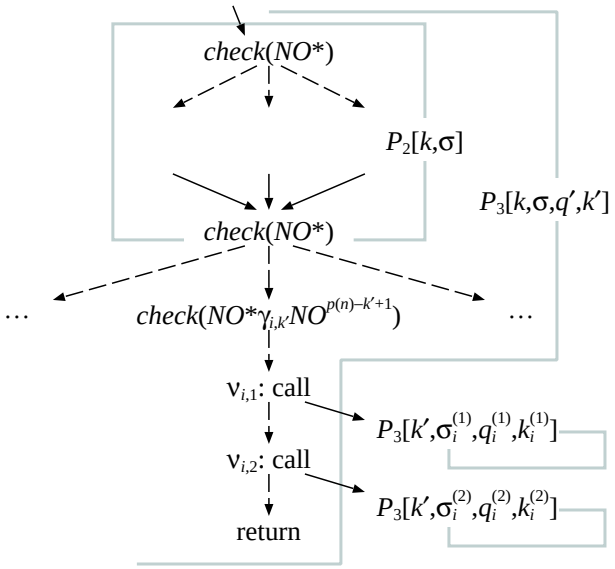
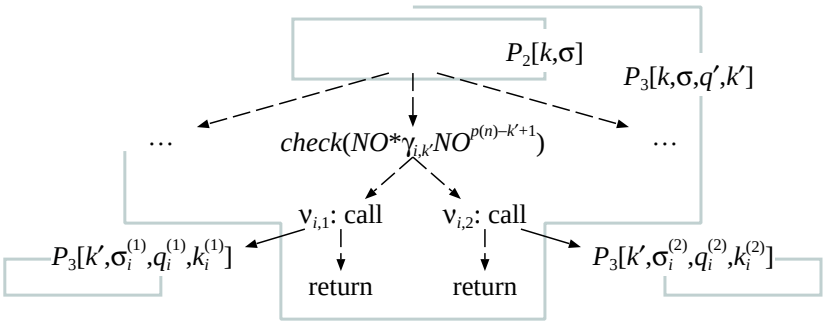


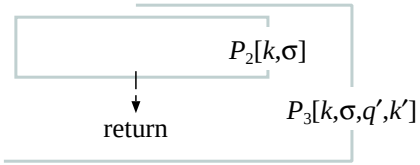
Figure 4.5. The stack of $P_2[k,\sigma]$ when the control is at $\chi_{i,j}$



(a) q' is a universal state



(b) q' is an existential state



(c) q' is a final state

In this figure, we assume that

$$\begin{aligned}\delta(q',\gamma_i) &= \{(q_i^{(1)},\sigma_i^{(1)},\Delta_i^{(1)}),(q_i^{(2)},\sigma_i^{(2)},\Delta_i^{(2)})\},\\ k_i^{(1)} &= k'+\Delta_i^{(1)}, \text{ and}\\ k_i^{(2)} &= k'+\Delta_i^{(2)}.\end{aligned}$$

Figure 4.6. Program $P_3[k,\sigma,q',k']$

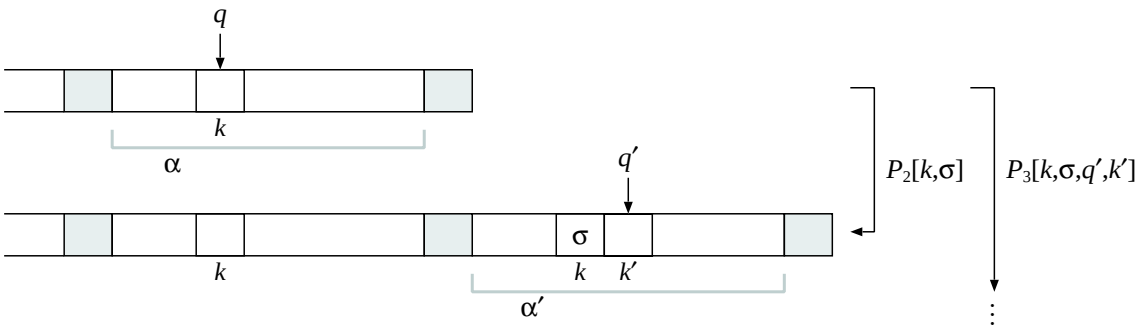


Figure 4.7. The extension of the stack made by $P_2[k,\sigma]$ and $P_3[k,\sigma,q,k']$

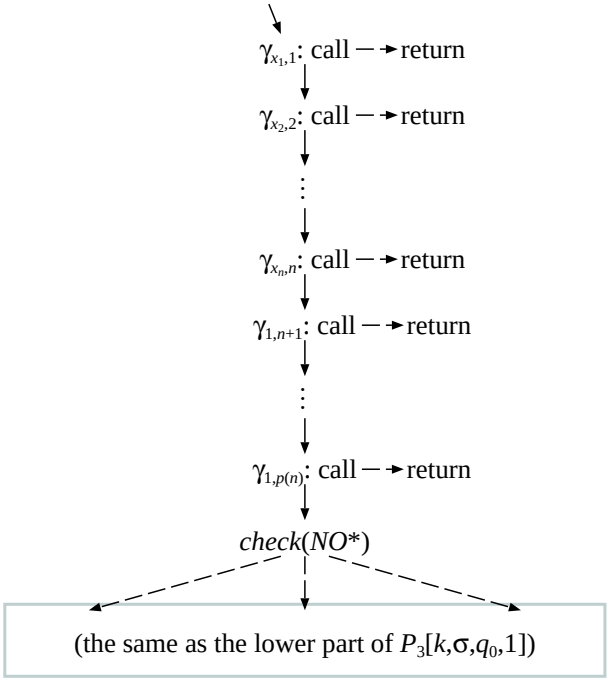
new ID α' onto the stack. After executing $P_2[k,\sigma]$, $P_3[k,\sigma,q',k']$ examines whether the current contents of the tape square k' equals $\gamma_i \in \Gamma$ by $check(NO^*\gamma_{i,k'}NO^{p(n)-k'+1})$, and calls $P_3[k',\sigma',q'',k'']$ for each $(q'',\sigma',\Delta') \in \delta(q',\gamma_i)$ and $k'' = k' + \Delta'$. $P_3[k,\sigma,q',k']$ calls all these $P_3[k',\sigma',q'',k'']$ sequentially if q' is a universal state of M. If q' is an existential state of M, $P_3[k,\sigma,q',k']$ calls any one of these $P_3[k',\sigma',q'',k'']$ and returns. Otherwise, that is, if q' is a final state of M, $P_3[k,\sigma,q',k']$ calls no $P_3[k',\sigma',q'',k'']$ and simply returns. Thus $P_3[k,\sigma,q',k']$ returns if and only if the configuration of M which consists of q', k' and the contents of the tape (ID) written on the stack is a yes-configuration.

A program P_x which simulates the initial configuration of M is similar to $P_3[k,\sigma,q_0,1]$ where q_0 is the initial state of M; P_x does not execute $P_2[k,\sigma]$ and instead it writes the input string x onto the stack (figure 4.8).

The overall program $P_{M,x}$ constructed by the transformation consists of P_x, $P_3[k,\sigma,q',k']$ for all k, σ, q' and k', and two call nodes n_s and n_t (figure 4.9). Note that the subgraph P_x and $P_3[k,\sigma,q',k']$ for all k, σ, q' and k' do not share a node with each other. This $P_{M,x}$ has many $\gamma_{i,j}$s for each i and j. Define the subexpression $\gamma_{i,j}$ appearing in the regular expressions in figures 4.4 and 4.6 as the union of all those nodes. An execution of $P_{M,x}$ reaches the node n_t if and only if M accepts x. We can simply let $L_\psi = (NO - \{n_t\})^*$.

A verification property $L_\psi = (NO - \{n_t\})^*$ can be represented by a DFA with two states. On the other hand, for each check node $\chi_{i,j} = check(L_{i,j})$ of $P_2[k,\sigma]$, we defined $L_{i,j} = NO^*\gamma_{i,j}NO^{p(n)+1}$ for all $j \neq k$. However, the number of states of a DFA which accepts $L_{i,j}$ is more than exponential to n. By replacing the definition of $L_{i,j}$ with

$$L_{i,j} = NO^*\nu\overline{\nu}^{j-1}\gamma_{i,j}\overline{\nu}^{p(n)-j}\nu\overline{\nu}^{j}, \tag{4.2}$$



Let $x = \gamma_{x_1}\gamma_{x_2}\ldots\gamma_{x_n}$. ($\gamma_1$ is the blank symbol.)

Figure 4.8. Program P_x

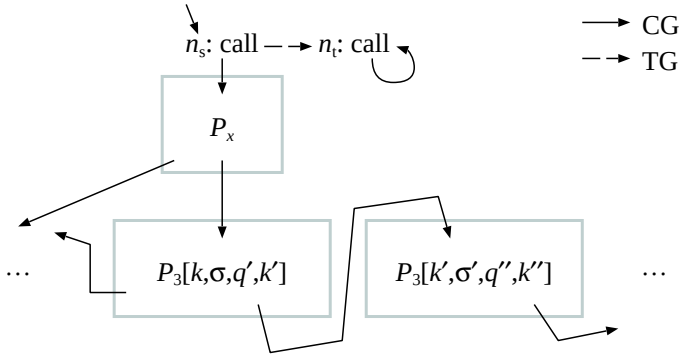
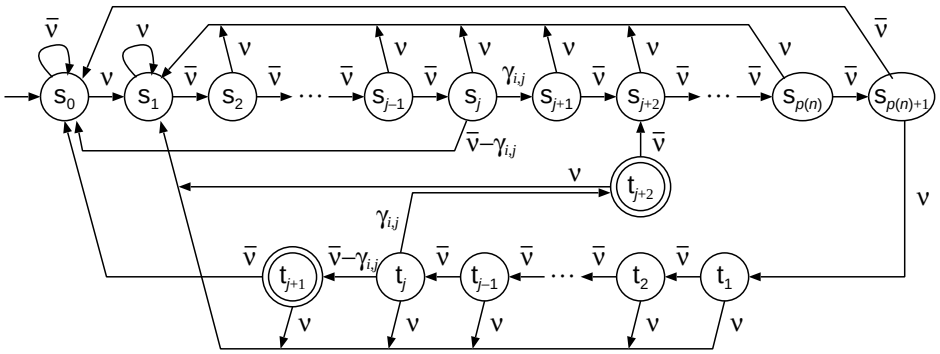


Figure 4.9. Program $P_{M,x}$



This figure is for the case that $j < p(n)$. When $j = p(n)$, replace the transitions from t_{j+2} (i.e., $t_{j+2} \xrightarrow{\overline{\nu}} s_{j+2}$ and $t_{j+2} \xrightarrow{\nu} s_1$) with $t_{j+2} \xrightarrow{\overline{\nu}} s_0$ and $t_{j+2} \xrightarrow{\nu} t_1$.

Figure 4.10. A DFA equivalent to $NO^*\nu\overline{\nu}^{j-1}\gamma_{i,j}\overline{\nu}^{p(n)-j}\nu\overline{\nu}^{j}$

we can show that the verification problem is still DEXP-POLY time-hard even if a regular language in each check node is specified by a DFA. The proof of this theorem is also valid under this definition. In (4.2), ν is the set of nodes which consists of n_s and all $\nu_{i,1}$ and $\nu_{i,2}$ of each $P_3[k,\sigma,q',k']$ (or P_x), and $\overline{\nu} = NO - \nu$. A symbol separating two IDs in figure 4.5 is an element ν. A DFA which accepts $L_{i,j}$ is shown in figure 4.10. The number of states of this DFA is $p(n) + j + 4$ ($\leq 2p(n) + 4$). In a similar way, we can also represent a check node in the lower part of each $P_3[k,\sigma,q',k']$ by a polynomially-sized DFA. □

Note. In [18], the complexity of their verification algorithm is not discussed. Below we briefly analyze their algorithm. The best known upper-bound of the time complexity of the verification problem for a Kripke structure S and an LTL formula f is linear in the size of S and exponential in the size of f [7]. Let us assume that a program P in the flow graph model of [18] and a verification property specified by an LTL formula ψ is given. The size of the Kripke structure induced by P and ψ in [18] is exponential in both the size $\|P\|$ of P and the size $\|\psi\|$ of ψ. Hence, the time complexity of the verification algorithm in [18] is exponential in both $\|P\|$ and $\|\psi\|$. Also note that although we extend the [18]'s model so that check statements in a program are specified by regular languages instead of LTL formulas, the PSPACE-hardness of the verifica-

tion problem shown by Proposition 5.2.3 holds even for the [18]'s model because the languages in check nodes of the program P_{Q3SAT} and the verification property L_ψ in the proof of Proposition 5.2.3 can be specified by simple LTL formulas. Also note that P_{Q3SAT} does not contain mutual recursion, and thus the verification problem is intractable even if we assume the absence of mutual recursion as is done in the verification algorithm in [18].

4.4. Reversing Stack Contents

When we write the contents of a stack as a sequence of nodes, there is no special reason for arranging nodes in such a way that the left-end is the stack bottom. Below we examine whether the complexity of the verification problem alters if we arrange nodes in the order reverse to the original one to denote the contents of a stack.

Let M be an FA $M = (\Sigma, Q, \delta, Q_0, Q_F)$ where Σ is a set of input symbols, Q is a finite set of states, $\delta : Q \times \Sigma \to 2^Q$ is a state transition function, $Q_0 \subseteq Q$ is the set of initial states and $Q_F \subseteq Q$ is the set of final states. Define $\delta^{-1} : Q \times \Sigma \to 2^Q$ as $\delta^{-1}(q,a) = \{q' \mid q \in \delta(q',a)\}$. Let us define the reverse of M as $M^R = (\Sigma, Q, \delta^{-1}, Q_F, Q_0)$. For a sequence $w = a_1 a_2 \cdots a_n$, we write $w^R = a_n \cdots a_2 a_1$ and for a language L, we let $L^R = \{w^R \mid w \in L\}$. It is clear that for an FA M, $L(M^R) = (L(M))^R$. For an FA M, if M^R is a DFA then M is called a DFA$^{\text{R}}$.

First, consider the lower-bound of the verification problem. In the proof of Theorem 4.3.1, the most complex check nodes in the transformed program have the form of $\mathit{check}(NO^*\gamma NO^{p(n)})$ where $\gamma \in NO$ and $p(n)$ is a polynomial in the input size. If $\alpha \in NO^*\gamma NO^{p(n)}$, then the number of symbols appearing to the right of γ in α should be $p(n)$ and hence $NO^*\gamma NO^{p(n)}$ can be accepted by a DFA$^{\text{R}}$ with $p(n)+3$ states. Hence, the problem is still DEXP-POLY time-hard if the properties in check nodes are specified by DFA$^{\text{R}}$s instead of DFAs. Also, the verification property $(NO - \{n_t\})^*$ constructed in the proofs of Theorem 4.3.1 and Proposition 5.2.3 can be accepted by a DFA$^{\text{R}}$ with two states.

Next, consider the upper-bound of the verification problem. Recall the proofs of Theorem 3.2.6 and Lemma 4.2.2. Let P be a program and L_ψ be a verification property specified by a DFA M_ψ. The verification problem is to decide whether $[\![P]\!] \subseteq L_{\text{safe}}[\psi]$, which is equivalent to deciding $(L_{vrf} =)[\![P]\!] \cap \overline{L_{\text{safe}}[\psi]} = \emptyset$. We can construct

an indexed grammar G such that $L(G) = L_{vrf}$ and the size of G is a polynomial in $||P||$ and $\sharp M_\psi$. Now let us assume that L_ψ is specified by a DFA$^{\mathrm{R}}$ M_ψ. Obviously, ${L_{vrf}}^R = [\![P]\!]^R \cap \overline{L_{\mathrm{safe}}[\psi]^R}$. The verification problem for P and L_ψ is equivalent to deciding ${L_{vrf}}^R = \emptyset$. In a similar way to the construction of the indexed grammar G described above, we can construct an indexed grammar G' such that $L(G') = {L_{vrf}}^R$. From this observation, we can show that the upper-bound of the complexity of the problem remains the same if a verification property is specified by a DFA$^{\mathrm{R}}$instead of a DFA.

Chapter 5

Program Subclasses

5.1. Programs with Trivial Check Nodes

In this section, we consider the subclass of programs which contain only $\mathit{check}(\mathit{NO}^*)$ as a check node. Since NO^* is the set of all states (i.e., $s \in \mathit{NO}^*$ for every state s), the execution of $\mathit{check}(\mathit{NO}^*)$ always succeeds. We will show that time complexity of the verification problem for this subclass is linear in the size of a program while the complexity depends on the representation of a verification property. This subclass, called $\Pi_{\text{check-free}}$, might seem of no practical use since no program in this class can substantially control any access. However, in section 5.2 we will introduce a broader subclass $\Pi_{\text{JDK1.2}}$ of programs which exactly model programs with *checkPermission* in JDK1.2, and show that the verification problem for $\Pi_{\text{JDK1.2}}$ can be efficiently solved by transforming a program in $\Pi_{\text{JDK1.2}}$ to a program in $\Pi_{\text{check-free}}$.

Let $\Pi_{\text{check-free}}$ denote the subclass of programs which contain only $\mathit{check}(\mathit{NO}^*)$ as a check node. We will show that for a program P in $\Pi_{\text{check-free}}$, $[P]$ is a regular language and hence we can efficiently decide whether $[P] \subseteq L_\psi$ (see section 3.3). For any given program $P = (\mathit{NO}, \mathit{IS}, \mathit{IT}, \mathit{TG}, \mathit{CG})$ in $\Pi_{\text{check-free}}$, we define a predicate CR (can return) : $\mathit{NO} \rightarrow \{\mathit{True},\ \mathit{False}\}$ such that $\mathrm{CR}(n) = \mathit{True}$ if and only if the control can return after n is invoked.

$$\frac{\mathit{IS}(n_1) = \mathit{call},\ n_1 \overset{\mathit{CG}}{\rightarrow} n_2,\ n_1 \overset{\mathit{TG}}{\rightarrow} n_3,\ \mathrm{CR}(n_2) = \mathrm{CR}(n_3) = \mathit{True}}{\mathrm{CR}(n_1) = \mathit{True}} \tag{5.1}$$

$$\frac{IS(n) = return}{\mathrm{CR}(n) = True} \tag{5.2}$$

$$\frac{IS(n_1) = check(NO^*),\ n_1 \xrightarrow{TG} n_2,\ \mathrm{CR}(n_2) = True}{\mathrm{CR}(n_1) = True} \tag{5.3}$$

Lemma 5.1.1 *Let $P = (NO, IS, IT, TG, CG)$ be a program in $\Pi_{\text{check-free}}$. For an arbitrary node $n \in NO$ of P,*

$$\mathrm{CR}(n) = True \quad \textit{if and only if} \quad \begin{array}{l} \textit{there exist a node } n' \textit{ such} \\ \textit{that } IS(n') = return \textit{ and} \\ \textit{a valid state sequence} \\ n \triangleright \cdots \triangleright n', \end{array}$$

where a valid state sequence T is a sequence of states satisfying the following condition:

$$T = s_1 \triangleright \cdots \triangleright s_k \ \textit{ such that } s_1, \ldots, s_k \in NO^* \textit{ and } \forall i < k.\ s_i \triangleright s_{i+1}.$$

Proof. The *only if* part can be shown by induction on the application number of inference rules of CR used for deriving $\mathrm{CR}(n) = True$. According to $IS(n)$, one of the following three cases holds.

- $IS(n) = return$. The proof is trivial. $(n' = n.)$
- $IS(n) = call$. Since $\mathrm{CR}(n) = True$ is obtained only by inference rule (5.1), there exists a node m_2 ($n \xrightarrow{CG} m_2$) and a node m_3 ($n \xrightarrow{TG} m_3$) such that $\mathrm{CR}(m_2) = \mathrm{CR}(m_3) = True$ holds. By the induction hypothesis for node m_2, there are a node m' ($IS(m') = return$) and a valid state sequence $m_2 \triangleright \cdots \triangleright m'$. Similarly, by the induction hypothesis for node m_3, there exist a node n' ($IS(n') = return$) and a valid state sequence $m_3 \triangleright \cdots \triangleright n'$. Since $n \xrightarrow{CG} m_2$ and $n \xrightarrow{TG} m_3$, there is a valid state sequence $n \triangleright n m_2 \triangleright \cdots \triangleright n m' \triangleright m_3 \triangleright \cdots \triangleright n'$.
- $IS(n) = check(NO^*)$. Since $\mathrm{CR}(n) = True$ is obtained only by inference rule (5.3), there exists a node m_2 ($n \xrightarrow{TG} m_2$) and $\mathrm{CR}(m_2) = True$ holds. By the induction hypothesis, there exist a node n' ($IS(n') = return$) and a valid sequence $m_2 \triangleright \cdots \triangleright n'$. Since $n \xrightarrow{TG} m_2$, we obtain $n \triangleright m_2 \triangleright \cdots \triangleright n'$.

The proof of the *if* part is as follows. Suppose that there is a valid state sequence $T_1 = s_1 \triangleright \cdots \triangleright s_l$ where $s_i \in NO^*$ $(1 \leq i \leq l)$, $s_1 = n$, $s_l = n'$, and $IS(n') = return$. This part is proved by induction on l. Note that in the case of $l = 1$, $IS(n) = return$.

- $IS(n) = return$. By inference rule (5.2), $\mathrm{CR}(n) = True$.
- $IS(n) = check(NO^*)$. There is a node n_2 such that $n \stackrel{TG}{\rightarrow} n_2$ and $s_2 = n_2$. The state sequence $s_2 \triangleright \cdots \triangleright s_l$ satisfies the condition of this lemma and is shorter than T_1. By the induction hypothesis, $\mathrm{CR}(n_2) = True$. Hence, $\mathrm{CR}(n) = True$ holds by inference rule (5.3).
- $IS(n) = call$. The valid state sequence T_1 can be written as $T_1 = n \triangleright nm \triangleright ns'_1 \triangleright \cdots \triangleright ns'_k \triangleright nm' \triangleright n_2 \triangleright \cdots \triangleright n'$, where $s'_i \in NO^+$ $(1 \leq i \leq k)$, $IS(m') = return$, $n \stackrel{CG}{\rightarrow} m$ and $n \stackrel{TG}{\rightarrow} n_2$. Since valid state sequence $m \triangleright s'_1 \triangleright \cdots \triangleright s'_k \triangleright m'$ obtained by removing the leftmost node n from every state in subsequence $nm \triangleright ns'_1 \triangleright \cdots \triangleright ns'_k \triangleright nm'$ of T_1 satisfies the condition of this lemma and is shorter than T_1, $\mathrm{CR}(m) = True$ holds by the induction hypothesis. Similarly, for subsequence $n_2 \triangleright \cdots \triangleright n'$ of T_1, $\mathrm{CR}(n_2) = True$ holds. Hence, we obtain $\mathrm{CR}(n) = True$ by inference rule (5.1).

□

Using the predicate CR, we can construct a regular grammar $G_{P,\mathrm{S}} = (N, T, R, S)$ which generates the set $[P]$ as follows.

(a) N is a finite set of nonterminal symbols and $N = \{S\} \cup \{N_i \mid n_i \in NO\}$.

(b) T is a finite set of terminal symbols and $T = NO$.

(c) S is the start symbol.

(d) R is the set of productions which consists of:

$$S \rightarrow N_1 \quad \text{for } n_1 = IT \tag{5.4}$$

$$N_i \rightarrow n_i \quad \text{for } \forall n_i \in NO. \tag{5.5}$$

For each node n_i, the following productions are added to R according to $IS(n_i)$.

(1) $IS(n_i) = call$:

$$N_i \to n_i N_j \ \text{ for } \forall n_j.\ n_i \overset{CG}{\to} n_j \tag{5.6}$$

If $\exists n_j.\ n_i \overset{CG}{\to} n_j$ and $\mathrm{CR}(n_j) = True$, then the following production is also added to R.

$$N_i \to N_k \ \text{ for } \forall n_k.\ n_i \overset{TG}{\to} n_k \tag{5.7}$$

(2) $IS(n_i) = check(NO^*)$:

$$N_i \to N_j \ \text{ for } \forall n_j.\ n_i \overset{TG}{\to} n_j \tag{5.8}$$

We can show that the language $L(G_{P,\mathrm{S}})$ generated by regular grammar $G_{P,\mathrm{S}}$ coincides with the set $[P]$ of states.

Theorem 5.1.2 *For a program* $P = (NO, IS, IT, TG, CG)$ *in* $\Pi_{\text{check-free}}$, $L(G_{P,\mathrm{S}}) = [P]$.

Proof. It suffices to show that for every $s \in NO^*$, $S \overset{*}{\to}_{G_{P,\mathrm{S}}} s$ *if and only if* $IT \triangleright \cdots \triangleright s \in [\![P]\!]$ holds.

The *only if* part is shown by induction on the sum of application numbers of production rules (5.6)–(5.8).

(basis) If the derivation $S \overset{*}{\to}_{G_{P,\mathrm{S}}} s$ is obtained without applying the production rules (5.6)–(5.8), then

$$\begin{array}{lll} S & \to_{G_{P,\mathrm{S}}} N_1 & \text{by } (5.4) \\ & \to_{G_{P,\mathrm{S}}} n_1 = s. & \text{by } (5.5) \end{array}$$

Clearly, $n_1 = IT \in [\![P]\!]$ holds.

(inductive step) We consider the case that the last production among (5.6)–(5.8) applied in the derivation is (5.7). The proof of the other cases is similar. Suppose there is a derivation of the following form.

$$\begin{array}{lll} S & \overset{*}{\to}_{G_{P,\mathrm{S}}} n_1 \cdots n_l N_i & \\ & \to_{G_{P,\mathrm{S}}} n_1 \cdots n_l N_k & \text{by } (5.7) \\ & \to_{G_{P,\mathrm{S}}} n_1 \cdots n_l n_k. & \text{by } (5.5) \end{array} \tag{5.9}$$

Then, the following derivation also exists.

$$\begin{array}{lll} S & \overset{*}{\to}_{G_{P,\mathrm{S}}} n_1 \cdots n_l N_i & \\ & \to_{G_{P,\mathrm{S}}} n_1 \cdots n_l n_i. & \text{by } (5.5) \end{array}$$

Since the application number of production rules (5.6)–(5.8) in the above derivation is less than that of the derivation (5.9), we can use the inductive hypothesis and obtain

$$IT \triangleright \cdots \triangleright n_1 \cdots n_l n_i \in [\![P]\!]. \tag{5.10}$$

On the other hand, by the existence of the production (5.7) $(N_i \to N_k)$, we can see that $IS(n_i) = call$, $\exists n_j.\ n_i \overset{CG}{\to} n_j$, $\mathrm{CR}(n_j) = True$ and $n_i \overset{TG}{\to} n_k$. Therefore, by Lemma 5.1.1, there is a node n_r such that $IS(n_r) = return$ and a valid state sequence:

$$n_j \triangleright \cdots \triangleright n_r. \tag{5.11}$$

From (5.10), (5.11) and the fact that $n_i \overset{CG}{\to} n_j$, $IT \triangleright \cdots \triangleright n_1 \cdots n_l n_i \triangleright n_1 \cdots n_l n_i n_j \triangleright \cdots \triangleright n_1 \cdots n_l n_i n_r \in [\![P]\!]$. Since $n_i \overset{TG}{\to} n_k$, $IT \triangleright \cdots \triangleright n_1 \cdots n_l n_k \in [\![P]\!]$.

The proof of *if* part is shown by induction on the length of the trace.
(basis) Clearly, $\langle IT \rangle \in [\![P]\!]$ and $S \overset{*}{\to}_{G_{P,\mathrm{S}}} n_1 = IT$.
(inductive step) The claim is proved according to the type $IS(n_r)$ of node n_r in trace $IT \triangleright \cdots \triangleright n_1 \cdots n_l n_i n_r \triangleright s \in [\![P]\!]$.

- $IS(n_r) = call$ or $check(NO^*)$. The claim can easily be proved in this case.
- $IS(n_r) = return$. In this case, $s = n_1 \cdots n_l n_k$, $\exists n_j.\ n_i \overset{CG}{\to} n_j$ and $n_i \overset{TG}{\to} n_k$. Therefore, we can write this trace as

  $$IT \triangleright \cdots \triangleright n_1 \cdots n_l n_i \triangleright n_1 \cdots n_l n_i n_j \triangleright \cdots \triangleright n_1 \cdots n_l n_i n_r \triangleright n_1 \cdots n_l n_k \in [\![P]\!]$$

  and by Lemma 5.1.1, $\mathrm{CR}(n_j) = True$. Therefore, $G_{P,\mathrm{S}}$ contains the rule (5.7) $N_i \to N_k$. On the other hand, since $IT \triangleright \cdots \triangleright n_1 \cdots n_l n_i \in [\![P]\!]$, it follows from the inductive hypothesis that there exists a derivation of the following form:

  $$S \overset{*}{\to}_{G_{P,\mathrm{S}}} n_1 \cdots n_l N_i \to_{G_{P,\mathrm{S}}} n_1 \cdots n_l n_i.$$

  Hence, the following derivation also exists.

$$\begin{array}{lll} S & \overset{*}{\to}_{G_{P,\mathrm{S}}} & n_1 \cdots n_l N_i \\ & \to_{G_{P,\mathrm{S}}} & n_1 \cdots n_l N_k \quad \text{by } (5.7) \\ & \to_{G_{P,\mathrm{S}}} & n_1 \cdots n_l n_k. \quad \text{by } (5.5) \end{array}$$

□

Theorem 5.1.3 *Let $P = (NO, IS, IT, TG, CG)$ be a program in $\Pi_{\text{check-free}}$. The verification problem for P and a verification property specified by a DFA M_ψ is solvable in $O((|NO| + |TG| + |CG|) \cdot \#M_\psi)$ time.*

Proof. From Theorem 5.1.2, we can construct a regular grammar G_P such that $L(G_P) = [P] \cap \overline{L(M_\psi)}$ and $\|G_P\|$ is $O((|NO| + |TG| + |CG|) \cdot \#M_\psi)$. The emptiness problem for the language generated by a regular grammar G is solvable in $O(\|G\|)$ time. Hence, we can decide whether $L(G_P) = \emptyset$ in $O((|NO| + |TG| + |CG|) \cdot \#M_\psi)$ time. Furthermore, all the values of predicate CR can be determined in $O(|NO| + |TG| + |CG|)$ time by applying the inference rules (in a non-redundant way) until no value of CR for each node changes. Therefore, the verification problem is solvable in $O((|NO| + |TG| + |CG|) \cdot \#M_\psi)$ time. □

As is the case with Lemma 4.2.2, the proof of the above theorem is not valid if a verification property L_ψ is specified by an NFA instead of a DFA. Especially, the verification problem in this case can be shown to be PSPACE-complete.

Proposition 5.1.4 *Let $P = (NO, IS, IT, TG, CG)$ be a program in $\Pi_{\text{check-free}}$. The verification problem for P and a verification property specified by an NFA M_ψ is PSPACE-complete.*

Proof. Both of the following two problems are known to be PSPACE-complete[12].

FINITE AUTOMATON INEQUIVALENCE
Instance: Two NFA M_1 and M_2 having the same input alphabet Σ.
Question: $L(M_1) \neq L(M_2)$?

REGULAR EXPRESSION NON-UNIVERSALITY
Instance: A regular expression E over a finite alphabet Σ.
Question: $L(E) \neq \Sigma^*$?

Note that for a program P and a verification property L_ψ, $[P] \subseteq L_\psi$ if and only if $[[P]] \subseteq L_{\text{safe}}[\psi]$ (see section 3.3). PSPACE-solvability of the verification problem can

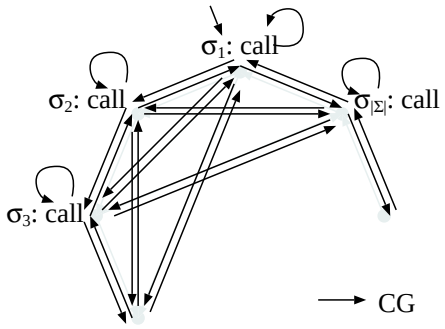


Figure 5.1. The program P_{Σ^*}

be shown by transforming the problem to the FINITE AUTOMATON INEQUIVALENCE problem. We can construct an NFA $M_{P,\mathrm{S}}$ such that $[P] = L(M_{P,\mathrm{S}})$ and $\#M_{P,\mathrm{S}} = O(|NO|)$ [25]. We can also construct an NFA $M_{P,\mathrm{S},\psi}$ such that $L(M_{P,\mathrm{S},\psi}) = L(M_{P,\mathrm{S}}) \cap L(M_\psi)$ and $\#M_{P,\mathrm{S},\psi} = \#M_{P,\mathrm{S}} \cdot \#M_\psi$. This construction of $M_{P,\mathrm{S}}$ and $M_{P,\mathrm{S},\psi}$ completes the transformation since $[P] \not\subseteq L(M_\psi)$ iff $[P] \neq [P] \cap L(M_\psi)$ iff $L(M_{P,\mathrm{S}}) \neq L(M_{P,\mathrm{S},\psi})$.

PSPACE-hardness of the verification problem can be shown by transforming REGULAR EXPRESSION NON-UNIVERSALITY to the problem. First we construct the program $P_{\Sigma^*} = (NO, IS, IT, TG, CG)$ in Figure 5.1, where the set NO of nodes is the alphabet Σ and the entry point IT is an arbitrary node $\sigma_1 \in \Sigma$. P_{Σ^*} is similar to the complete graph with $|\Sigma|$ nodes and obviously $[P_{\Sigma^*}] = \sigma_1\Sigma^*$. Second we construct an NFA M_ψ such that $L(M_\psi) = L(\sigma_1 \cdot E)$ from the regular expression E. This construction of M_ψ can be performed in polynomial time[15]. The construction of P_{Σ^*} and M_ψ completes the transformation since $[P_{\Sigma^*}] \not\subseteq L(M_\psi)$ iff $\sigma_1\Sigma^* \not\subseteq L(\sigma_1 \cdot E)$ iff $\Sigma^* \neq L(E)$. □

5.2. Programs with JDK1.2 Stack Inspection

5.2.1 Permission Based Model

As shown in chapter 4, the verification problem is computationally intractable while the problem for the subclass $\Pi_{\text{check-free}}$ is solvable in linear time in the program size. In this section, we introduce another subclass $\Pi_{\text{JDK1.2}}$ of programs which contain only check nodes equivalent to *checkPermission* in JDK1.2. $\Pi_{\text{JDK1.2}}$ properly includes

$\Pi_{\text{check-free}}$. Also we show that the verification problem for class $\Pi_{\text{JDK1.2}}$ is solvable in linear time in the size of a program by reducing the problem for $\Pi_{\text{JDK1.2}}$ to the problem for $\Pi_{\text{check-free}}$.

In JDK1.2, an access to a resource is controlled by inspecting the current contents of the stack. This mechanism can be implemented as follows (called *eager evaluation* in section 2.4 of [13]).

- Assume that a set of permissions is granted to each method and every node in a method has the permissions granted to that method. Also assume that at each state of a program, the control keeps the set of effective permissions. The set of effective permissions is updated as follows when a method invocation or a return occurs.

- When a method m_2 is invoked from a method m_1 and the invocation is not privileged, then the set of effective permissions becomes the intersection of the current set and the set of permissions granted to m_2.

- When method m_2 is invoked from method m_1 and the invocation is privileged, then the set of effective permissions becomes the intersection of the sets of permissions granted to m_1 and granted to m_2.

- An access is controlled by inspecting the current set of effective permissions instead of by inspecting the contents of the stack.

From a program $P_{\text{JDK1.2}}$ in $\Pi_{\text{JDK1.2}}$, we can construct an equivalent program $\widehat{P}$ where each node is a pair of a node of $P_{\text{JDK1.2}}$ and the set of effective permissions at that node. For every check node of $P_{\text{JDK1.2}}$, using the set of effective permissions in the node, we can statically know the result of the execution of the check node. Therefore, by removing all transfer edges emitted from the check nodes at which the execution is aborted and by replacing all check nodes with $\mathit{check}(NO^*)$, we can obtain an equivalent program $\widehat{P}$ which belongs to $\Pi_{\text{check-free}}$. By Theorem 5.1.3, the verification problem can be solved in linear time in the size of $\widehat{P}$.

For example, from the program shown in figure 2.1, we can construct an equivalent program shown in figure 5.2, where each node is labeled with the set of effective permissions. Since there are two paths in figure 2.1 from the initial node to the *debit* method and the sets of effective permissions at the entry point of the *debit* method are

different according to the selected path, the *debit* method is duplicated in figure 5.2. Since the execution of node $n_7^{\emptyset}$ does not succeed ($p_{debit} \notin \emptyset$), transfer edge from $n_7^{\emptyset}$ to $n_8^{\emptyset}$ is removed. The details of this construction are described as follows.

Let *PRM* be a finite set of access permissions. A program $P_{\text{JDK1.2}}$ in $\Pi_{\text{JDK1.2}}$ is a 7-tuple $P_{\text{JDK1.2}} = (NO, IS, IT, TG, CG, P\_BY, PRV)$. The first five components (NO, IS, IT, TG, CG) are the same as those defined in section 2.1. The last two components are:

$$P\_BY : NO \to 2^{PRM}$$
$$PRV \subseteq \{n \mid n \in NO,\ IS(n) = call\}.$$

$P\_BY(n)$ (possessed by n) is the set of permissions which a node n has. In this model, we assume that all nodes in a method have the same set of permissions, that is,

$$n \overset{TG}{\to} n' \Rightarrow P\_BY(n) = P\_BY(n').$$

The set $N(p)$ of nodes which have a permission p is

$$N(p) = \{n \mid p \in P\_BY(n)\}.$$

PRV is the set of privileged nodes. As a check node, only $check(JDK(p))$ is allowed, where $p \in PRM$. Recall that $JDK(p)$ is represented as:

$$JDK(p) = (NO^*(PRV \cap N(p)) \cup \varepsilon)(N(p))^*. \tag{5.12}$$

We present a transformation from a given program in $\Pi_{\text{JDK1.2}}$ to a program in $\Pi_{\text{check-free}}$.

Construction 5.2.1
Input: a program $P_{\text{JDK1.2}} = (NO, IS, IT, TG, CG, P\_BY, PRV)$.
Output: the program $\widehat{P} = (\widehat{NO}, \widehat{IS}, \widehat{IT}, \widehat{TG}, \widehat{CG})$ in $\Pi_{\text{check-free}}$ where:

(1) $\widehat{NO} = NO \times 2^{PRM}$. An element of $\widehat{NO}$ is represented as n^{P_i} $(n \in NO,\ P_i \subseteq PRM)$.

(2) For arbitrary $n \in NO$ and $P_i \subseteq PRM$,

(2.1) $IS(n) = call \Rightarrow \widehat{IS}(n^{P_i}) = call$,

(2.2) $IS(n) = return \Rightarrow \widehat{IS}(n^{P_i}) = return$,

(2.3) $IS(n) = check(JDK(p)) \Rightarrow \widehat{IS}(n^{P_i}) = check(\widehat{NO}^*)$.

(3) $\widehat{IT} = IT^{P\_BY(IT)}$.

(4) $\widehat{TG}$ and $\widehat{CG}$ are defined as follows. For an arbitrary $P_i \subseteq PRM$,

(4.1) $n_i \xrightarrow{CG} n_j \Rightarrow n_i^{P_i} \xrightarrow{\widehat{CG}} n_j^{P_j}$ where

$$P_j = \begin{cases} P_i \cap P\_BY(n_j) & n_i \notin PRV, \\ P\_BY(n_i) \cap P\_BY(n_j) & n_i \in PRV, \end{cases}$$

(4.2) $n_i \xrightarrow{TG} n_j$ and $IS(n_i) = call \Rightarrow n_i^{P_i} \xrightarrow{\widehat{TG}} n_j^{P_i}$,

(4.3) $n_i \xrightarrow{TG} n_j$, $IS(n_i) = check(JDK(p))$ and $p \in P_i \Rightarrow n_i^{P_i} \xrightarrow{\widehat{TG}} n_j^{P_i}$.

□

Note that program $\widehat{P}$ contains only $check(\widehat{NO}^*)$ as a check node and hence $\widehat{P} \in \Pi_{\text{check-free}}$. In practice, it suffices to construct the nodes reachable from the initial node $\widehat{IT}$ and the edges connecting them.

We show that the set $[\![P_{\text{JDK1.2}}]\!]$ of traces of a program $P_{\text{JDK1.2}}$ in $\Pi_{\text{JDK1.2}}$ coincides with the set $[\![\widehat{P}]\!]$ of traces of program $\widehat{P}$ (modulo the homomorphism which erases the effective permissions).

Lemma 5.2.1 *For a program $P_{\text{JDK1.2}}$ =(NO, IS, IT, TG, CG, P_BY, PRV), let $\widehat{P}$ =($\widehat{NO}$, $\widehat{IS}$, $\widehat{IT}$, $\widehat{TG}$, $\widehat{CG}$) be the program obtained from $P_{\text{JDK1.2}}$ by Construction 5.2.1. Let us define the homomorphism $h : (\widehat{NO} \cup \{\triangleright\})^* \to (NO \cup \{\triangleright\})^*$ as $h(n^P) = n$ for $n^P \in \widehat{NO}$ and $h(\triangleright) = \triangleright$. Then, $[\![P_{\text{JDK1.2}}]\!] = h([\![\widehat{P}]\!])$. Note that since h is a homomorphism, for any words $a, b \in (\widehat{NO} \cup \{\triangleright\})^*$, $h(ab) = h(a)h(b)$ holds.*

Proof. It suffices to show that $IT \triangleright \cdots \triangleright n_1 n_2 \cdots n_k \in [\![P_{\text{JDK1.2}}]\!]$ *if and only if* $\widehat{IT} \triangleright \cdots \triangleright n_1^{P_1} n_2^{P_2} \cdots n_k^{P_k} \in [\![\widehat{P}]\!]$ where $P_1 = P\_BY(n_1)$ and

$$P_i = \begin{cases} P_{i-1} \cap P\_BY(n_i) & n_{i-1} \notin PRV \\ P\_BY(n_{i-1}) \cap P\_BY(n_i) & n_{i-1} \in PRV \end{cases}$$

for each $1 < i \leq k$.

The *only if* part is shown by induction on the length of a trace of $P_{\text{JDK1.2}}$.

(**basis**) Clearly, $\langle IT \rangle \in [\![P_{\text{JDK1.2}}]\!]$ and
$\langle IT^{P\_BY(IT)} \rangle \in [\![\widehat{P}]\!]$.

(**inductive step**) There are three cases to consider.

Case 1: $IT \triangleright \cdots \triangleright n_1 \cdots n_{k-1} \triangleright n_1 \cdots n_{k-1} n_k \in [\![P_{\mathrm{JDK1.2}}]\!]$, $IS(n_{k-1}) = call$ and $n_{k-1} \overset{CG}{\rightarrow} n_k$. By the induction hypothesis, we can see that

$$\widehat{IT} \triangleright \cdots \triangleright n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} \in [\![\widehat{P}]\!],$$

where

$$\underset{(1 \leq i \leq k-1)}{P_i} = \begin{cases} P_{i-1} \cap P\_BY(n_i) & n_{i-1} \notin PRV, \\ P\_BY(n_{i-1}) \cap P\_BY(n_i) & n_{i-1} \in PRV. \end{cases}$$

By the definition of $\widehat{CG}$,

$$n_{k-1}^{P_{k-1}} \xrightarrow{\widehat{CG}} n_k^{P_k},$$

where

$$P_k = \begin{cases} P_{k-1} \cap P\_BY(n_k) & n_{k-1} \notin PRV, \\ P\_BY(n_{k-1}) \cap P\_BY(n_k) & n_{k-1} \in PRV. \end{cases}$$

Hence,

$$\widehat{IT} \triangleright \cdots \triangleright n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} \triangleright n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} n_k^{P_k} \in [\![\widehat{P}]\!].$$

Case 2: $IT \triangleright \cdots \triangleright n_1 \cdots n_{k-1} n_k n_{k+1} \triangleright n_1 \cdots n_{k-1} n'_k \in [\![P_{\mathrm{JDK1.2}}]\!]$, $IS(n_{k+1}) = return$ and $n_k \overset{TG}{\rightarrow} n'_k$. The induction hypothesis implies that $\widehat{IT} \triangleright \cdots \triangleright n_1^{P_1} \cdots n_k^{P_k} n_{k+1}^{P_{k+1}} \in [\![\widehat{P}]\!]$. By the definition of $\widehat{TG}$, there is a node $n_k'^{P_k}$ such that $n_k^{P_k} \xrightarrow{\widehat{TG}} n_k'^{P_k}$. Hence,

$$\widehat{IT} \triangleright \cdots \triangleright n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} n_k^{P_k} n_{k+1}^{P_{k+1}} \triangleright n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} n_k'^{P_k} \in [\![\widehat{P}]\!].$$

Case 3: $IT \triangleright \cdots \triangleright n_1 \cdots n_{k-1} n_k \triangleright n_1 \cdots n_{k-1} n'_k \in [\![P_{\mathrm{JDK1.2}}]\!]$, $IS(n_k) = check(JDK(p))$ and $n_k \overset{TG}{\rightarrow} n'_k$. By the induction hypothesis, we can see that $\widehat{IT} \triangleright \cdots \triangleright n_1^{P_1} \cdots n_k^{P_k} \in [\![\widehat{P}]\!]$. From the condition among $P_1, \cdots, P_k$ and $n_1 \cdots n_k \in JDK(L[p])$, $p \in P_k$. Therefore, by the definition of $\widehat{TG}$, there is $n_k^{P_k} \xrightarrow{\widehat{TG}} n_k'^{P_k}$. Hence,

$$\widehat{IT} \triangleright \cdots \triangleright n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} n_k^{P_k} \triangleright n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} n_k'^{P_k} \in [\![\widehat{P}]\!].$$

We will give a proof for case 3, the most difficult case. By the induction hypothesis on $IT \triangleright \cdots \triangleright n_1 \cdots n_{k-1} n_k$, we can see that $\widehat{IT} \triangleright \cdots \triangleright n_1^{P_1} \cdots n_k^{P_k} \in [\![\widehat{P}]\!]$ where $P_1 = P\_BY(n_1)$ and

$$P_i = \begin{cases} P_{i-1} \cap P\_BY(n_i) & n_{i-1} \notin PRV, \\ P\_BY(n_{i-1}) \cap P\_BY(n_i) & n_{i-1} \in PRV \end{cases} \tag{5.13}$$

for $1 < i \leq k$. Hence, it suffices to show that

$$n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} n_k^{P_k} \rhd n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} {n'_k}^{P_k}. \tag{5.14}$$

It follows from $n_1 \cdots n_{k-1} n_k \rhd n_1 \cdots n_{k-1} n'_k$ and $IS(n_k) = check(JDK(p))$ that

$$\begin{aligned} n_1 \cdots n_{k-1} n_k \quad &\in \quad check(JDK(p)) \\ &= (NO^*(PRV \cap N(p)) \cup \varepsilon)(N(p))^*. \end{aligned} \tag{5.15}$$

There are two cases.

- Assume that $n_i \notin PRV$ for $1 \leq i < k$. By (5.13),

  $$P_k = \bigcap_{1 \leq i \leq k} P\_BY(n_i). \tag{5.16}$$

  By (5.15), $n_i \in N(p)$ for $1 \leq i \leq k$, which implies $p \in \bigcap_{1 \leq i \leq k} P\_BY(n_i)$ by the definition of $N(p)$. Hence, $p \in P_k$ by (5.16).

- Assume that there exists a node $n_j \in PRV$ $(1 \leq j < k)$ and $n_i \notin PRV$ for $j < i < k$. By (5.13),

  $$P_k = \bigcap_{j \leq i \leq k} P\_BY(n_i). \tag{5.17}$$

  By (5.15), $n_i \in N(p)$ $(j \leq i \leq k)$ and thus $p \in \bigcap_{j \leq i \leq k} P\_BY(n_i) = P_k$ by the definition of $N(p)$ and (5.17).

In either case, $p \in P_k$ holds and $n_k^{P_k} \xrightarrow{\widehat{TG}} {n'_k}^{P_k}$ is constructed by (4.3) of Construction 5.2.1. Therefore, (5.14) holds.

The *if* part can be shown by induction on the length of a trace of $\widehat{P}$.
(**basis**) The claim holds since $IT^{P\_BY(IT)} \in [\![\widehat{P}]\!]$ and $IT \in [\![P_{\mathrm{JDK1.2}}]\!]$.
(**inductive step**) Again, we will prove the claim for the most difficult case: assume that

- $\widehat{IT} \rhd \cdots \rhd n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} \rhd n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} n_k^{P_k} \in [\![\widehat{P}]\!]$, $IS(n_{k-1}^{P_{k-1}}) = call$, $n_{k-1}^{P_{k-1}} \xrightarrow{\widehat{CG}} n_k^{P_k}$.
  From the induction hypotesis, we can see $IT \rhd \cdots \rhd n_1 \cdots n_{k-1} \in [\![P_{\mathrm{JDK1.2}}]\!]$. On the other hand, from $n_{k-1}^{P_{k-1}} \xrightarrow{\widehat{CG}} n_k^{P_k}$ and the definition of $\widehat{CG}$, there must be $n_{k-1} \xrightarrow{CG} n_k$. Hence,

  $$IT \rhd \cdots \rhd n_1 \cdots n_{k-1} \rhd n_1 \cdots n_{k-1} n_k \in [\![P_{\mathrm{JDK1.2}}]\!].$$

- $\widehat{IT} \triangleright \cdots \triangleright n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} n_k^{P_k} n_{k+1}^{P_{k+1}} \triangleright n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} n_k'^{P_k} \in [\![\widehat{P}]\!]$, $IS(n_{k+1}^{P_{k+1}}) = return$, $n_k^{P_k} \xrightarrow{\widehat{TG}} n_k'^{P_k}$. (In this case, $IS(n_k^{P_k}) = call$.)
  From the induction hypothesis, we can see that $IT \triangleright \cdots \triangleright n_1 \cdots n_k n_{k+1} \in [\![P_{\mathrm{JDK1.2}}]\!]$. On the other hand from $n_k^{P_k} \xrightarrow{\widehat{TG}} n_k'^{P_k}$ and the definition of $\widehat{TG}$, there must be $n_k \xrightarrow{TG} n_k'$. Hence,

$$IT \triangleright \cdots \triangleright n_1 \cdots n_{k-1} n_k n_{k+1} \triangleright n_1 \cdots n_{k-1} n_k' \in [\![P_{\mathrm{JDK1.2}}]\!].$$

- $\widehat{IT} \triangleright \cdots \triangleright n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} n_k^{P_k} \triangleright n_1^{P_1} \cdots n_{k-1}^{P_{k-1}} n_k'^{P_k} \in [\![\widehat{P}]\!]$, $IS(n_k^{P_k}) = check(\widehat{NO}^*)$ and $n_k^{P_k} \xrightarrow{\widehat{TG}} n_k'^{P_k}$ where $P_1 = P\_BY(n_1)$ and (5.13) holds. Note that $n_k^{P_k} \xrightarrow{\widehat{TG}} n_k'^{P_k'}$ exists only if $P_k = P_k'$ by (4.3) in Construction 5.2.1. By the induction hypothesis, $IT \triangleright \cdots \triangleright n_1 \cdots n_k \in [\![P_{\mathrm{JDK1.2}}]\!]$. Since the transfer edge $n_k^{P_k} \xrightarrow{\widehat{TG}} n_k'^{P_k}$ exists, and the edge $n_k \xrightarrow{TG} n_k'$ also exists, $IS(n_k) = check(JDK(p))$ and $p \in P_k$ for some $p \in PRM$. If we can prove (5.15), then $n_1 \cdots n_{k-1} n_k \triangleright n_1 \cdots n_{k-1} n_k'$ and the claim holds.

  - Assume that $n_i \notin PRV$ for $1 \leq i < k$. Then (5.13) implies (5.16). Since $p \in P_k$, we know that $p \in \bigcap_{1 \leq i \leq k} P\_BY(n_i)$, which implies $n_i \in N(p)$ for $1 \leq i \leq k$. Therefore, (5.15) holds.
  - Assume that there exists a node $n_j \in PRV\,(1 \leq j < k)$ and $n_i \notin PRV$ for $j < i < k$. By (5.13), we have (5.17). Since $p \in P_k$, (5.17) implies $n_i \in N(p)$ for $j \leq i \leq k$, which together with $n_j \in PRV$ implies (5.15). By (5.15), $n_i \in L[p] (j \leq i \leq k)$ and thus $p \in \bigcap_{j \leq i \leq k} PBY n_i = P_k$ by the definition of $L[p]$ and (5.17).

  By the condition among $P_1, \cdots, P_k$, $n_k^{P_k} \xrightarrow{\widehat{TG}} n_k'^{P_k}$ and the definition of $\widehat{TG}$, we can see that $IS(n_k) = check(JDK(L[p]))$, $p \in P_k$ and $n_k \xrightarrow{TG} n_k'$. On the other hand, from

$$P_k = \begin{cases} P_{k-1} \cap P\_BY(n_k) & n_{k-1} \notin PRV \\ P\_BY(n_{k-1}) \cap P\_BY(n_k) & n_{k-1} \in PRV \end{cases},$$

  $n_1 \cdots n_{k-1} n_k \in JDK(p)$. Hence,

$$IT \triangleright \cdots \triangleright n_1 \cdots n_{k-1} n_k \triangleright n_1 \cdots n_{k-1} n_k' \in [\![P_{JDK}]\!].$$

Therefore, $[\![P_{\text{JDK1.2}}]\!] = h([\![\widehat{P}]\!])$ holds by the definition of h. □

Theorem 5.2.2 *The verification problem for a program $P_{\text{JDK1.2}}$ in $\Pi_{\text{JDK1.2}}$ and a verification property specified by a DFA M_ψ is solvable in linear time in the program size $||P_{\text{JDK1.2}}||$ and the number $\sharp M_\psi$ of states of M_ψ.*

Proof. Let $P_{\text{JDK1.2}} = (NO, IS, IT, TG, CG, P\_BY, PRV)$ be a program in $\Pi_{\text{JDK1.2}}$ and let $\widehat{P}$ be the program in $\Pi_{\text{check-free}}$ obtained from $P_{\text{JDK1.2}}$ by Construction 5.2.1. Solving the verification problem for $P_{\text{JDK1.2}}$ and $L(M_\psi)$ is equivalent to deciding whether $[P_{\text{JDK1.2}}] \subseteq L(M_\psi)$ or not (see section 3.3). By Lemma 5.2.1, this decision is further equivalent to deciding whether $h([\widehat{P}]) \subseteq L(M_\psi)$. $\widehat{P}$ belongs to
$\Pi_{\text{check-free}}$ and the class of regular languages is closed under homomorphism (for any regular grammar G, a regular grammar G' can be constructed such that $L(G') = h(L(G))$ and $||G|| = O(||G||)$). Hence, by Theorem 5.1.3, the verification problem for $P_{\text{JDK1.2}}$ and $L(M_\psi)$ is solvable in polynomial time in $||\widehat{P}||$ and $\sharp M_\psi$. More specifically, the problem is solvable in $O((|\widehat{NO}| + |\widehat{TG}| + |\widehat{CG}|) \cdot \sharp M_\psi)$ time. By Lemma 5.2.1, solving the verification problem for $P_{\text{JDK1.2}}$ and $L_\psi = L(M_\psi)$ is equivalent to deciding whether $h([\![\widehat{P}]\!]) \subseteq L_{\text{safe}}[\psi]$. The latter can be shown to be decidable in $O((|\widehat{NO}| + |\widehat{TG}| + |\widehat{CG}|) \cdot \sharp M_\psi)$ time by slightly modifying the algorithm used in the proof of Theorem 5.1.3 so that the effect of the homomorphism h can be incorporated into the algorithm. Since $|\widehat{NO}|$, $|\widehat{TG}|$ and $|\widehat{CG}|$ are $O(|NO| \cdot 2^{|PRM|})$, $O(|TG| \cdot 2^{|PRM|})$ and $O(|CG| \cdot 2^{|PRM|})$ respectively, the time complexity is $O((|NO| + |TG| + |CG|) \cdot \sharp M_\psi \cdot 2^{|PRM|})$. □

Proposition 5.2.3 *The verification problem for a program $P_{\text{JDK1.2}}$ in $\Pi_{\text{JDK1.2}}$ and a verification property specified by a DFA M_ψ is PSPACE-hard.*

Proof. We transform QUANTIFIED 3-SATISFIABILITY (QUANTIFIED 3SAT) problem to the verification problem. An instance of QUANTIFIED 3SAT is a Boolean formula $F = (Q_1 x_1)(Q_2 x_2) \ldots (Q_n x_n) E$ where E is a conjunction of 3-literal disjunctive clauses involving the variables $x_1, x_2, \ldots, x_n$ and each Q_i is either "$\exists$" or "$\forall$."

The program P_{Q3SAT} constructed by the transformation consists of three parts: *header*, *truth assignment*, and *satisfaction check* (figure 5.3). A header consists of two nodes n_{s} and n_{t}. A truth assignment consists of $x_{i,\text{F}}$ and $x_{i,\text{T}}$ for $1 \leq i \leq n$. We

consider that "false" is assigned to x_i when the control passes through $x_{i,\mathrm{F}}$ and "true" is assigned to x_i when the control passes through $x_{i,\mathrm{T}}$. We put a call edge between $x_{i-1,V}$ and $x_{i,V'}$ for every $V,V' \in \{\mathrm{F},\mathrm{T}\}$ if Q_i is "$\exists$" (figure 5.4). That is, in this case the program tries assigning one of the truth values to x_i. If Q_i is "$\forall$," we put a call edge between $x_{i-1,V}$ and $y_{i,\mathrm{F}}$ for each $V \in \{\mathrm{F},\mathrm{T}\}$ and put a transfer edge from $y_{i,\mathrm{F}}$ to $y_{i,\mathrm{T}}$. We also put a call edge between $y_{i,V}$ and $x_{i,V}$ for each $V \in \{\mathrm{F},\mathrm{T}\}$. The program tries assigning each truth value to x_i in this case.

Let $PRM = \{p_1,\ldots,p_n,\overline{p_1},\ldots,\overline{p_n}\}$, and each method is granted to a subset of permissions as figures 5.3 and 5.4.

Suppose $E = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ and $C_i = u_{i1} \vee u_{i2} \vee u_{i3}$ for $1 \leq i \leq m$, where u_{ij} is a literal over $\{x_1,\ldots,x_n\}$. A satisfaction check consists of $check(L_{ij})$ for $1 \leq i \leq m$ and $1 \leq j \leq 3$, where

$$L_{ij} = \begin{cases} JDK(\overline{p_k}) & \text{if } u_{ij} = \overline{x_k}, \\ JDK(p_k) & \text{if } u_{ij} = x_k. \end{cases}$$

When the control reaches an arbitrary node in a satisfaction check, the contents s of stack contains either $x_{i,\mathrm{F}}$ or $x_{i,\mathrm{T}}$ for all i. If $x_{i,\mathrm{F}}$ is contained in s, then $s \in JDK(\overline{p_k})$ and $s \notin JDK(p_k)$. If $x_{i,\mathrm{T}}$ is contained in s, then $s \in JDK(p_k)$ and $s \notin JDK(\overline{p_k})$. Hence, the control can reach the return node r_t if the current truth assignment satisfies E.

Therefore, an execution of P_{Q3SAT} reaches the node n_t if and only if F is true. That is, $[\![P_{\mathrm{Q3SAT}}]\!] \not\subseteq L_{\mathrm{safe}}[\psi]$ for $L_\psi = (NO - \{n_\mathrm{t}\})^*$ if and only if F is true. □

Table 5.1. Complexity of the verification problem

		verification property	
		NFA, RE	DFA, DFA^R
language in a check node	NFA, RE, DFA^R, DFA	in double exponential time DEXP-POLY time-hard	DEXP-POLY time-complete
$\Pi_{\mathrm{JDK1.2}}$	general case	PSPACE-hard	PSPACE-hard
	$min(\lvert PRM\rvert, \lvert DOM\rvert) \leq O(\log \lvert\lvert P\rvert\rvert)$	PSPACE-complete	PTIME
$\Pi_{\text{check-free}}$		PSPACE-complete	PTIME

5.2.2 Domain Based Model

In Construction 5.2.1, we let each node of $\widehat{P}$ be a pair of a node of $P_{\text{JDK1.2}}$ and the set of effective permissions, and thus the size of $\widehat{P}$ is $O(\|P_{\text{JDK1.2}}\| \cdot 2^{|PRM|})$. In this chapter we show an alternative way of constructing $\widehat{P}$ from $P_{\text{JDK1.2}}$ where each node of $\widehat{P}$ is a pair of a node of $P_{\text{JDK1.2}}$ and a subset of protection domains; then the size of $\widehat{P}$ is $O(\|P_{\text{JDK1.2}}\| \cdot 2^{|DOM|})$ where DOM is the set of protection domains. We refer to this construction algorithm as the *domain based construction* of $\widehat{P}$. Practically, the number of protection domains is often smaller than the number of permissions, in which case the domain based construction is more efficient than Construction 5.2.1.

Below we describe the domain based construction of $\widehat{P}$. In JDK1.2, each method belongs to exactly one protection domain (or for short, domain) and each domain is granted a set of permissions. Let $P_{\text{JDK1.2}} = (NO, IS, IT, TG, CG, P\_BY, PRV)$ be a program in $\Pi_{\text{JDK1.2}}$ and let DOM a finite set of domains. Assume that the domain which a node $n \in NO$ belongs to is given and is denoted by $D\_OF(n)$ ($\in DOM$). The set of domains which is granted a permission p is also given and is denoted by $GRNT(p)$ ($\subseteq DOM$). We assume that every node in a method belongs to the same domain, that is,

$$n \xrightarrow{TG} n' \Rightarrow D\_OF(n) = D\_OF(n').$$

(Note that $P\_BY(n) = \{p \mid D\_OF(n) \in GRNT(p)\}$ should hold and thus this model does not conflict with the assumptions in chapter 5.2.1.) The access control of JDK1.2 can be achieved by inspecting whether the domain of every node in the stack (precisely, every node which is not a proper ancestor of any privileged node) is granted a specified permission. The domain based construction of a program $\widehat{P} = (\widehat{NO}, \widehat{IS}, \widehat{IT}, \widehat{TG}, \widehat{CG})$ in $\Pi_{\text{check-free}}$ from $P_{\text{JDK1.2}}$ is the same as Construction 5.2.1, but we let each node of $\widehat{P}$ be augmented by the set of domains of nodes in the stack instead of the set of effective permissions. The differences between this construction and Construction 5.2.1 are:

(1) $\widehat{NO} = NO \times 2^{DOM}$,

(3) $\widehat{IT} = IT^{\{D\_OF(IT)\}}$,

(4.1) $n_i \xrightarrow{CG} n_j \Rightarrow n_i^{D_i} \xrightarrow{\widehat{CG}} n_j^{D_j}$ where

$$D_j = \begin{cases} D_i \cup \{D\_OF(n_j)\} & \text{if } n_i \notin PRV, \\ \{D\_OF(n_i), D\_OF(n_j)\} & \text{if } n_i \in PRV, \end{cases}$$

(4.3) $n_i \xrightarrow{TG} n_j$, $IS(n_i) = check(JDK(p))$ and $D_i \subseteq GRNT(p) \Rightarrow n_i^{D_i} \xrightarrow{\widehat{TG}} n_j^{D_i}$.

The equality between $\widehat{P}$ and $P_{\mathrm{JDK1.2}}$ can be shown in the same way as shown in Lemma 5.2.1.

Although so far we assumed that DOM, D_OF and $GRNT$ are given, we can derive them from P_BY as follows: Define DOM as the set of the equivalence classes on NO defined by the relation $\sim$ such that $n \sim n' \Leftrightarrow P\_BY(n) = P\_BY(n')$, $D\_OF(n) = \{n' \mid P\_BY(n) = P\_BY(n')\}$ and $GRNT(p) = \{D\_OF(n) \mid p \in P\_BY(n)\}$. DOM, D_OF and $GRNT$ can be calculated by the following algorithm.

(1) Let $GRNT(p) = \emptyset$ for each $p \in PRM$ and let $DOM = \emptyset$.

(2) For each $n \in NO$:

   (a) If there exists $d \in DOM$ such that $p \in P\_BY(n) \Leftrightarrow d \in GRNT(p)$ for every $p \in PRM$, then let $D\_OF(n) = d$.

   (b) Otherwise, add a new element d to DOM, add d to $GRNT(p)$ for each $p \in P\_BY(n)$ and let $D\_OF(n) = d$.

This algorithm runs in $O(|NO| \cdot \min(|NO|, 2^{|PRM|}) \cdot |PRM|)$ time. Note that the cardinality of DOM in this definition is not larger than that of any given set of protection domains (there exists at most one protection domain which is granted a given subset of permissions in this definition).

By Theorem 5.2.2, the verification problem for a program $P_{\mathrm{JDK1.2}}$ in $\Pi_{\mathrm{JDK1.2}}$ and a verification property specified by a DFA M_ψ is solvable in $O(\|P_{\mathrm{JDK1.2}}\| \cdot \sharp M_\psi \cdot 2^{|PRM|})$ by Construction 5.2.1, and in $O(\|P_{\mathrm{JDK1.2}}\| \cdot \sharp M_\psi \cdot 2^{|DOM|})$ by the domain based construction. We can choose one of these algorithms based on whether $|PRM| < |DOM|$.

The results on the complexity of the verification problem shown in chapters 4 and 5 are summarized in Table 5.1.

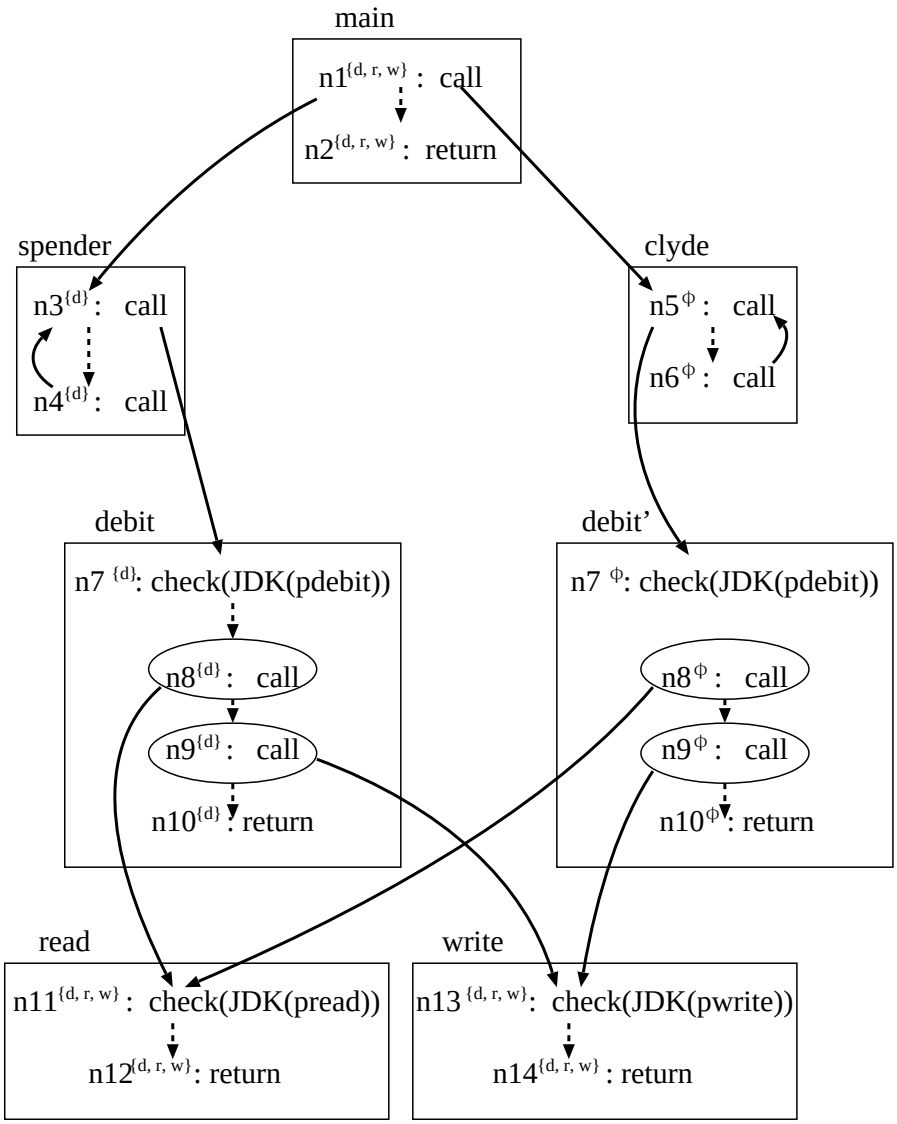


Figure 5.2. An equivalent program in $\Pi_{\text{check-free}}$

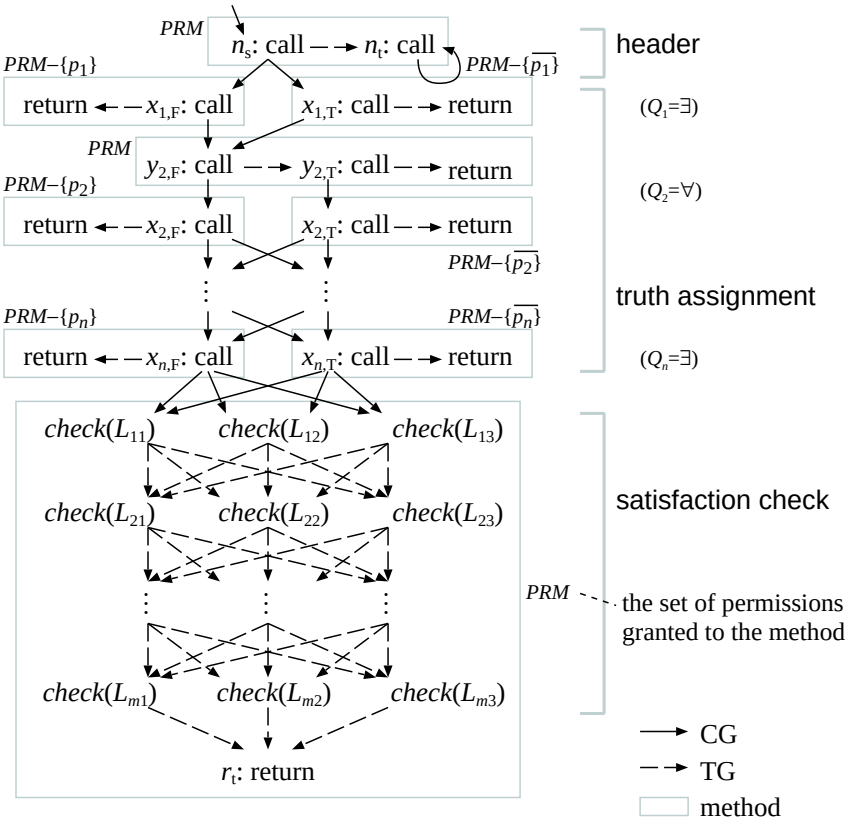
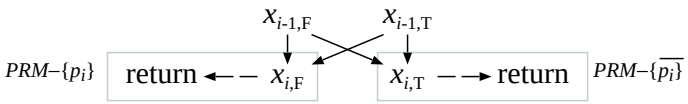
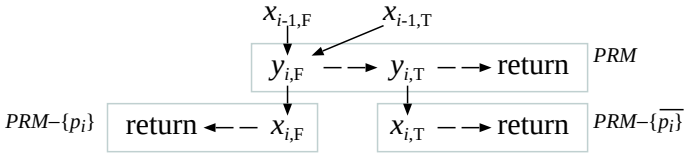


Figure 5.3. The program P_{Q3SAT} to solve QUANTIFIED 3SAT



(a) Q_i=∃



(b) Q_i=∀

Figure 5.4. Edges in the truth assignment-part of P_{Q3SAT}

Chapter 6

Evaluation of the Verification Method

6.1. Implementation

In the previous chapter, we introduced a subclass $\Pi_{\text{JDK1.2}}$ of programs and showed that the verification problem for a program in $\Pi_{\text{JDK1.2}}$ is solvable in linear time in the program size. However, the worst case time complexity of the algorithm is exponential to the number of permissions or the number of protection domains. To estimate the actual computation time needed to solve the verification problem for a real-world program, we implemented a verification system based on the proposed algorithm and measured the computation time in the system. The verification system is implemented in Java. An input to the system is a pair of a program $P_{\text{JDK1.2}}$ in $\Pi_{\text{JDK1.2}}$ and a DFA M_ψ which specifies a verification property L_ψ (*i.e.*, $L_\psi = L(M_\psi)$). The system mainly consists of the following classes: *FlowGraphJDK1_2*, *FlowGraphCheckFree*, *NFA* and *DFA*, which represent $\Pi_{\text{JDK1.2}}$, $\Pi_{\text{check-free}}$, the class of NFAs and the class of DFAs, respectively. This system performs the verification procedure in the following three steps.

(Step 1) $P_{\text{JDK1.2}}$ is converted into a program $\widehat{P}$ in $\Pi_{\text{check-free}}$ by Construction 5.2.1.

(Step 2) An NFA $M_{\widehat{P}}$ such that $L(M_{\widehat{P}}) = h([\widehat{P}]) = [P_{\text{JDK1.2}}]$ is generated from $\widehat{P}$ according to the method proposed (see Theorem 5.1.3 in this paper) where h is the homomorphism defined in Lemma 5.2.1.

(Step 3) Whether $[P_{\text{JDK1.2}}] \subseteq L_\psi$ holds or not is decided by examining whether an

equivalent condition $L(M_{\widehat{P}}) \cap \overline{L(M_{\Psi})} = \emptyset$ holds.

6.2. Experiments

We measured the computation time in the verification system for the following two programs in $\Pi_{\text{JDK1.2}}$.

$P_1(k)$**:** The on-line banking program in Example 2.1 has three permissions. We extend this program to have $3 \times k$ permissions by copying *read*, *write* and *debit* methods $k-1$ times (program $P_1(k)$). $P_1(k)$ models a part of an integrated on-line banking system supplied by k banks, which is considered as a real-world program with many permissions.

$P_2(k)$**:** We also consider a worst case program $P_2(k) = (NO, IS, IT, TG, CG, P\_BY, PRV)$, where $NO = \{n_0, n_1, \cdots, n_k\}$, $IS(n) = call$ for each $n \in NO$, $IT = n_0$, $TG = \emptyset$, $n_0 \stackrel{CG}{\rightarrow} n_i (1 \leq i \leq k), n_i \stackrel{CG}{\rightarrow} n_j (1 \leq i, j \leq k, i \neq j)$, $P\_BY(n_0) = \{p_0, \cdots, p_k\}$, $P\_BY(n_i) = \{p_1, \cdots, p_{i-1}, p_{i+1}, \cdots, p_k\}$ and $PRV = \emptyset$. In this program, the set of effective permissions at the initial node is the set of all permissions. Remark that the control can traverse the program from each node $n_i (1 \leq i \leq k)$ to any other node $n_j (1 \leq j \leq k)$ by a call edge. Also note that n_j has all permissions except p_j. Hence, when the control is at n_i with P as the current set of effective permissions, the control can reach n_i again with $P - \{p_j\}$ as the current set of effective permissions for any permission $p_j \in P$, by going through the call edge from n_i to n_j and then coming back through the call edge from n_j to n_i. Therefore, the pair of each node $n_i (1 \leq i \leq k)$ and each subset of $P\_BY(n_i)$ is created by Construction 5.2.1 and hence the size of the constructed program becomes exponential to k.

In program $P_1(k)$, let N_{clyde} be the set of nodes in method *clyde*, N_{read_i} the set of nodes in method *read* of the i-th bank, N_{write_i} the set of nodes in method *write* of the i-th bank, $N(p_{debit}) = NO - N_{clyde}$ and $E_{RW} = \bigcup_{1 \leq i \leq k} (N_{read_i} \cup N_{write_i})$. In the experiment, we specify a verification property for program $P_1(k)$ by a DFA which accepts $N(p_{debit})^* \overline{N(p_{debit})}^* \overline{E_{RW}}^*$. For program $P_2(k)$, the verification property is given by a DFA which accepts NO^*. Note that in Step 3, the algorithm searches a

path which simultaneously leads $M_{\widehat{P}}$ and $M_{\overline{\Psi}}$ to their final states where $M_{\overline{\Psi}}$ is a DFA such that $L(M_{\overline{\Psi}}) = \overline{L(M_{\Psi})}$. If such a path exists, then $L(M_{\widehat{P}}) \cap \overline{L(M_{\Psi})} \neq \emptyset$, and vice versa. Hence, the verification condition being NO^* means that there exists no path which satisfies the above condition and therefore the algorithm searches all the paths exhaustively. The profiles of the verification for $P_1(k)$ and $P_2(k)$ are summarized in Table 6.1 and the relation between computation time and the number of permissions of $P_1(k)$ ($P_2(k)$) is showed in figure 6.1 (figure 6.2).

Table 6.1. Verification profiles of example programs

		$P_1(k)$				$P_2(k)$		
		$k=5$	$k=10$	$k=20$	$k=30$	$k=3$	$k=5$	$k=7$
the number of permissions		15	30	60	90	3	5	7
P	$\lvert NO\rvert$	46	86	166	246	4	6	8
	$\lvert TG\rvert + \lvert CG\rvert$	52	97	187	277	9	25	49
$\widehat{P}$	$\lvert\widehat{NO}\rvert$	51	96	186	276	13	81	449
	$\lvert\widehat{TG}\rvert + \lvert\widehat{CG}\rvert$	52	97	187	277	27	325	2695
M_{Ψ}	$\#M_{\Psi}$	2	2	2	2	1	1	1
computation time (m sec) †	Step 1	62	208.3	1577	5662	25.3	419	137700
	Step 2	17.3	47.3	297.3	902.3	3.7	40.3	1883
	Step 3	878.3	6984	50550	165900	15.3	2643	415100
	total	957.7	7239	52420	172400	44.3	3102	554700
verification result		*True*	*True*	*True*	*True*	*True*	*True*	*True*

†JVM build J2SDK.v.1.2.2, on DEC PersonalWorkstation 500au (Alpha21164A(500MHz), 128MB RAM)

6.3. Discussion

As shown in Table 6.1, the computation time needed to verify program $P_1(k)$ is within one minute when $k \leq 20$. On the other hand, for program $P_2(k)$, the computation time exceeds nine minutes when $k = 7$. The results in figure 6.1 and figure 6.2 suggest that for a real-world program, the size of program constructed in Construction 5.2.1 and the computation time do not increase as in the worst case program according as the number of permissions increases. On the other hand, we estimate that the number of permissions used in an ordinary network application is at most several tens, because one permission is usually related to one security class [8, 32] and users can hardly

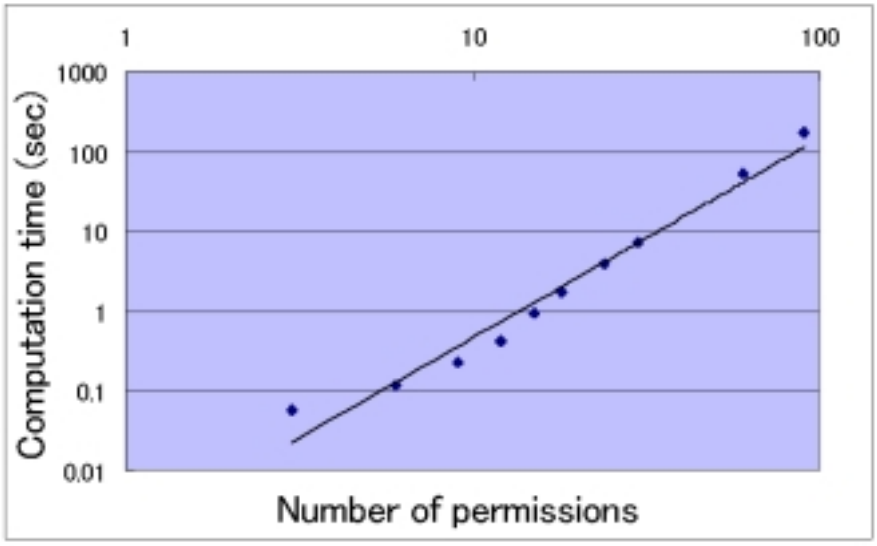


Figure 6.1. Verification time and number of permissions for $P_1(k)$

manage over thirty security classes in one application. Here, a security class denotes a set of user resources which have the same level of importance (also see Related Works). For example, the number of permissions used in the access control system of Dresdner Bank is under twenty [29]. Hence, we can conclude that for most of real-world programs, the proposed verification method is feasible.

Similarly, in the domain-based model, the time complexity of the verification is expected to be less than exponential to the number of domains for typical real-world programs. The number of domains used in an ordinary network application is also at most several tens, because one domain is related to one URL of a remote code and users can hardly manage over thirty URLs in one application. For example, the sandbox model of JDK1.0.x provides only two domains (remote code and local code). Hence, the proposed algorithm based on the domain-based model is also expected to be feasible for real-world programs.

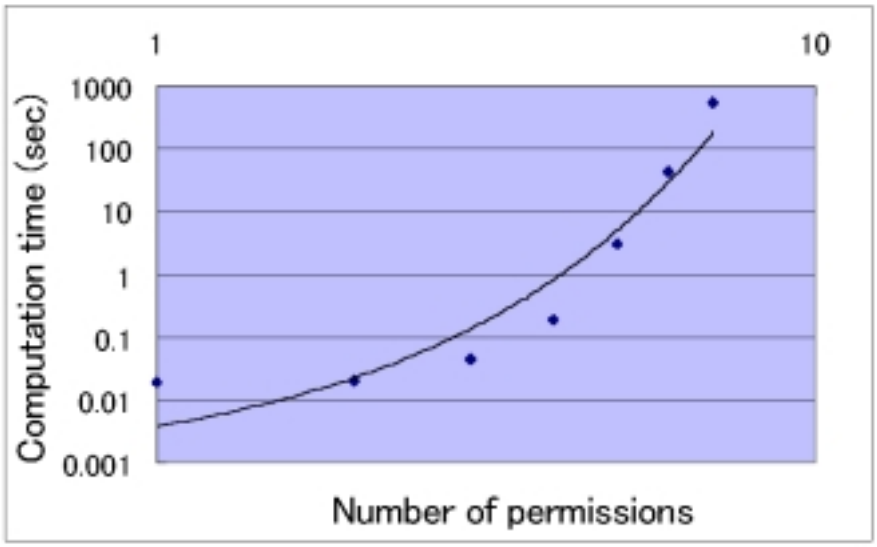


Figure 6.2. Verification time and number of permissions for $P_2(k)$

Chapter 7

Conclusion

In this thesis, we defined the verification problem of deciding whether a given program which may contain stack inspection satisfies a given security property and showed that the problem is decidable. The result is an improvement of the result in [18]. In chapter 2, flow graph based on [18] is defined as a program model. In chapter 4, we analyzed the computational complexity of the verification problem and showed that the problem is computationally intractable. For practical purposes, we proposed an efficient verification algorithm of which target programs are limited to containing only stack inspection of JDK1.2. In chapter 5, we showed that the time complexity of the verification algorithm is linear in the size of the program, however, is exponential to the number of permissions. To estimate the impact of the number of permissions on the actual computation time, we built a verification system based on the algorithm and performed experiments in the system for both a real-world program and one of the worst case programs (chapter 6). Experimental results suggest that the computation time needed to verify the real-world program is polynomial to the number of permissions. Therefore, we can conclude that the method is feasible for most of real-world programs. Closing the gap between the upperbound and the lowerbound of the problem's complexity in three cases in Table 5.1 is a future study. Proposing other efficient methods for the verification problem is another interesting question.

References

[1] M. Abadi, M. Burrows, B. Lampson and G. Plotkin: A calculus for access control in distributed systems, ACM Trans. on Prog. Lang. and Systems, 15(4), 706–734, 1993.

[2] M. Abadi, C. Fournet and G. Gonthier: Secure communications processing for distributed languages, 1999 IEEE Symp. on Security and Privacy, 74–88.

[3] A. V. Aho: Indexed grammars — An extension of context-free grammars, Journal of the ACM, 15(4), 647–671, 1968.

[4] J. Banâtre, C. Bryce and D. Le Métayer: Compile-time detection of information flow in sequential programs, 3rd ESORICS, LNCS 875, 55–73, 1994.

[5] E. Bertino and H. Weigand: An approach to authorization modeling in object-oriented database systems, Data & Knowledge Engineering, 12, 1–29, 1994.

[6] A. K. Chandra, D. C. Kozen and L. J. Stockmeyer: Alternation, Journal of the ACM, 28, 114–133, 1981.

[7] E. M. Clarke, Jr., O. Grumberg and D. A. Peled: *Model Checking*, The MIT Press, 1999.

[8] D. E. Denning: A lattice model of secure information flow, Comm. of the ACM, 19(5), 236–243, 1976.

[9] D. E. Denning and P. J. Denning: Certification of programs for secure information flow, Comm. of the ACM, 20(7), 504–513, 1977.

[10] E. A. Emerson: *Temporal and Modal Logic*, in *Handbook of Theoretical Computer Science*, 1023–1024, Elsevier, 1990.

[11] J. Esparza, A. Kučera and S. Schwoon: Model-checking LTL with regular valuations for pushdown systems, TACS 2001, LNCS 2215, 316–339.

[12] M. R. Garey and D. S. Johnson: *Computers and Intractability*, W. H. Freeman and Company, 1979.

[13] L. Gong, M. Mueller, H. Prafullchandra and R. Schemers: Going beyond the sandbox: An overview of the new security architecture in the Java™ development kit 1.2, USENIX Symp. on Internet Technologies and Systems, 103–112, 1997.

[14] N. Heintze and J. G. Riecke: The SLam calculus: Programming with secrecy and integrity, 25th ACM Symp. on Principles of Programming Languages, 365–377, 1998.

[15] J. E. Hopcroft and J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[16] Y. Ishihara, T. Morita and M. Ito: The security problem against inference attacks on object-oriented databases, IFIP TC11 WG11.3 13th Working Conf. on Database Security, 1999, published as *Research Advances in Database and Information Systems Security*, 303–316, Kluwer Academic Publ.

[17] S. Ikada, N. Nitta, Y. Takata and H. Seki: A security verification method for programs with stack inspection, Technical Report of IEICE, ISEC2000-78, 2000 (in Japanese).

[18] T. Jensen, D. Le Métayer and T. Thorn: Verification of control flow based security properties, 1999 IEEE Symp. on Security and Privacy, 89–103.

[19] X. Leroy and F. Rouaix: Security properties of typed applets, 25th ACM Symp. on Principles of Programming Languages, 391–403, 1998.

[20] A. C. Myers: JFLOW: Practical mostly-static information flow control, 26th ACM Symp. on Principles of Programming Languages, 228–241, 1999.

[21] A. C. Myers and B. Liskov: Complete, safe information flow with decentralized labels, 1998 IEEE Symp. on Security and Privacy, 186–197.

[22] N. Nitta, S. Ikada, Y. Takata and H. Seki: Decidability and complexity of the security verification problem for programs with stack inspection, Technical Report NAIST-IS-TR2001003, Nara Institute of Science and Technology, 2001.

[23] N. Nitta, Y. Takata and H. Seki: Complexity of the security verification problem for programs with stack inspection, 3rd JSSST Workshop on Programming and Programming Languages (PPL2001), 53–60, 2001.

[24] N. Nitta, Y. Takata and H. Seki: Security verification of programs with stack inspection, 6th ACM Symp. on Access Control Models and Technologies (SACMAT2001), 31–40, 2001.

[25] N. Nitta, Y. Takata and H. Seki: Decidability of the security verification problem for programs with stack inspection, IEICE Transactions on Information and Systems, to appear (in Japanese).

[26] P. Ørbæk: Can you trust your data?, TAPSOFT '95, LNCS 915, 575–589.

[27] J. Palsberg and M. I. Schwartzbach: *Object-Oriented Type Systems*, John Wiley & Sons, 1994.

[28] P. Samarati, E. Bertino, A. Ciampichetti and S. Jajodia: Information flow control in object-oriented systems, IEEE Trans. on Knowledge and Data Engneering, 9(4), 524–538, 1997.

[29] A. Schaad, J. Moffett and J. Jacob: The role-based access control system of a European Bank: A case study and discussion, 6th ACM Symp. on Access Control Models and Technologies, 3–9, 2001.

[30] G. Smith and D. Volpano: Secure information flow in a multi-threaded imperative language. 25th ACM Symp. on Principles of Programming Languages, 355–364, 1998.

[31] K. Tajima: Static detection of security flaws in object-oriented databases, 1996 ACM SIGMOD Intl. Conf. on Management of Data, 341–352.

[32] D. Volpano and G. Smith: A type-based approach to program security, TAPSOFT '97,

LNCS 1214, 607–621.

[33] D. S. Wallach and E. W. Felten: Understanding Java stack inspection, 1998 IEEE Symp. on Security and Privacy, 52–63.