NAIST-IS-DT0161014

Doctor's Thesis

Security Assurance Methods for Access Control Systems Using Static Analysis

Shigeta Kuninobu

February 6, 2004

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

Shigeta Kuninobu

Thesis committee: Hiroyuki Seki, Professor
Katsumasa Watanabe, Professor
Katsuro Inoue, Professor

Security Assurance Methods for Access Control Systems Using Static Analysis*

Shigeta Kuninobu

Abstract

As computer networks grow rapidly, it becomes more important to assure the security of a computer system against attacks by malicious users. Access control, as well as cryptography, is one of the fundamental technologies for establishing computer security. Access control mechanisms are roughly classified into mandatory access control (MAC) and discretionary access control (DAC). In MAC, a security level (e.g., top-secret, secret and unclassified) is assigned to each user and data as a security policy and access control is performed based on the distance between the security level assigned to a user who requests an access to a data and the security level assigned to the data. In DAC, the owner of each data freely assigns a security policy (permission) to the data. Unix file permission is a typical example of DAC. In both of MAC and DAC, however, it is difficult to manually check whether access control under given security policies satisfies the security goals which the whole system is required to achieve.

In this thesis, we propose static analysis methods which check whether the whole system behavior determined by access control based on given security policies satisfies security goals of the system.

In Chapter 2, an information flow analysis algorithm for a procedural program is proposed as a security assurance method for MAC based system. This algorithm infers the security level of an output of a given program relative to the security levels of program input. In the proposed method, a structure of security levels is formally represented as an arbitrary finite lattice. Adopting abstract interpretation as an analysis method enables us to infer the information flow of an arbitrary recursive program. Soundness of the proposed algorithm is formally proved and complexity of the algorithm is shown to be cubic time in the size of a program. Furthermore, the algorithm is extended so that operations (such as an encryption function) which hide information on

*Doctor's Thesis, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DT0161014, February 6, 2004.

their arguments can be appropriately modeled by using a congruence relation. Analysis results by using a prototype system are also presented.

In Chapter 3, a security assurance method for a DAC based system is proposed. First, a simple but useful policy description language is proposed. In a conventional DAC model, only permission (positive authorization) and prohibition (negative authorization) can be specified. However, an obligation policy is also important, which describes that a specified subject is obliged to perform a specified action when a certain event occurs under a certain condition. The proposed language can specify not only permission and prohibition policies but also obligation policies. A policy controlled system (PCS) is a system in which each object has its own security policy and objects' behaviors are autonomously controlled based on those policies when they interact with one another. Operational semantics of PCS is formally defined in Chapter 3. Next, the safety verification problem is defined as the problem to decide for a given PCS and a security goal (also called a safety property) whether every reachable state of the PCS satisfies the safety property. To solve this problem, we use a model checking method for pushdown system. An automatic verification tool has been implemented, and effectiveness of the proposed method is shown through verification of a sample PCS.

Keywords:

access control, policy, information flow analysis, security verification, model checking

静的解析を用いたアクセス制御システムの セキュリティ保証技術に関する研究*

國信 茂太

内容梗概

近年, コンピュータネットワークの普及により, 不正な攻撃者に対する計算機システムのセキュリティ保全が重要な課題となっている. セキュリティ保全の基本的な技術として, 暗号とともにアクセス制御法が古くから研究されている. アクセス制御法は, 必須アクセス制御 (Mandatory Access Control, MAC) と任意アクセス制御 (Discretionary Access Control, DAC) に大別される. MACでは, セキュリティレベル (top-secret, secret, unclassified 等) が各ユーザおよび各データにセキュリティポリシーとして割り振られており, アクセス制御はユーザのセキュリティレベルとそのユーザがアクセスしようとするデータのセキュリティレベルを比較することにより実行される. DACでは, 各データの所有者がそのデータに対し, 自由にセキュリティポリシー (アクセス条件) を設定することができ, アクセス制御はアクセスされるデータのセキュリティポリシーに基づいて実行される. UNIX のファイルパーミッションは DAC の典型的な例である. しかしながら, DAC/MAC のどちらのアクセス制御法においても, 与えられたセキュリティポリシーによりシステム全体の本来満たすべき目的が達成されているかを人手で確認することは難しい.

本論文では, MAC および DAC の両モデルについて, 設定されたアクセス制御ポリシーによって, システム全体に対するセキュリティ要求が達成されているかを, ソフトウェアの静的解析を用いて判定する方法を提案する.

2 章では, MACに対するセキュリティ保全手法として, 手続き型プログラムに対する情報フロー解析アルゴリズムを提案する. 提案アルゴリズムは, 解析対象プログラムとそのプログラムへの入力データのセキュリティレベルが与えられたとき, 出力データのセキュリティレベルを推論する. 提案手法では, セキュリティレベルの構造は任意の有限束で表現することができる. 解析手法として, 抽象解釈 (abstract interpretation) 法を用いることにより, 任意の再帰プログラムに対する解析を可能としている. 解析法の健全性の証明をおこない, 提案アルゴリズムの時間計算量がプログラムサイズの 3 乗オーダで実行

*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-DT0161014, 2004 年 2 月 6 日.

できることを示した. さらに, 暗号化関数等, 入力情報を隠蔽する機能を組込み演算としてもつプログラムの解析を目的として提案アルゴリズムを拡張する方法についても述べる. 試作システムを用いて, 例プログラムを解析した結果についても議論をおこなう.

3 章では, DAC に基づいたシステムのセキュリティ保全手法を提案する. まず, 簡潔なポリシー記述言語を提案する. 従来の DAC モデルでは, ポリシーとしてアクセス許可およびアクセス禁止の 2 種類を記述することができるが, 提案するポリシー記述言語では, 義務ポリシーも記述することができる. 義務ポリシーとは, あるイベントが発生したときに義務的に実行されなければならない動作を記述するポリシーである. ポリシー制御系 (Policy Controlled System, PCS) とは, 個々のオブジェクトがそれぞれにセキュリティポリシーをもち, オブジェクトのふるまいをポリシーにより制御するようなシステムである. この PCS の操作的意味を 3 章で定義する. 次に, 「PCS P と検証条件 F が与えられたとき, P の到達可能な状態は全て F を満たすか」という問題を安全性検証問題として定義する. この問題の解析手法として, プッシュダウンシステムに対するモデル検査法を用いた. 最後に, 試作システムを用いて例 PCS の検証を行った結果を紹介する.

キーワード

アクセス制御, ポリシー, 情報フロー解析, セキュリティ検証, モデル検査

List of Publications

1 Journal Papers

(1) Shigeta Kuninobu, Yoshiaki Takata, Hiroyuki Seki and Katsuro Inoue: "An Information Flow Analysis of Recursive Programs based on a Lattice Model of Security Classes," IEICE Transactions on Information and Systems (D-I), J85-D-I(10), pp.961–973, Oct. 2002 (in Japanese).
English translation will appear in Systems and Computers in Japan.

(2) Shigeta Kuninobu, Naoya Nitta, Yoshiaki Takata and Hiroyuki Seki: "Policy Controlled System and Its Model Checking," Submitted to IEICE Transactions on Information and Systems.

2 International Conferences (Reviewed)

(3) Shigeta Kuninobu, Yoshiaki Takata, Hiroyuki Seki and Katuro Inoue: "An Efficient Information Flow Analysis of Recursive Programs based on a Lattice Model of Security Classes," Proceedings of Third International Conference on Information and Communications Security (ICICS 2001), Xian, China, Nov. 2001, Lecture Notes in Computer Science 2229, pp.292–303.

(4) Shigeta Kuninobu, Yoshiaki Takata, Daigo Taguchi, Masayuki Nakae and Hiroyuki Seki: "A Specification Language for Distributed Policy Control," Proceedings of Fourth International Conference on Information and Communications Security (ICICS 2002), Singapore, Dec. 2002, Lecture Notes in Computer Science 2513, pp.386–398.

3 Workshop (Reviewed)

(5) Shigeta Kuninobu, Yoshiaki Takata, Hiroyuki Seki and Katsuro Inoue: "An Information Flow Analysis Algorithm based on a Lattice Model of Security Classes," Proceedings of the 3rd Programming and Programming Language Workshop (PPL2001), pp.109–118, Mar. 2001 (in Japanese).

4 Other Workshops

(6) Shigeta Kuninobu, Yoshiaki Takata, Hiroyuki Seki and Katsuro Inoue: "An Information Flow Analysis of Programs based on a Lattice Model," Technical Report of IEICE, SS2000-11, pp.25–32, Nov. 2000 (in Japanese).

(7) Shigeta Kuninobu, Yoshiaki Takata, Daigo Taguchi, Masayuki Nakae and Hiroyuki Seki: "A Specification Language for Distributed Policy Control," Technical Report of IEICE, SS2002-19, pp.25–30, Sep. 2002.

(8) Shigeta Kuninobu, Yoshiaki Takata, Naoya Nitta and Hiroyuki Seki: "A Verification Method for Distributed Policy Control," Technical Report of IEICE, SS2002-44, pp.43–48, Jan. 2003.

(9) Hiroyuki Seki, Naoya Nitta, Yoshiaki Takata and Shigeta Kuninobu: "Infinite State Model Checking and Its Application to Software Verification," Conference Proceedings of the 2nd Workshop of Critical Software (WOCS), pp.20–22, Mar. 2003.

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 Katsuro Inoue for his valuable suggestion in this research. I would like to express my sincere gratitude to Assistant Professor Yoshiaki Takata for his support and advice throughout the research. I would like to thank Assistant Professor Naoya Nitta for his help of the work. He provided me beneficial comments for verification system of PCS. I also thank Mr. Daigo Taguchi of NEC Corporation for valuable discussions on the design of policy specification language. 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 computer networks grow rapidly, it becomes more important to assure the security of a computer system against attacks by malicious users. Access control, as well as cryptography, is one of the fundamental technologies for establishing computer security. Access control mechanisms are roughly classified into mandatory access control (MAC) and discretionary access control (DAC). In both of MAC and DAC, however, it is difficult to manually check whether access control under given security policies satisfies the security goals which the whole system is required to achieve. In this thesis, we propose static analysis methods which check whether the whole system behavior determined by access control based on given security policies satisfies security goals of the system.

Model checking [CGP00] is a well-known technique of automatically verifying whether a system satisfies a given property. Most of the existing model checking methods and tools assume that a system to be verified has finite state space. However, since a system which executes a program has an infinite state space, static analysis, including model checking, which checks all the states in the system is impossible. Therefore, the system should be abstracted according to the purpose of analysis before the analysis. However, if the abstracting method is not well-considered, the abstract system does not always retain the desirable property of the original system, in which case the analysis fails.

In Chapter 2, an information flow analysis for a procedural program is proposed as a security assurance method for MAC based system. Information flow analysis is in a sense a special kind of data flow analysis problems. The proposed algorithm uses abstract interpretation as an analysing method. In Chapter 3, a security assurance method for a DAC based system is proposed. The proposed method verifies whether the access control under given security policies satisfies the security goals which the

Table 1.1. Access Control Model and Analysis Method

	Chapter 2	Chapter 3
Access Control Model	Mandatory Access Control	Discretionary Access Control
Type of the Analysis	Data-Flow Analysis	Control-Flow Analysis
Analysing Method	Abstract Interpretation	Model Checking

whole system is required to achieve by using the model checking technique for pushdown system (PDS). The verification problem in Chapter 3 can be considered as a (semantic) control flow analysis problem. Table 1.1 shows the relation of the analysing technique used in Chapter 2 and Chapter 3. Finally, in Chapter 4, we conclude the paper.

In the rest of this chapter, backgrounds and research motivations are summarized and related works are also surveyed.

Security Assurance Method for MAC based system In MAC, security levels such as *top-secret*, *confidential* and *unclassified* are assigned to data and users (or processes). Security level for data is called security class (SC) and security level for users (or processes) is called clearance. User can access to data d if and only if the clearance of user is higher or equal to the SC of d. However, if a program with clearance *top-secret* reads data d with SC *top-secret*, creates some data d' from d and writes d' as a data with SC *unclassified* then an undesirable leaking may occur since data d' may contain some information on data d. One way to prevent these kinds of information leaks is to conduct a program analysis which statically infers the SC of each output of the program when the SC of each input is given. Several program analyses based on a lattice model of SC have been proposed. [De76] and [DD77] are the pioneering works which proposed a systematic method of analyzing information flow based on a lattice model of SCs. Subsequently, Denning's analysis method has been formalized and extended in a various way by Hoare-style axiomatization [BBM94], by abstract interpretation [O95], and by type theory [VS97, HR98, LR98].

In a type theoretic approach, a type system is defined so that if a given program is well-typed then the program has *noninterference* property such that it does not cause undesirable information flow. [VS97] provides a type system for statically analyzing information flow of a simple procedural program and proves its correctness. The method in [VS97] assumes a program without a recursive procedure while method proposed in this paper can analyze a program which may contain recursive procedures. [HR98] defines a type system for a functional language called Slam calculus to analyze

noninterference property. However, [HR98] does not prove the correctness of the analysis algorithm for an extended model in which assignment statements were introduced. [SV98] showed that their type system in [VS97] is no longer correct in a distributed environment and presented a new type system for a multi-threaded language. How to extend the proposed method to fit a distributed environment is a future study.

A structure of SCs modeled as a finite lattice is usually a simple one such as {*top-secret, confidential, unclassified*}. [ML98] proposes a finer grained model of SCs called decentralized labels. Based on this model, [My99] 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 is provided. However, their type system has not been formally verified to be sound.

Noninterference property in a distributed system has also been extensively studied by using process algebra (for example, [RWW94, AFG99, RS99]). [AFG99] and its companion papers use a process algebra called spi calculus to analyze cryptographic protocols.

In Chapter 2, we propose an algorithm which analyzes information flow of a program containing recursive procedures. The algorithm constructs equations from statements in the program. The equation constructed from a statement represents the information flow caused by the execution of the statement. The algorithm computes the least fixpoint of these equations. We describe the algorithm as an abstract interpretation and prove the soundness of the algorithm. For a given program *Prog*, the algorithm can be executed in $O(N^3)$ time where N is the total size of *Prog*. Based on the proposed method, a prototypic system has been implemented. Experimental results by using the system are also presented.

In the algorithm proposed in this paper and most of all other existing methods, the SC of the result of a built-in operation θ (e.g., addition) is assumed to be the least upper bound of the SCs of all input arguments of θ. This means that information on each argument may flow into the result of the operation. However, this assumption is not appropriate for some operations such as an encryption operation. For these operations, it is practically difficult to recover information on input arguments from the result of the operation. Considering the above discussions, the proposed method is extended so that these operations can be appropriately modeled by using a congruence relation.

Security Assurance Method for DAC based system In DAC, the owner of each data freely assigns a security policy (permission) to the data and access control for a data is performed based on the security policy of the data. In a conventional DAC

model, only permission (positive authorization) and prohibition (negative authorization) can be specified. Unix file permission is a typical example of conventional DAC model. However, an obligation policy is also important, which describes that a specified subject is obliged to perform a specified action when a certain event occurs under a certain condition. A few specification languages are proposed which have rich functions to describe policies in a concise way (for example, [DDLS01, JSS97, KS02]). In [JSS97] and their companion papers, logical languages based on the Horn-clause are used as specification languages, and various theoretical problems such as authorization inheritance and authorization conflict detection/resolution are discussed. In [KS02], the model of [JSS97] is extended so that one can express an obligation policy. However, in these papers, how to define a formal semantics has not been discussed.

Ponder [DDLS01, Da02] is a general purpose policy specification language in which one can specify obligation, conditional and data-dependent policies. Also the formal semantics of a subset of the language is defined in [Da02, T01]. However, they define only the execution order of methods under given policies, leaving the rest of the system as a blackbox. In this thesis, the entire behavior of a system under given policies is formally defined by explicitly describing the change of runtime stack configuration. Defining a formal semantics of the system including policy specification language is important since a correct policy control is possible only by correctly interpreting the meaning of a given policy specification.

In Chapter 3, first, we propose a simple but useful policy description language which can specify not only permission and prohibition policies but also obligation policies. Secondly, using the proposed language, we define a policy controlled system (PCS). PCS is a system in which each object has its own security policy (specified by the proposed language) and objects' behaviors are autonomously controlled based on those policies when they interact with one anther. Operational semantics of PCS is formally defined.

We define the (safety) verification problem for PCS as the problem to decide for a given PCS S and a goal (called *safety property*) Ψ, whether every reachable state of P satisfies Ψ. As defined later, in this thesis, Ψ is represented as a regular language. Using a model checking technique for pushdown system (PDS), we propose a method for solving the (safety) verification problem for PCS.

A pushdown system (PDS) is an infinite state transition system with a pushdown stack as well as a finite control. A PDS is a formal model of a system with well-nested structure such as a program which involves recursive procedure calls. Recently, efficient model checking algorithms for PDS and an equivalent model (namely, recursive state

machines (RSM)) have been proposed in [AEY01, BGR01, EHRS00, EKS01].

Efficient algorithms and complexity of LTL and CTL* model checking for PDS are extensively studied in [EHRS00, EKS01]. Verification results using an automatic verification tool are reported in [ES01]. Model checking algorithms for RSM, which is equivalent to PDS, are studied in [AEY01, BGR01]. Their algorithms work in the same complexity as [EHRS00]'s algorithm for a general RSM, and in linear time for a single-exit RSM which models a usual recursive program. The first work which applies model checking of a pushdown-type system to security verification is Jensen et al.'s study [JMT99]. In that paper, the authors formally define a verification problem for a program with an access control which generalizes JDK (Java development kit) stack inspection. However, their approach has severe restrictions, e.g., a mutual recursion is prohibited. Nitta et al. [NTS01a, NTS01b] improved the result of [JMT99] by using indexed grammar in formal language theory, showing that the verification problem is decidable for programs with arbitrary recursion and stack inspection. Esparza et al. independently showed that the problem for a program with stack inspection can be reduced to an LTL model checking for a PDS with regular valuation [EKS01]. Especially, the problem for a program which contains *no* stack inspection is equivalent to the model checking of a safety property ($AG\Psi$) for a PDS with regular valuation. However, the verification problem is computationally intractable (deterministic exponential time complete) [EKS01, NTS01b]. In [NTS01b], a subclass of programs which exactly represents programs with JDK stack inspection is proposed and it is shown that verification of a safety property can be performed in polynomial time of the program size in the subclass. Jha and Reps show that name reduction in SPKI can be represented as a PDS, and prove the decidability of a number of security problems by reductions from decidability properties of the PDS model checking [JR02].

Our verification tool works as follows. First, a PDS is abstracted from a given PCS and a nondeterministic finite automaton (NFA) which accepts the set of all reachable states of the PDS is constructed. As described in section 3.3.3, the NFA construction algorithm in this thesis works in linear time which matches the algorithms in [AEY01] and [BGR01] for a single-exit RSM (see related works). Next, we decide whether every state accepted by the NFA satisfies a given safety property.

The main contribution of Chapter 3 of the thesis is that the proposed method is one of the first attempts to verify a safety property of a policy controlled system. Also, this thesis presents an application of model checking for PDS to a real world verification problem and shows verification results conducted on an automatic tool.

Chapter 2

Information Flow Analysis of Recursive Programs

2.1. Introduction

This chapter proposes an information flow analysis for a procedural program as a security assurance method for Mandatory Access Control (MAC) [P94] based system.

The aim of MAC is to ensure that information flows in one direction. MAC requires that data and users (or processes) be assigned certain security levels represented by a label. A label for a data d is called the security class (SC) of d, denoted as $SC(d)$. A label for a user u is called the clearance of u, denoted as $clear(u)$. In MAC, user u can read data d if and only if $clear(u) \geq SC(d)$. However, if a program with clearance higher than $SC(d)$ reads data d, creates some data d' from d and writes d' to a storage which a user with clearance lower than $SC(d)$ can read then an undesirable leaking may occur since data d' may contain some information on data d. One way to prevent these kinds of information leaks is to conduct a program analysis which statically infers the SC of each output of the program when the SC of each input is given. In this chapter, we propose an algorithm which analyzes information flow of a program containing recursive procedures. In the algorithm, SC of data can be formalized as an arbitrary finite lattice. The algorithm constructs equations from statements in the program. The equation constructed from a statement represents the information flow caused by the execution of the statement. The algorithm computes the least fix-point of these equations. We describe the algorithm as an abstract interpretation and prove the soundness of the algorithm. For a given program *Prog*, the algorithm can be executed in $O(N^3)$ time where N is the total size of *Prog*. Based on the proposed method, a

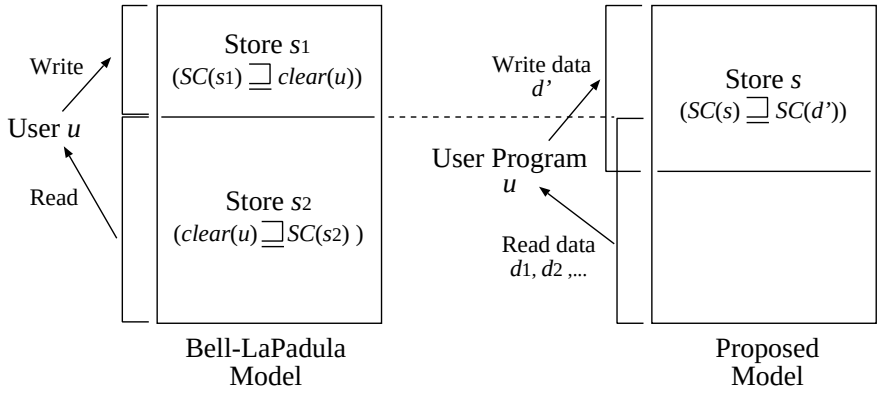

Figure 2.1. Access Control Models

prototypic system has been implemented. Experimental results by using the system are also presented.

Using the information flow analysis algorithm, MAC with Bell-LaPadula model can be made more flexible. MAC with Bell-LaPadula model [PHS03] controls the access based on the following two rules.

- User u may read data d if and only if $SC(d) \sqsubseteq clear(u)$.
- User u may write data to store s if and only if $clear(u) \sqsubseteq SC(s)$.

Note that store s with $SC(s)$ contain data d with $SC(s)$.

In Bell-LaPadula model, SC of data never decrease by these access control rules. Therefore, undesirable information leak may not occur. However, the SC of a output data by user u will become unnecessarily high if u has a high clearance.

To solve this problem, we can use the information flow analysis algorithm as follows. When there is a program which reads data $d_1, d_2, \ldots, d_m$ within the database as one or more arguments and creates a new data d', the analysis will determine the SC of d' using the SCs of $d_1, d_2, \ldots, d_m$. We can write d' to specific store of the database by using the analysis result. That is, d' can be written back to store s if and only if $SC(d') \sqsubseteq SC(s)$ (See Figure 2.1).

The assumption that security classes have a lattice structure is made because it is natural to define the security class of data d_3 which contains information of the data d_1 and d_2 as the least upper bound of $SC(d_1)$ and $SC(d_2)$. On the other hand, in an actual system, the security classes may not have a lattice structure. For example, assume that there exist two users U_1 and U_2 and two user groups G_1 and G_2. Also assume that both

users U_1 and U_2 belong to both groups G_1 and G_2. We can naturally assume $SC(G_i) \leq SC(U_j)(i = 1, 2, j = 1, 2)$. If we hypothetically consider the least upper bound of $SC(G_1)$ and $SC(G_2)$ (which is denoted by $SC(G_3)$), then from $SC(G_i) \leq SC(U_j)(i = 1, 2, j = 1, 2)$ and the definition of the least upper bound, $SC(G_3) \leq SC(U_j)(j = 1, 2)$ holds. Intuitively, $SC(G_3)$ is the security class which includes information of data d_1 with $SC(G_1)$ and data d_2 with $SC(G_2)$. $SC(G_3) \leq SC(U_j)(j = 1, 2)$ represents a reasonable property such that the data which includes information of data d_1 and data d_2 can be read by both users U_1 and U_2. Thus, making the security class have a lattice structure is a natural assumption.

The rest of Chapter 2 of the thesis is organized as follows. Section 2.2 defines the syntax and the operational semantics of a program language which will be the target language of the analysis. In section 2.3, we formally describe the program analysis algorithm, prove the correctness of the algorithm and show the time complexity of the algorithm. A brief example is also presented in section 2.3. The method is extended in section 2.4. Experimental results are briefly presented in section 2.5.

2.2. Definitions

In this section, we define the syntax and semantics of a programming language which will be the input language to the proposed algorithm. This language is a simple procedural language similar to C.

2.2.1 Syntax of Program

A program is a finite set of function definitions. A function definition has the following form:

$$f(x_1, \ldots, x_n) \text{ local } y_1, \ldots, y_m \{P_f\}$$

where f is a function name, $x_1, \ldots, x_n$ are formal arguments of f, $y_1, \ldots, y_m$ are local variables and P_f is a function body. The syntax of P_f is given below where c is a constant, x is a local variable or a formal argument, f is a function name defined in the program and θ is a built-in operator such as addition and multiplication. Any object generated by $cseq$ can be P_f.

$$\begin{aligned} cseq &::= cmd \mid cmd_1; cseq \\ cmd &::= \text{if } exp \text{ then } cseq \text{ else } cseq \text{ fi} \mid \text{return} exp \\ cseq_1 &::= cmd_1 \mid cmd_1; cseq_1 \end{aligned}$$

$$
\begin{aligned}
cmd_1 &::= x := exp \mid \text{if } exp \text{ then } cseq_1 \text{ else } cseq_1 \text{ fi} \mid \text{while } exp \text{ do } cseq_1 \text{ od} \\
exp &::= c \mid x \mid f(exp, \ldots, exp) \mid \theta(exp, \ldots, exp)
\end{aligned}
$$

Objects derived from exp, cmd or cmd_1, $cseq$ or $cseq_1$ are called an expression, a command, a sequence of commands, respectively. An execution of a program $Prog$ is the evaluation of the function named $main$, which should be defined in $Prog$. Inputs for $Prog$ are actual arguments of $main$ and the output of $Prog$ for these inputs is the return value of $main$.

2.2.2 Semantics of Program

We assume the following types to define the operational semantics of a program. Let $\times$ denote the cartesian product and $+$ denote the disjoint union.

type val **(values)** We assume for each n-ary built-in operator θ, n-ary operation $\theta_{\mathcal{I}} : val \times \cdots \times val \to val$ is defined. Every value manipulated or created in a program has the same type val.

type $store$ There exist two functions

$$
\begin{aligned}
&lookup : store \times var \to val \\
&update : store \times var \times val \to store
\end{aligned}
$$

which satisfies:

$$
lookup(update(\sigma, x, v), y) \quad = \quad \text{if } x = y \text{ then } v \text{ else } lookup(\sigma, y).
$$

For readability, we use the following abbreviations:

$$
\sigma(x) \equiv lookup(\sigma, x), \quad \sigma[x := v] \equiv update(\sigma, x, v).
$$

Let $\bot_{store}$ denote the store such that $\bot_{store}(x)$ is undefined for every x.

We define a mapping which provides the semantics of a program. This mapping takes a store and one of an expression, a command and a sequence of commands as arguments and returns a store or a value.

$\models$: $(store \to exp \to val) + (store \to cmd \to (store + val)) + (store \to cseq \to (store + val))$

- "$\sigma \models M \Rightarrow v$" means that a store σ evaluates an expression M to the value v, that is, if M is evaluated by using σ then v is obtained.

- "$\sigma \models C \Rightarrow \sigma'$" means that a store σ becomes σ' if a command C is executed.
- "$\sigma \models C \Rightarrow v$" means that if a command C is executed when the store is σ then the value v is returned. This mapping is defined only when C has the form of 'return M' for some expression M.
- Similar for a sequence of commands.

Figure 2.2 shows axioms and inference rules which define the semantic mapping, where the following meta-variables are used.

$x, x_1, \ldots, y_1, \ldots : var$	$M, M_1, \ldots : exp$	$C : cmd$ or cmd_1
$P, P_1, P_2 : cseq$ or $cseq_1$	$\sigma, \sigma', \sigma'' : store$	

2.3. The Analysis Algorithm

A security class (abbreviated as SC) represents the security level of a value in a program. Let $SCset$ be a finite set of security classes. Also assume that a partial order $\sqsubseteq$ is defined on $SCset$ and ($SCset$, $\sqsubseteq$) forms a lattice; let $\bot$ denote the minimum element of $SCset$ and let $a_1 \sqcup a_2$ denote the least upper bound of a_1 and a_2 for $a_1, a_2 \in SCset$. Intuitively, $\tau_1 \sqsubseteq \tau_2$ means that τ_2 is more secure than τ_1; it is legal that a user with clearance τ_2 can access a value with SC τ_1. A simple example of $SCset$ is:

$$SCset = \{low, high\}, \quad low \sqsubseteq high.$$

The purpose of the analysis is to infer (an upper bound of) the SC of the output value when an SC of each input is given. Precisely, the analysis problem for a given program *Prog* is to infer an SC of the output value of *Prog* which satisfies the soundness property defined in section 2.3.3.

One of the security preservation method which uses the approach of setting the security classes to "*high*" and "*low*" is MAC which is the main access control method for database as mentioned before.

We describe the analysis algorithm in section 2.3.1 and the soundness of the proposed algorithm is proved in section 2.3.4.

2.3.1 The Algorithm

To describe the algorithm, we use the following types.

type *sc* **(security class)** .

(CONST) $\sigma \models c \Rightarrow c_{\mathcal{I}}$

(VAR) $\sigma \models x \Rightarrow \sigma(x)$

(PRIM)
$$\frac{\sigma \models M_i \Rightarrow v_i \quad (1 \leq i \leq n)}{\sigma \models \theta(M_1, \ldots, M_n) \Rightarrow \theta_{\mathcal{I}}(v_1, \ldots, v_n)}$$

(CALL)
$$\frac{\sigma \models M_i \Rightarrow v_i \quad (1 \leq i \leq n) \qquad \sigma' \models P_f \Rightarrow v}{\sigma \models f(M_1, \ldots, M_n) \Rightarrow v}$$
$$\left(\begin{array}{l} f(x_1, \ldots, x_n) \text{ local } y_1, \ldots, y_m \quad \{P_f\} \\ \sigma' = \bot_{store}[x_1 := v_1] \cdots [x_n := v_n] \end{array} \right)$$

(ASSIGN)
$$\frac{\sigma \models M \Rightarrow v}{\sigma \models x := M \Rightarrow \sigma[x := v]}$$

(IF1)
$$\frac{\sigma \models M \Rightarrow \text{true} \qquad \sigma \models P_1 \Rightarrow \sigma' \ (rsp.\ v)}{\sigma \models \text{if } M \text{ then } P_1 \text{ else } P_2 \text{ fi} \Rightarrow \sigma' \ (rsp.\ v)}$$

(IF2)
$$\frac{\sigma \models M \Rightarrow \text{false} \qquad \sigma \models P_2 \Rightarrow \sigma' \ (rsp.\ v)}{\sigma \models \text{if } M \text{ then } P_1 \text{ else } P_2 \text{ fi} \Rightarrow \sigma' \ (rsp.\ v)}$$

(WHILE1)
$$\frac{\sigma \models M \Rightarrow \text{true} \qquad \sigma \models P \Rightarrow \sigma' \qquad \sigma' \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma''}{\sigma \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma''}$$

(WHILE2)
$$\frac{\sigma \models M \Rightarrow \text{false}}{\sigma \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma}$$

(RETURN)
$$\frac{\sigma \models M \Rightarrow v}{\sigma \models \text{return } M \Rightarrow v}$$

(CONCAT)
$$\frac{\sigma \models C \Rightarrow \sigma' \qquad \sigma' \models P \Rightarrow \sigma'' \ (rsp.\ v)}{\sigma \models C; P \Rightarrow \sigma'' \ (rsp.\ v)}$$

Figure 2.2. Semantic mapping

type $\underline{store}$ **(SC of store)**

$$update : \underline{store} \times var \times sc \to \underline{store}$$
$$lookup : \underline{store} \times var \to sc$$

For $\underline{store}$ type, we use the same abbreviations as for $store$ type. If $\underline{\sigma}$ is an element of type $\underline{store}$, then $\underline{\sigma}(x)$ is the SC of variable x inferred by the algorithm. By extending the partial order $\sqsubseteq$ defined on sc to type $\underline{store}$ as shown below, we can provide a lattice structure to $\underline{store}$:

For $\underline{\sigma}$ and $\underline{\sigma}'$ of type $\underline{store}$, $\underline{\sigma} \sqsubseteq \underline{\sigma}' \Leftrightarrow \forall x \in var.\ \underline{\sigma}(x) \sqsubseteq \underline{\sigma}'(x)$.

The minimum element of $\underline{store}$ is $\underline{\sigma}$ satisfying $\forall x \in var.\ \underline{\sigma}(x) = \bot$. We write this minimum element as $\bot_{\underline{store}}$.

type $\underline{fun}$ **(SC of function)** Similarly to type $\underline{store}$, the following functions are defined.

$$lookup : \underline{fun} \times fname \to (sc \times \cdots \times sc \to sc)$$
$$update : \underline{fun} \times fname \times (sc \times \cdots \times sc \to sc) \to \underline{fun}$$

We use the following abbreviations for $F \in \underline{fun}$, $f \in fname$ and $\psi : sc \times \cdots \times sc \to sc$.

$$F[f] \equiv lookup(F, f)$$
$$F[f := \psi] \equiv update(F, f, \psi)$$

For n-ary function f and SCs $\tau_1, \ldots, \tau_n$, $F[f](\tau_1, \ldots, \tau_n)$ is the SC of the returned value of f inferred by the algorithm when the SC of i-th argument is specified as $\tau_i\,(1 \leq i \leq n)$. Similarly to type $\underline{store}$, we can provide a lattice structure to type $\underline{fun}$. The minimum element of $\underline{fun}$ is denoted as $\bot_{\underline{fun}}$.

type cv-$\underline{fun}$ **(covariant** $\underline{fun}$**)** This type consists of every F of type $\underline{fun}$ which satisfies the next condition:

If $\tau_i \sqsubseteq \tau_i'$ for $1 \leq i \leq n$ then $F[f](\tau_1, \ldots, \tau_n) \sqsubseteq F[f](\tau_1', \ldots, \tau_n')$.

We use the following meta-variables.

$$\underline{\sigma}, \underline{\sigma}', \underline{\sigma}'' : \underline{store} \qquad F, F_1, F_2 : \underline{fun}$$

Below we define a function $\mathcal{A}[\![\cdot]\!]$ which analyzes the information flow. Before defining the analysis function, we explain implicit flow [De76]. Consider the following command.

$$\text{if } x = 0 \text{ then } y := 0 \text{ else } y := 1 \text{ fi}$$

In this command, the variable x occurs neither in $y := 0$ nor in $y := 1$. However, after executing this command, we can know whether x is 0 or not by checking whether y is 0 or 1. Therefore, we can consider information on the value stored in the variable x flows into the variable y. In general, information may flow from the conditional clause of a "if" command into "then" and "else" clauses and also it may flow from the conditional clause of a "while" command into "do" clause. Such information flow is called *implicit flow*. The function $\mathcal{A}[\![\cdot]\!]$ infers that the SC of implicit flow caused by a command C or a sequence P of commands is the least upper bound of the SCs of the conditional clauses of all the "if" and "while" commands which contain C or P in their scopes. $\mathcal{A}[\![\cdot]\!]$ takes the SC of implicit flow as its fourth argument.

$$\begin{aligned}\mathcal{A} : &(exp \times \underline{fun} \times \underline{store} \to sc) + (cmd \times \underline{fun} \times \underline{store} \times sc \to \underline{store}) \\ &+ (cseq \times \underline{fun} \times \underline{store} \times sc \to \underline{store})\end{aligned}$$

- "$\mathcal{A}[\![M]\!](F, \underline{\sigma}) = \tau$" means that, for SCs F of functions and an SC $\underline{\sigma}$ of a store, the SC of an expression M is analyzed as τ.
- "$\mathcal{A}[\![C]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}'$" means that, for SCs F of functions, an SC $\underline{\sigma}$ of a store and an SC ν of implicit flow, the SC of the store after executing a command C is analyzed as $\underline{\sigma}'$.
- Similar for a sequence of commands.

The definition of $\mathcal{A}$ is shown in Figure 2.3.

Define the function $\mathcal{A}[\![\cdot]\!] : program \to \underline{fun} \to \underline{fun}$, which performs 'one-step' analysis of information flow for each function f defined in a given program as follows:

$$\begin{aligned}&\text{For } Prog \equiv \{f(x_1, \ldots, x_n) \text{ local } y_1, \ldots, y_m \ \ \{P_f\}, \ldots\}, \\ &\quad \mathcal{A}[\![Prog]\!](F) = \\ &\qquad F[f := \lambda\tau_1 \ldots \tau_n.(\mathcal{A}[\![P_f]\!](F, \bot_{\underline{store}}[x_1 := \tau_1] \cdots [x_n := \tau_n], \bot)(ret)) \\ &\qquad\quad \mid f \text{ is an } n\text{-ary function defined in } Prog] \end{aligned} \tag{2.1}$$

For a lattice $(S, \preceq)$ and a function $f : S \to S$, we write the least fix-point of f as $fix(f)$. For a program $Prog$, the function $\mathcal{A}^*[\![Prog]\!]$ which analyzes information flow of $Prog$ is defined as the least fix-point of $\mathcal{A}[\![Prog]\!]$, that is,

$$\mathcal{A}^*[\![Prog]\!] = fix(\lambda F.\mathcal{A}[\![Prog]\!](F)). \tag{2.2}$$

(CONST)	$\mathcal{A}[\![c]\!](F, \underline{\sigma}) = \bot$
(VAR)	$\mathcal{A}[\![x]\!](F, \underline{\sigma}) = \underline{\sigma}(x)$
(PRIM)	$\mathcal{A}[\![\theta(M_1, \ldots, M_n)]\!](F, \underline{\sigma}) = \bigsqcup_{1 \leq i \leq n} \mathcal{A}[\![M_i]\!](F, \underline{\sigma})$
(CALL)	$\mathcal{A}[\![f(M_1, \ldots, M_n)]\!](F, \underline{\sigma}) = F[f](\mathcal{A}[\![M_1]\!](F, \underline{\sigma}), \ldots, \mathcal{A}[\![M_n]\!](F, \underline{\sigma}))$
(ASSIGN)	$\mathcal{A}[\![x := M]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}[x := \mathcal{A}[\![M]\!](F, \underline{\sigma}) \sqcup \nu]$
(IF)	$\mathcal{A}[\![\text{if } M \text{ then } P_1 \text{ else } P_2 \text{ fi}]\!](F, \underline{\sigma}, \nu)$
	$= \mathcal{A}[\![P_1]\!](F, \underline{\sigma}, \nu \sqcup \tau) \sqcup \mathcal{A}[\![P_2]\!](F, \underline{\sigma}, \nu \sqcup \tau)$
	where $\tau = \mathcal{A}[\![M]\!](F, \underline{\sigma})$
(WHILE)	$\mathcal{A}[\![\text{while } M \text{ do } P \text{ od}]\!](F, \underline{\sigma}, \nu) = \mathcal{A}[\![P]\!](F, \underline{\sigma}, \nu \sqcup \mathcal{A}[\![M]\!](F, \underline{\sigma})) \sqcup \underline{\sigma}$
(RETURN)	Let ret be a fresh variable which contains a return value of a function.
	$\mathcal{A}[\![\text{return } M]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}[ret := \mathcal{A}[\![M]\!](F, \underline{\sigma}) \sqcup \nu]$
(CONCAT)	$\mathcal{A}[\![C; P]\!](F, \underline{\sigma}, \nu) = \mathcal{A}[\![P]\!](F, \mathcal{A}[\![C]\!](F, \underline{\sigma}, \nu), \nu)$

Figure 2.3. Definition of $\mathcal{A}$

As will be shown in lemma 2.3.3, $\mathcal{A}[\![Prog]\!]$ is a monotonic function on the finite lattice cv-$\underline{fun}$. Therefore,

$$\mathcal{A}^*[\![Prog]\!] = \bigsqcup_{i \geq 0} \mathcal{A}[\![Prog]\!]^i(\bot_{\underline{fun}}) \tag{2.3}$$

holds [Mi96] where $f^0(x) = x$, $f^{i+1}(x) = f(f^i(x))$. Hence, $\mathcal{A}^*[\![Prog]\!]$ can be calculated by starting with $\bot_{\underline{fun}}$ and repeatedly applying $\mathcal{A}[\![Prog]\!]$ to the SCs of functions until the SCs of the functions remains unchanged.

2.3.2 Analysis Example

In this subsection, we show how our analysis algorithm works. The program which we are going to analyze is written below. In this example, we assume $SCset = \{low, high\}$, $low \sqsubseteq high$.

```
main(x, y) {                    f(x) {
  while x > 0 do                  if x > 0 then
    y := x + 1;                     return x * f(x − 1)
    x := y − 4                    else
  od;                               return 0
  return f(x) + y                 fi
}                               }
```

In order to analyze this program, we continue updating F using the following relation until F does not change any more.

$$F = F[main := \lambda\tau_1\tau_2.(\mathcal{A}[\![P_{main}]\!](F, \bot_{\underline{store}}[x{:=}\tau_1][y{:=}\tau_2], \bot)(ret))]$$
$$[f := \lambda\tau_1.(\mathcal{A}[\![P_f]\!](F, \bot_{\underline{store}}[x := \tau_1], \bot)(ret))]$$

The table below shows how F changes. The SCs of the i-th column are calculated by using the SCs of the $(i-1)$th column.

	0	1	2	3
$F[main]$	$\lambda\tau_1\tau_2.\bot$	$\lambda\tau_1\tau_2.\tau_2$	$\lambda\tau_1\tau_2.\tau_1 \sqcup \tau_2$	$\lambda\tau_1\tau_2.\tau_1 \sqcup \tau_2$
$F[f]$	$\lambda\tau_1.\bot$	$\lambda\tau_1.\tau_1$	$\lambda\tau_1.\tau_1$	$\lambda\tau_1.\tau_1$

From this table, we can know that $\mathcal{A}^*[\![Prog]\!][main](\tau_1, \tau_2) = \tau_1 \sqcup \tau_2$, that is, the SC of the return value of the main function is *low* when the SC of both actual arguments are *low*. Otherwise, SC of the return value of the main function could be *high*.

2.3.3 Definition of Soundness

Generally, the analysis algorithm is a function $\mathcal{Z}$ of the following type:

$$\mathcal{Z}^*[\![\cdot]\!] : program \to fname \to (sc \times \cdots \times sc \to sc).$$

$\mathcal{Z}^*[\![Prog]\!][f](\tau_1, \ldots, \tau_n) = \tau$ means that for an n-ary function f defined in $Prog$ and for SCs $\tau_1, \ldots, \tau_n$ of arguments of f, $\mathcal{Z}^*[\![\cdot]\!]$ infers that the SC of f is τ.

Definition 2.3.1. An analysis algorithm $\mathcal{Z}^*[\![\cdot]\!]$ is **sound** if the following condition (called *noninterference* property) is satisfied.

Assume $Prog$ is a program and $main$ is the main function of $Prog$. If

$$\mathcal{Z}^*[\![Prog]\!][main](\tau_1, \ldots, \tau_n) = \tau,$$
$$\bot_{store} \models main(v_1, \ldots, v_n) \Rightarrow v, \quad \bot_{store} \models main(v'_1, \ldots, v'_n) \Rightarrow v',$$
$$\forall i\,(1 \le i \le n) : \tau_i \sqsubseteq \tau.\ v_i = v'_i$$

then $v = v'$ holds. □

By the above definition, an analysis algorithm is sound if and only if the following condition is satisfied: assume that the analysis algorithm answers "the SC of the returned value of the main function is τ if the SC of the i-th argument is τ_i." If every

actual argument with SC equal to or less than τ remain the same then returned values of the main function also remains the same even if an actual argument with SC higher than or incomparable with τ changes. Intuitively, this means that if the analysis algorithm answers "the SC of the main function is τ," then information contained in each actual argument with SC higher than or incomparable with τ does not flow into the return value of the main function.

2.3.4 Soundness of the Algorithm

This section shows the soundness of the proposed algorithm $\mathcal{A}$ (and $\mathcal{A}^*$) defined in subsection 2.3.1. The following two lemmas guarantee the validity of the equation (2.3).

Lemma 2.3.2. (*monotonicity*) This lemma is used to prove lemma 2.3.3.
Assume F_1, F_2 are of type cv-*fun*, $F_1 \sqsubseteq F_2$, $\underline{\sigma}_1 \sqsubseteq \underline{\sigma}_2$ and $\nu_1 \sqsubseteq \nu_2$.

(a) $\mathcal{A}[\![M]\!](F_1, \underline{\sigma}_1) \sqsubseteq \mathcal{A}[\![M]\!](F_2, \underline{\sigma}_2)$.

(b) $\mathcal{A}[\![P]\!](F_1, \underline{\sigma}_1, \nu_1) \sqsubseteq \mathcal{A}[\![P]\!](F_2, \underline{\sigma}_2, \nu_2)$.

Proof. We will prove the lemma by induction on the structure of M and P.
(Proof of *(a)*) (CALL) Let $M = f(M_1, \ldots, M_n)$. By the inductive hypothesis, $\mathcal{A}[\![M_i]\!](F_1, \underline{\sigma}_1) \sqsubseteq \mathcal{A}[\![M_i]\!](F_2, \underline{\sigma}_2)(1 \leq i \leq n)$.

$$
\begin{array}{lll}
\mathcal{A}[\![f(M_1, \ldots, M_n)]\!](F_1, \underline{\sigma}_1) & & \\
\quad = & F_1[f](\mathcal{A}[\![M_1]\!](F_1, \underline{\sigma}_1), \ldots, \mathcal{A}[\![M_n]\!](F_1, \underline{\sigma}_1)) & ((\underline{\text{CALL}})) \\
\quad \sqsubseteq & F_1[f](\mathcal{A}[\![M_1]\!](F_2, \underline{\sigma}_2), \ldots, \mathcal{A}[\![M_n]\!](F_2, \underline{\sigma}_2)) & (F_1 \text{ is of type cv-}\underline{fun}) \\
\quad \sqsubseteq & F_2[f](\mathcal{A}[\![M_1]\!](F_2, \underline{\sigma}_2), \ldots, \mathcal{A}[\![M_n]\!](F_2, \underline{\sigma}_2)) & (F_1 \sqsubseteq F_2) \\
\quad = & \mathcal{A}[\![f(M_1, \ldots, M_n)]\!](F_2, \underline{\sigma}_2) & ((\underline{\text{CALL}}))
\end{array}
$$

The lemma can be easily proved for the other cases.

(Proof of *(b)*) (CONCAT) Let $P = C; P_1$. By the inductive hypothesis on C, $\mathcal{A}[\![C]\!](F_1, \underline{\sigma}_1, \nu_1) \sqsubseteq \mathcal{A}[\![C]\!](F_2, \underline{\sigma}_2, \nu_2)$. Also by the inductive hypothesis on P_1,

$$
\begin{array}{lll}
\mathcal{A}[\![C; P_1]\!](F_1, \underline{\sigma}_1, \nu_1) & & \\
\quad = & \mathcal{A}[\![P_1]\!](F_1, \mathcal{A}[\![C]\!](F_1, \underline{\sigma}_1, \nu_1), \nu_1) & ((\underline{\text{CONCAT}})) \\
\quad \sqsubseteq & \mathcal{A}[\![P_1]\!](F_2, \mathcal{A}[\![C]\!](F_2, \underline{\sigma}_2, \nu_2), \nu_2) & (\textit{ind. hypo.}) \\
\quad = & \mathcal{A}[\![C; P_1]\!](F_2, \underline{\sigma}_2, \nu_2). & ((\underline{\text{CONCAT}}))
\end{array}
$$

The lemma can be easily proved for the other cases. □

Lemma 2.3.3. *(a)* If F is of type cv-$\underline{fun}$ then $\mathcal{A}[\![Prog]\!](F)$ is also of type cv-$\underline{fun}$.

(b) (*monotonicity*) Assume F_1 and F_2 are of type cv-$\underline{fun}$. If $F_1 \sqsubseteq F_2$ then $\mathcal{A}[\![Prog]\!](F_1) \sqsubseteq \mathcal{A}[\![Prog]\!](F_2)$.

Proof. We will prove the lemma by induction on the structure of *Prog*.

(Proof of *(a)*) Assume $\tau_i \sqsubseteq \tau_i'$ for $1 \leq i \leq n$.

$$\begin{array}{lll}
& \mathcal{A}[\![Prog]\!](F)[f](\tau_1, \ldots, \tau_n) & \\
= & \mathcal{A}[\![P_f]\!](F, \bot_{\underline{store}}[x_1 := \tau_1] \cdots [x_n := \tau_n], \bot) & ((2.1)) \\
\sqsubseteq & \mathcal{A}[\![P_f]\!](F, \bot_{\underline{store}}[x_1 := \tau_1'] \cdots [x_n := \tau_n'], \bot) & (\text{lemma } 2.3.2(b)) \\
= & \mathcal{A}[\![Prog]\!](F)[f](\tau_1', \ldots, \tau_n') & ((2.1))
\end{array}$$

(Proof of *(b)*)

$$\begin{array}{lll}
& \mathcal{A}[\![Prog]\!](F_1)[f](\tau_1, \ldots, \tau_n) & \\
= & \mathcal{A}[\![P_f]\!](F_1, \bot_{\underline{store}}[x_1 := \tau_1] \cdots [x_n := \tau_n], \bot) & ((2.1)) \\
\sqsubseteq & \mathcal{A}[\![P_f]\!](F_2, \bot_{\underline{store}}[x_1 := \tau_1] \cdots [x_n := \tau_n], \bot) & (\text{lemma } 2.3.2(b)) \\
= & \mathcal{A}[\![Prog]\!](F_2)[f](\tau_1, \ldots, \tau_n) & ((2.1))
\end{array}$$

□

The following two lemmas are used to prove lemma 2.3.6

Lemma 2.3.4. (*property of implicit flow*) If $\mathcal{A}[\![P]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}'$, $\sigma \models P \Rightarrow \sigma'$ and $\nu \not\sqsubseteq \underline{\sigma}'(y)$ then $\sigma(y) = \sigma'(y)$.

Proof. We will prove the lemma by induction on the application number of inference rules to deduce $\sigma \models P \Rightarrow \sigma'$.

(ASSIGN) Assume

$$\underline{\sigma}' = \mathcal{A}[\![x := M]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}[x := \mathcal{A}[\![M]\!](F, \underline{\sigma}) \sqcup \nu] \tag{2.4}$$

$$\frac{\sigma \models M \Rightarrow v}{\sigma \models x := M \Rightarrow \sigma[x := v]} \tag{2.5}$$

$$\nu \not\sqsubseteq \underline{\sigma}'(y). \tag{2.6}$$

If $x = y$, then $\underline{\sigma}'(y) = \mathcal{A}[\![M]\!](F, \underline{\sigma}) \sqcup \nu \sqsupseteq \nu$ by (2.4). This conflicts with (2.6). Therefore $x \neq y$. By (2.5), $\sigma(y) = \sigma[x := v](y)$.

(WHILE) Assume

$$\underline{\sigma}'' = \underline{\sigma}' \sqcup \underline{\sigma} \tag{2.7}$$

$$\underline{\sigma}' = \mathcal{A}[\![P]\!](F, \underline{\sigma}, \nu \sqcup \mathcal{A}[\![M]\!](F, \underline{\sigma})) \tag{2.8}$$

$$\frac{\sigma \models M \Rightarrow \text{true} \qquad \sigma \models P \Rightarrow \sigma' \qquad \sigma' \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma''}{\sigma \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma''} \tag{2.9}$$

$$\nu \not\sqsubseteq \underline{\sigma}''(y). \tag{2.10}$$

By the inductive hypothesis on $\sigma' \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma''$, we see $\sigma'(y) = \sigma''(y)$. By (2.7), we obtain $\underline{\sigma}'(y) \sqsubseteq \underline{\sigma}''(y)$. Therefore, by (2.10),

$$\nu \sqcup \mathcal{A}[\![M]\!](F, \underline{\sigma}) \not\sqsubseteq \underline{\sigma}'(y). \tag{2.11}$$

By (2.8), (2.11) and the inductive hypothesis on $\sigma \models P \Rightarrow \sigma'$, we see $\sigma(y) = \sigma'(y)$. Summarizing, $\sigma(y) = \sigma''(y)$.

The proof for the case $\dfrac{\sigma \models M \Rightarrow \text{false}}{\sigma \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma}$ is straightforward.

(CONCAT) Assume

$$\frac{\sigma \models C \Rightarrow \sigma' \qquad \sigma' \models P \Rightarrow \sigma''}{\sigma \models C; P \Rightarrow \sigma''} \tag{2.12}$$

$$\underline{\sigma}'' = \mathcal{A}[\![P]\!](F, \underline{\sigma}', \nu) \tag{2.13}$$

$$\underline{\sigma}' = \mathcal{A}[\![C]\!](F, \underline{\sigma}, \nu) \tag{2.14}$$

$$\nu \not\sqsubseteq \underline{\sigma}''(y). \tag{2.15}$$

By (2.12), (2.13), (2.15) and the inductive hypothesis, we obtain $\sigma'(y) = \sigma''(y)$. It follows from (2.13) and (2.15) that $\underline{\sigma}'(y) \sqsubseteq \underline{\sigma}''(y)$. Hence, by (2.15),

$$\nu \not\sqsubseteq \underline{\sigma}'(y). \tag{2.16}$$

By (2.12), (2.14), (2.16) and the inductive hypothesis, $\sigma(y) = \sigma'(y)$ holds. Therefore, $\sigma(y) = \sigma''(y)$ holds.

The proof is similar for case (IF). □

Lemma 2.3.5. (*property of implicit flow*) If $\mathcal{A}[\![P]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}'$, $\sigma \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma'$ and $\nu \not\sqsubseteq \underline{\sigma}'(y)$ then $\sigma(y) = \sigma'(y)$.

Proof. Assume

$$\underline{\sigma}' = \mathcal{A}[\![P]\!](F, \underline{\sigma}, \nu) \tag{2.17}$$

$$\nu \not\sqsubseteq \underline{\sigma}'(y). \tag{2.18}$$

We will prove this lemma by induction on the application number of inference rules to deduce $\sigma \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma'$.
(Basis) In this case,

$$\frac{\sigma \models M \Rightarrow \text{false}}{\sigma \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma}.$$

The lemma holds obviously.

(Inductive step) Assume

$$\frac{\sigma \models M \Rightarrow \text{true} \qquad \sigma \models P \Rightarrow \sigma' \quad \sigma' \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma''}{\sigma \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma''} \quad . \tag{2.19}$$

By (2.17), (2.19), (2.18) and the inductive hypothesis, $\sigma'(y) = \sigma''(y)$ holds. By (2.17), (2.18) and the assumption $\sigma \models P \Rightarrow \sigma'$, we obtain $\sigma(y) = \sigma'(y)$ by lemma 2.3.4. Therefore, $\sigma(y) = \sigma''(y)$. □

The following lemma shows the relationship between the analysis result using the proposed algorithm and the noninterference property.

Lemma 2.3.6. Let $F = \mathcal{A}^*[\![Prog]\!]$.

(a) If $\mathcal{A}[\![M]\!](F, \underline{\sigma}) = \tau$, $\sigma_1 \models M \Rightarrow v_1$, $\sigma_2 \models M \Rightarrow v_2$ and $\forall x : \underline{\sigma}(x) \sqsubseteq \tau.\ \sigma_1(x) = \sigma_2(x)$, then $v_1 = v_2$.

(b) If $\mathcal{A}[\![P]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}'$, $\sigma_1 \models P \Rightarrow \sigma_1'$, $\sigma_2 \models P \Rightarrow \sigma_2'$, $\underline{\sigma}'(y) = \tau$ and $\forall x : \underline{\sigma}(x) \sqsubseteq \tau.\ \sigma_1(x) = \sigma_2(x)$, then $\sigma_1'(y) = \sigma_2'(y)$.

(c) If $\mathcal{A}[\![P]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}'$, $\sigma_1 \models P \Rightarrow v_1$, $\sigma_2 \models P \Rightarrow v_2$ and $\forall x : \underline{\sigma}(x) \sqsubseteq \underline{\sigma}'(ret).\ \sigma_1(x) = \sigma_2(x)$, then $v_1 = v_2$.

(Proof of *(a)*) (CALL) Assume

$$\tau = \mathcal{A}[\![f(M_1, \ldots, M_n)]\!](F, \underline{\sigma}) = F[f](\mathcal{A}[\![M_1]\!](F, \underline{\sigma}), \ldots, \mathcal{A}[\![M_n]\!](F, \underline{\sigma})) \tag{2.20}$$

$$\frac{\sigma_k \models M_i \Rightarrow u_{ki}\ (1 \le i \le n) \qquad \sigma_k' \models P_f \Rightarrow v_k}{\sigma_k \models f(M_1, \ldots, M_n) \Rightarrow v_k} \qquad (k = 1, 2) \tag{2.21}$$

$$\sigma_k' = \bot_{store}[x_1 := u_{k1}] \cdots [x_n := u_{kn}] \qquad (k = 1, 2) \tag{2.22}$$

$$\forall x : \underline{\sigma}(x) \sqsubseteq \tau.\ \sigma_1(x) = \sigma_2(x). \tag{2.23}$$

Since $F = \mathcal{A}[\![Prog]\!](F)$, by (2.1) and (2.20),

$$\begin{aligned}\tau &= \mathcal{A}[\![f(M_1,\ldots,M_n)]\!](F,\underline{\sigma}) \\ &= \mathcal{A}[\![P_f]\!](F, \bot_{\underline{store}}[x_1 := \mathcal{A}[\![M_1]\!](F,\underline{\sigma})]\cdots[x_n := \mathcal{A}[\![M_n]\!](F,\underline{\sigma})], \bot)(ret). \quad (2.24)\end{aligned}$$

If we let $\tau_i = \mathcal{A}[\![M_i]\!](F,\underline{\sigma})$ $(1 \le i \le n)$ and assume $\tau_i \sqsubseteq \tau$, then by (2.23) $\forall x : \underline{\sigma}(x) \sqsubseteq \tau_i.\ \sigma_1(x) = \sigma_2(x)$. The inductive hypothesis *(a)* implies $u_{1i} = u_{2i}$. Hence,

$$\forall i\,(1 \le i \le n) : \mathcal{A}[\![M_i]\!](F,\underline{\sigma}) \sqsubseteq \tau.\ u_{1i} = u_{2i}.$$

By (2.22),

$$\forall x : \bot_{\underline{store}}[x_1 := \mathcal{A}[\![M_1]\!](F,\underline{\sigma})]\cdots[x_n := \mathcal{A}[\![M_n]\!](F,\underline{\sigma})](x) \sqsubseteq \tau.\ \sigma_1'(x) = \sigma_2'(x). \quad (2.25)$$

The inductive hypothesis *(c)* together with (2.24), (2.21), (2.25) implies $v_1 = v_2$.
The other cases can be easily proved.

(Proof of *(b)*) (ASSIGN) Assume

$$\underline{\sigma}' = \mathcal{A}[\![x := M]\!](F,\underline{\sigma},\nu) = \underline{\sigma}[x := \mathcal{A}[\![M]\!](F,\underline{\sigma}) \sqcup \nu] \quad (2.26)$$

$$\frac{\sigma_k \models M \Rightarrow v_k}{\sigma_k \models x := M \Rightarrow \sigma_k'} \quad (k = 1,2) \quad (2.27)$$

$$\sigma_k' = \sigma[x := v_k] \quad (k = 1,2) \quad (2.28)$$

$$\tau = \underline{\sigma}'(y) \quad (2.29)$$

$$\forall z : \underline{\sigma}(z) \sqsubseteq \tau.\ \sigma_1(z) = \sigma_2(z). \quad (2.30)$$

If $x \neq y$, then $\underline{\sigma}'(y) = \underline{\sigma}(y)$ by (2.26). Hence, by (2.29) and (2.30) we obtain $\sigma_1(y) = \sigma_2(y)$. This implies $\sigma_1'(y) = \sigma_2'(y)$ by (2.28). If $x = y$, then $\underline{\sigma}'(y) = \mathcal{A}[\![M]\!](F,\underline{\sigma}) \sqcup \nu = \tau$ by (2.26) and (2.29). Therefore, $\mathcal{A}[\![M]\!](F,\underline{\sigma}) \sqsubseteq \tau$. By this fact and (2.30),

$$\forall z : \underline{\sigma}(z) \sqsubseteq \mathcal{A}[\![M]\!](F,\underline{\sigma}).\ \sigma_1(z) = \sigma_2(z). \quad (2.31)$$

By (2.27), (2.31) and the inductive hypothesis *(a)*, we obtain $v_1 = v_2$. That is, $\sigma_1'(y) = \sigma_2'(y)$.
(WHILE) Assume

$$\begin{aligned}&\underline{\sigma}'' = \mathcal{A}[\![\text{while } M \text{ do } P \text{ od}]\!](F,\underline{\sigma},\nu) = \\ &\quad fix(\lambda X.\underline{\sigma} \sqcup \mathcal{A}[\![P]\!](F,\underline{\sigma} \sqcup X, \nu \sqcup \mathcal{A}[\![M]\!](F,\underline{\sigma} \sqcup X))) \quad (2.32)\end{aligned}$$

$$\tau' = \underline{\sigma}''(y) \quad (2.33)$$

$$\forall x : \underline{\sigma}(x) \sqsubseteq \tau'.\ \sigma_1(x) = \sigma_2(x). \quad (2.34)$$

And let

$$\underline{\rho} = \mathcal{A}[\![P]\!](F, \underline{\sigma}, \nu \sqcup \tau) \tag{2.35}$$

$$\tau = \mathcal{A}[\![M]\!](F, \underline{\sigma}). \tag{2.36}$$

We use induction on the application number of (WHILE1). Let

$$\underline{\sigma}' = \underline{\sigma} \sqcup \underline{\rho}. \tag{2.37}$$

Then by the properties of a fixed point, (2.32) and (2.35), we obtain

$$\underline{\sigma}'' = \mathcal{A}[\![\text{while } M \text{ do } P \text{ od}]\!](F, \underline{\sigma}', \nu). \tag{2.38}$$

(i) Assume $\dfrac{\sigma_k \models M \Rightarrow \text{false}}{\sigma_k \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma_k}$ $(k = 1, 2)$. While $\underline{\sigma} \sqsubseteq \underline{\sigma}''$ by (2.32), $\underline{\sigma}(y) \sqsubseteq \underline{\sigma}''(y) = \tau'$ by (2.33). By (2.34), $\sigma_1(y) = \sigma_2(y)$.

(ii) Assume

$$\frac{\sigma_1 \models M \Rightarrow \text{true} \qquad \sigma_1 \models P \Rightarrow \sigma_1' \qquad \sigma_1' \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma_1''}{\sigma_1 \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma_1''} \tag{2.39}$$

$$\frac{\sigma_2 \models M \Rightarrow \text{false}}{\sigma_2 \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma_2}. \tag{2.40}$$

If $\tau \sqsubseteq \tau'$, by (2.36), (2.34), (2.39), (2.40) and the inductive hypothesis *(a)*, true = false, which is a contradiction. Hence, $\tau \not\sqsubseteq \tau'$. If $\nu \sqcup \tau \sqsubseteq \underline{\rho}(y)$, by (2.37), (2.38) and (2.33), we get $\tau \sqsubseteq \nu \sqcup \tau \sqsubseteq \underline{\rho}(y) \sqsubseteq \underline{\sigma}''(y) = \tau'$ which is a contradiction. Therefore,

$$\nu \sqcup \tau \not\sqsubseteq \underline{\rho}(y). \tag{2.41}$$

By (2.35), $\sigma_1 \models P \Rightarrow \sigma_1'$, (2.41) and lemma 2.3.4 we get $\sigma_1(y) = \sigma_1'(y)$. Also, by (2.35), $\sigma_1' \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma_1''$, and (2.41) imply $\sigma_1'(y) = \sigma_1''(y)$ by lemma 2.3.5. On the other hand, we can show $\sigma_1(y) = \sigma_2(y)$ in the same way as in case (i). Summarizing, $\sigma_1''(y) = \sigma_2(y)$.

(iii) Assume

$$\frac{\sigma_k \models M \Rightarrow \text{true} \quad \sigma_k \models P \Rightarrow \sigma_k' \quad \sigma_k' \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma_k''}{\sigma_k \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma_k''} \ (k = 1, 2). \tag{2.42}$$

If $\underline{\rho}(z) \sqsubseteq \tau'$, then by (2.35), (2.34), (2.42) and the inductive hypothesis *(b)*, $\sigma_1'(z) = \sigma_2'(z)$. Hence,

$$\forall x : \underline{\rho}(z) \sqsubseteq \tau'.\ \sigma_1'(z) = \sigma_2'(z). \tag{2.43}$$

By (2.38), (2.33), (2.42), (2.43) and the inductive hypothesis *(b)* of the application number of (WHILE1), $\sigma_1''(y) = \sigma_2''(y)$.

(CONCAT) Assume

$$\underline{\sigma}'' = \mathcal{A}[\![C; P]\!](F, \underline{\sigma}, \nu) = \mathcal{A}[\![P]\!](F, \underline{\sigma}', \nu) \tag{2.44}$$

$$\underline{\sigma}' = \mathcal{A}[\![C]\!](F, \underline{\sigma}, \nu) \tag{2.45}$$

$$\frac{\sigma_k \models C \Rightarrow \sigma_k' \qquad \sigma_k' \models P \Rightarrow \sigma_k''}{\sigma_k \models C; P \Rightarrow \sigma_k''} \ (k = 1, 2) \tag{2.46}$$

$$\tau = \underline{\sigma}''(y) \tag{2.47}$$

$$\forall x : \underline{\sigma}(x) \sqsubseteq \tau.\ \sigma_1(x) = \sigma_2(x). \tag{2.48}$$

If $\underline{\sigma}'(x) \sqsubseteq \tau$, by (2.45), (2.46), (2.48) and the inductive hypothesis *(b)*, $\sigma_1'(x) = \sigma_2'(x)$. That is,

$$\forall x : \underline{\sigma}'(x) \sqsubseteq \tau.\ \sigma_1'(x) = \sigma_2'(x). \tag{2.49}$$

By (2.44), (2.46), (2.47), (2.49) and the inductive hypothesis *(b)*, $\sigma_1''(y) = \sigma_2''(y)$.
The proof for the case of (IF) is easy.
The proof of *(c)* is similar to the proof of *(b)*. □

Theorem 2.3.7. The algorithm $\mathcal{A}^*[\![\cdot]\!]$ is sound.

Proof. By lemma 2.3.6*(c)*. □

Since the proposed algorithm is sound, if the algorithm answers "this program does not leak input information to the output" for some program P, P never leaks input information to the output. However, conversely, if the algorithm answers "this program may leak input information to the output" for some program P, it does not mean P always leaks input information to the output. In other words, the completeness property is not satisfied. Intuitively, to satisfy the completeness property, the semantics of expressions which appear in the conditional clause of if and while commands must be accurately analyzed. However, since this is an undecidable problem, an algorithm which satisfies the completeness property does not exist. Nevertheless, under the prerequisite that the semantics (interpretation) of the built-in functions is freely given, a weak completeness property [B76] is satisfied, which means that there exist an interpretation of the built-in functions and inputs to the program that actually cause the information flow which is detected by the analysis algorithm.

```
    main(x, y) local z
    {
 1:    if y < 0 then
 2:       return 0
       else
 3:       y := 0;
 4:       while y < 10 do
 5:          z := y;
 6:          y := y + g(y, x)
          od;
 7:       return z
       fi
    }

    g(x, y)
    {
 8:    if y <= 0 then
 9:       return 0
       else
10:       return g(x, y-1) + 1
       fi
    }
```

Figure 2.4. A program to analyze.

2.3.5 Time Complexity

In this subsection, the time complexity of the algorithm $\mathcal{A}^*[\![\cdot]\!]$ presented in section 2.3.1 is examined.

If the algorithm $\mathcal{A}^*[\![\cdot]\!]$ which includes the fixpoint operation in (WHILE) and (2.2), is implemented by using a simple iteration, the time complexity can become large. $\mathcal{A}^*[\![\cdot]\!]$ can be computed more efficiently by constructing a directed graph which represents a control flow of a program to be analyzed (See Figure 2.5 for an example in Figure 2.4) and searching within this graph.

Each part enclosed in ellipses in Figure 2.5 represents the results obtained by applying the algorithm $\mathcal{A}$ to one of the commands within the program. For example, the assignment command $y := y + g(y, x)$, which is line 6 in Figure 2.4, is transformed to $y := y \sqcup g(y, x) \sqcup \nu_2$ (ν_2 is a variable that was added to represent the implicit flow), and the term on the right hand side is represented as a tree (ellipse 6 in Figure 2.5). This transformation differs from the method described in subsection 2.3.1 in that it contains the variables and functions obtained by the transformation as subterm. Also, for each n-ary function f in Figure 2.5, the constants $\tau_{f,1}, \ldots, \tau_{f,n}$ representing the security classes of the actual arguments of f are provided, and commands for assigning these constants to the formal arguments are added. The arrows drawn between the ellipses

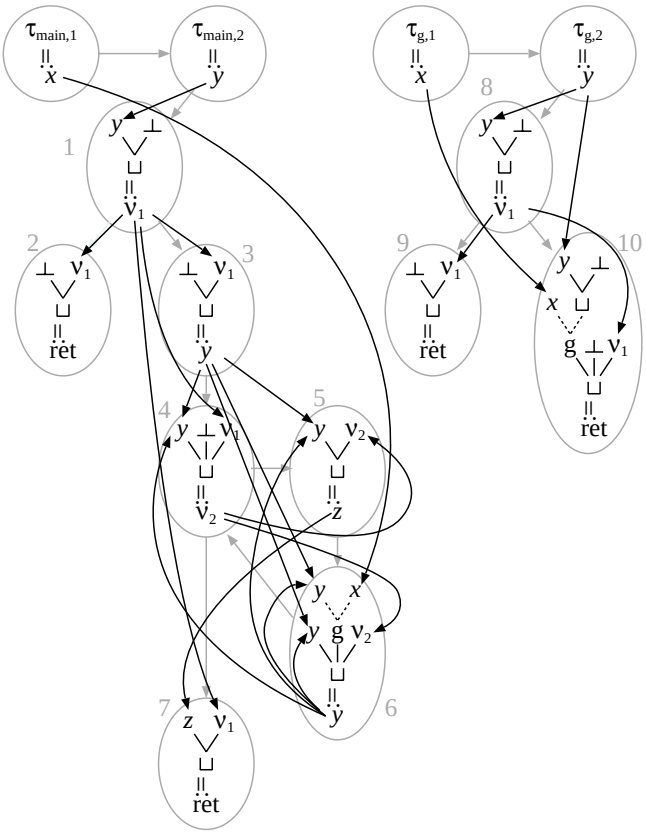

Figure 2.5. An information flow graph.

in Figure 2.5 represent relationships between the definitions and uses of variables. For example, there is a possibility that the y value defined in ellipse 3 or ellipse 6 will be used in the $\mathcal{A}[\![Prog]\!]$ calculation for the y on the right hand side in ellipse 5. This is represented by the arrows to the y in ellipse 5 from the left hand sides in ellipse 3 and ellipse 6.

The $\mathcal{A}[\![Prog]\!]$ calculation described in subsection 2.3.1 is equivalent to checking whether or not there exists a path in Figure 2.5 from each of the constants $\tau_{f,1}, \ldots, \tau_{f,n}$ representing the security classes of the actual arguments to *ret*. In other words, if we let

$$\mathcal{A}[\![Prog]\!](F)[f](\tau_1, \ldots, \tau_n) = \tau_{i_1} \sqcup \ldots \sqcup \tau_{i_m}$$

(where $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$; since the only operations which appear in the analysis algorithm are $\bot$ and $\sqcup$, $\mathcal{A}[\![Prog]\!](F)[f](\tau_1, \ldots, \tau_n)$ can always be written this way),

the fact that a path exists from $\tau_{f,j}$ to *ret* on the graph is equal to the fact that $j \in \{i_1, \ldots, i_m\}$. The F representing the security class of a function here corresponds to the state of the edges between the vertex for a function call within the graph and its children (edges drawn with dashed lines in Figure 2.5). For example, the fact that $F[g](\tau_1, \tau_2) = \tau_2$ corresponds to the fact that among the edges between a vertex corresponding to a call of g (both of the two calls) and its children. That is, the edge corresponding to the first argument is unusable (information of the first argument does not flow into the function value) and the edge corresponding to the second argument is usable (information of the second argument can flow into the function value). Therefore, when (1) all of these edges drawn with dashed lines are first marked as being unusable and (2) it is known that function f enables information of the i-th argument to flow into its function value (there exists a path from $\tau_{f,j}$ to *ret*), the edge corresponding to the i-th argument of function f is known to be usable. This can be done by searching backwards along directed edges from each *ret* vertex. However, when the constant $\tau_{f,j}$ representing the security class of an actual argument is reached, the edge corresponding to the i-th argument of each call of function f should be changed to usable, and if a vertex corresponding to a call of f has already been visited, the vertex corresponding to the i-th argument should be added as a search starting position to the processing wait list.

If we let V the number of vertices of this graph and E the number of edges, then the graph can be searched in $O(V + E)$ time. If we let N the description size of the program to be analyzed, then $V = O(N)$ and $E = O(N^2)$. On the other hand, to construct this graph, the relationships between definitions and uses must be obtained. This can be accomplished by calculating the reaching definition [ASU86]. If we let d the maximum nesting depth of while commands in the program to be analyzed, the reaching definition of each command is obtained by performing set operations related to a subset of the assignment statement $O(N \cdot d)$ times. Since $O(N)$ time is required for the first set operation and $d = O(N)$, the total calculation time for the reaching distance is $O(N^3)$. However, in an actual program, since the nesting depth of the while commands often can be considered to be less than or equal to a constant or since a technique such as bit vectors [ASU86, A98] can be used to speed up set operations. The calculation time often will be $O(N)$ or $O(N^2)$[A98].

2.4. An Extended Model

2.4.1 Extension of Analysis for Built-in Function

Generally, methods of statically analyzing the information flow of a program including the method proposed in the previous sections assume that information in every argument of a built-in function flows into the output and define the security class of the output as the least upper bound of the security classes of all of the arguments. With this definition, the security class of the output is analysed to be *high* when the security class of an argument is *high* even if information in the argument does not flow into the output. To solve this problem, the model is extended so that the security class should not actually increase. We extend the model so that the security class of the output of an arbitrary built-in function can be defined according to a given analysis policy.

The algorithm $\mathcal{A}$ in the previous section has been defined for any built-in function θ as:

(PRIM) $\mathcal{A}[\![\theta(M_1, \ldots, M_n)]\!](F, \underline{\sigma}) = \bigsqcup_{1 \leq i \leq n} \mathcal{A}[\![M_i]\!](F, \underline{\sigma})$.

This means that we assume information contained in each argument may flow into the result of the operation $\theta_{\mathcal{I}}$. However, this assumption is too conservative for a certain function. For example, if a function $\theta_{\mathcal{I}}$ is defined as $\theta_{\mathcal{I}}(x, y) = x$, then it is clear that information in the second argument does not flow into the result of the operation. Another example is an encryption. Assume that for a plain text d and an encryption key k, the result of the operation $E_{\mathcal{I}}(d, k)$ is the cipher text of d with key k. We may consider that the SC of $E(x, y)$ is *low* even if the SCs of x and y are both *high*.

To express the above mentioned properties of particular built-in operations, we generalize the above definition as:

(PRIM) $\mathcal{A}[\![\theta(M_1, \ldots, M_n)]\!](F, \underline{\sigma}) = \mathcal{B}[\![\theta]\!](\mathcal{A}[\![M_1]\!](F, \underline{\sigma}), \ldots, \mathcal{A}[\![M_n]\!](F, \underline{\sigma}))$,

where $\mathcal{B}[\![\theta]\!]$ is an arbitrary monotonic total function on sc:

$$\mathcal{B}[\![\theta]\!] : sc \times \cdots \times sc \to sc.$$

The extension in this section is the assumption that $\mathcal{B}[\![\theta]\!]$ can be defined for an arbitrary built-in function θ according to a given analysis policy. In particular, $\mathcal{B}[\![\theta]\!](\tau_1, \ldots, \tau_n) = \bigsqcup_{1 \leq i \leq n} \tau_i$ for the original definition of $\mathcal{A}$.

We use the extended model below to show three examples in which the analysis result of a built-in function is more accurate approximation than the original analysis method.

Example 2.4.1 (nonstrict function). For the built-in function $\theta_{\mathcal{I}}(x, y) = x$, since information in argument y cannot be obtained from the output x of the function, we can define $\mathcal{B}[\![\theta]\!](\tau_1, \tau_2) = \tau_1$. □

Example 2.4.2 (encryption function). Let E be an encryption function which takes the plaintext and the encryption key as its arguments. If we assume that information contained in the plaintext and encryption key cannot be obtained no matter how an illegal attacker manipulates the cipher text, then we can define $\mathcal{B}[\![E]\!](high, high) = low$. □

Example 2.4.2 is useful when the analysis policy is: "Can illegal attacker obtain information in plaintext or encryption key from program output without decryption key? Under the assumption that an illegal attacker cannot decode a cipher text without a decryption key."

Example 2.4.3 (average value calculation function). Let the average value calculation function Ave be a function for calculating the average value of i arguments $n_1, \ldots, n_i$. If we assume that information contained in the arguments cannot be obtained from the average value no matter what operations are performed, the we can define $\mathcal{B}[\![Ave]\!](\tau_1, \ldots, \tau_i) = low$. □

Defining $\mathcal{B}[\![Ave]\!](\tau_1, \ldots, \tau_i) = low$ for the average value calculation function is valid when the number of arguments i is sufficiently large. For example, since $Ave_{\mathcal{I}}(n_1) = n_1$ when $i = 1$, the assumption that "information contained in the arguments cannot be obtained from the average value" does not hold, and defining $\mathcal{B}[\![Ave]\!](\tau_1) = low$ is clearly unnatural. It depends of an analysis policy what is the minimum number of arguments with which we can assume that information contained in the arguments cannot be obtained from the average value no matter what operation are performed. We should not use the definition $\mathcal{B}[\![Ave]\!](\tau_1, \ldots, \tau_i) = low$ if it is unknown whether or not a large number of arguments are assigned to the Ave function within the program to be analyzed.

The generalized algorithm using the new definition is no longer sound in the sense of definition 2.3.1. Suppose that we define $\mathcal{B}[\![E]\!](\tau_1, \tau_2) = low$, and consider a program

$$Prog = \{main(x, y) \ \ \{ \text{ return } E(x, y) \ \} \}.$$

$\mathcal{A}^*[\![Prog]\!][main](high, high) = low$ holds while for distinct plain texts d_1, d_2 and a key k, $E_{\mathcal{I}}(d_1, k) \neq E_{\mathcal{I}}(d_2, k)$. Hence $\mathcal{A}^*[\![\cdot]\!]$ is not sound. Intuitively, the fact that the SC of expression $E(x, y)$ is inferred as low means that we cannot recover information

contained in the arguments x, y from the result of the encryption. In other words, $E_{\mathcal{I}}(d_1, k)$ and $E_{\mathcal{I}}(d_2, k)$ are indistinguishable with respect to the information in the arguments. To express this indistinguishability, we introduce the following notions.

A relation R on type val is called a congruence relation if R is an equivalence relation which satisfies:

> for each n-ary built-in operator θ, if $c_i \ R \ c'_i$ for $1 \leq i \leq n$ then
> $\theta_{\mathcal{I}}(c_1, \ldots, c_n) \ R \ \theta_{\mathcal{I}}(c'_1, \ldots, c'_n)$.

In the following, we assume that a particular congruence relation $\sim$ is given. For v, v' of type val, if $v \sim v'$ then we say that v and v' are indistinguishable. By the definition, if v_i and v'_i for $1 \leq i \leq n$ are indistinguishable then for any built-in operator θ, $\theta_{\mathcal{I}}(c_1, \ldots, c_n)$ and $\theta_{\mathcal{I}}(c'_1, \ldots, c'_n)$ are also indistinguishable. This implies that once v and v' become indistinguishable, we cannot obtain any information to distinguish v and v' through any operations.

Next, we require $\mathcal{B}[\![\cdot]\!]$ to satisfy the following condition.

Condition 2.4.4. Assume $\mathcal{B}[\![\theta]\!](\tau_1, \ldots, \tau_n) = \tau$ for an n-ary built-in operator θ. Let c_i, c'_i be of type val $(1 \leq i \leq n)$. If $c_j \sim c'_j$ for each j $(1 \leq j \leq n)$ such that $\tau_j \sqsubseteq \tau$, then $\theta_{\mathcal{I}}(c_1, \ldots, c_n) \sim \theta_{\mathcal{I}}(c'_1, \ldots, c'_n)$. □

The above condition states that:

> Let $\mathcal{B}[\![\theta]\!](\tau_1, \ldots, \tau_n) = \tau$. Assume that arguments of θ are changed from $c_1, \ldots, c_n$ to $c'_1, \ldots, c'_n$. As long as c_j and c'_j are indistinguishable for each argument position j such that $\tau_j \sqsubseteq \tau$, $\theta_{\mathcal{I}}(c_1, \ldots, c_n)$ and $\theta_{\mathcal{I}}(c'_1, \ldots, c'_n)$ remain indistinguishable.

In example 2.4.1, condition 2.4.4 requires $\theta_{\mathcal{I}}(c_1, c_2) \sim \theta_{\mathcal{I}}(c'_1, c'_2)$ for arbitrary values c_1, c'_1, c_2 and c'_2. Since $\theta_{\mathcal{I}}(c_1, c_2) = c_1 \sim c'_1 = \theta_{\mathcal{I}}(c'_1, c'_2)$ if $c_1 \sim c'_1$, then condition 2.4.4 is satisfied for any congruence relation $\sim$.

In example 2.4.2, condition 2.4.4 requires $E_{\mathcal{I}}(d, k) \sim E_{\mathcal{I}}(d', k)$ for arbitrary plain text d and d' and arbitrary keys k and k'. This means that information in the plain texts or the keys cannot be obtained from the cipher text.

In example 2.4.3, condition 2.4.4 requires $Ave_{\mathcal{I}}(n_1, \ldots, n_i) \sim Ave_{\mathcal{I}}(n'_1, \ldots, n'_i)$ for arbitrary integers $n_1, \ldots n_i$, $n'_1, \ldots, n'_i$. This means that information contained in the arguments cannot be obtained no matter how the average value is manipulated.

Now we can define the soundness by using the notion of indistinguishability as follows:

Definition 2.4.5 (generalized soundness). Let $\sim$ be a congruence relation. We say that an algorithm $\mathcal{A}^*[\![\cdot]\!]$ is sound (with respect to $\sim$) if the following condition holds:

If $\mathcal{A}^*[\![Prog]\!][main](\tau_1, \ldots, \tau_n) = \tau$,
$\bot_{store} \models main(v_1, \ldots, v_n) \Rightarrow v$, $\bot_{store} \models main(v'_1, \ldots, v'_n) \Rightarrow v'$, and
$\forall i\, (1 \leq i \leq n) : \tau_i \sqsubseteq \tau.\, v_i \sim v'_i$
then $v \sim v'$ holds. □

It is not difficult to prove the following theorem in a similar way to the proof of theorem 2.3.7.

Theorem 2.4.6. If condition 2.4.4 is satisfied, then the generalized algorithm $\mathcal{A}^*[\![\cdot]\!]$ is sound in the sense of definition 2.4.5. □

2.4.2 An Example

Consider the following program which provides a digital signature only to data owned by authenticated user.

```
main(id, pass, d) {
  if (verify(id, pass)) then
    return sign(d)
  else
    return d
  fi
}
```

Input of this program is a user ID (id), user password ($pass$), and data (d) to which the digital signature is to be provided. The function *verify* is a built-in function which takes the user ID and user password as arguments and returns "true" if the user is authenticated and returns "false" if not. And the function *sign* is a built-in function which takes the data as an argument, attaches a digital signature to that data and returns the signed data. Assume that the security class of the user ID and user password is the maximum value $\top$ and the security class of the input data is τ. If this program analyzed without using the extended model ($\mathcal{A}[\![verify]\!](\tau_1, \tau_2) = \tau_1 \sqcup \tau_2$), then information of the condition part of the if command flows into the return command due to an implicit flow and the analysis algorithm determines that the security class of the output data will be $\top$. Assume that there is an user who is the owner of data d and assume the user can read data of security class lower than τ. The analysis algorithm

says if the user attach the digital signature to the data d the security class will become $\top$. This means even the owner of data d may no longer be able to read his/her own data if a digital signature is attached to d. Therefore, we can say the analysis result is unnatural.

As a natural assumption, we consider that no information of the user ID and user password can be obtained no matter how the output of the *verify* function (true or false) is manipulated and define $\mathcal{B}[\![verify]\!](\top, \top) = \bot$. Using this definition, the analysis algorithm for this program determines that the security class of the output data will be τ. This analysis result is natural since it means that signed data $sign(d)$ has the same security class as the input d. Also, under the assumption (condition 2.4.4) which says "$verify_{\mathcal{I}}(c_1, c_2) \sim verify_{\mathcal{I}}(c_1', c_2')$ for arbitrary values c_1, c_1', c_2 and c_2'" or in other words "no information of the user ID and user password can be obtained no matter how the output of the *verify* function (true or false) is manipulated," this analysis guarantees that the information of the user ID and user password will not leak into the digitally signed data.

2.5. Conclusion of Chapter 2

In this chapter, we have proposed an algorithm which can statically analyze the information flow of a procedural program containing recursive definitions. It has been shown that the algorithm is sound and that the algorithm can be executed in polynomial time in the size of an input program. In addition, we generalized the definition for an arbitrary built-in function and extended the model so that the security class of the result of an arbitrary built-in function can be defined according to a given analysis policy. Finally, we proved the soundness of the extended model. In [Y01], the proposed algorithm is extended to be able to analyze a program which may contain global variables and a prototypic analysis system has been implemented. Table 2.1 shows the execution time to analyze sample programs by the implemented system (CPU:1.5GHz Pentium 4, main memory: 512MB). However, in this prototype system, only two values $\{high, low\}(low \sqsubseteq high)$ can be used as security classes.

The ticket reservation system in Table 2.1 contains a certification module which deals with a credit card number. It is assumed that the security class of the credit card number which is the input of the system is *high*. The analysis determined that the security class could be *high* for 13 outputs among the 36 output commands (among the 13 outputs for which the security class became *high*, 7 existed within the certification module). Next, as a sample application of the extended model, we assume the certi-

Table 2.1. Analysis time

Program	Number of lines	Average analysis time (sec)
Ticket reservation system	419	0.050
Sorting algorithm	825	0.130
A program library	2471	2.270

fication module as a built-in function and the security class of the operation result is assumed to be *low*. As a result, the only outputs for which the security class became *high* were the 7 within the certification module and for the remaining 6 outputs, which are outside of the certification module, the security class was *low*. Also, for the every outputs which the security class could become *high*, credit card information was flowed into all those outputs.

Extending the proposed method so that we can analyze a program which has pointers and/or object-oriented features is a future study.

Using the method proposed in section 2.4, we can appropriately analyze one-way functions. However, trapdoor functions (e.g., a plaintext can be obtained by decrypting the encrypted one) cannot be appropriately analyzed. In [OYF03], the information flow analysis algorithm proposed in this chapter is extended so that can handle trapdoor functions.

Chapter 3

Policy Controlled System and Its Model Checking

3.1. Introduction

This chapter proposes a security assurance method for a Discretionary Access Control (DAC) based system. DAC assumes that the owner of an object controls access permissions to the object. It is at the owner's discretion to assign security policy (permission) to objects. In a conventional DAC model, only permission (positive authorization) and prohibition (negative authorization) can be specified. Unix file permission is a typical example of conventional DAC model. However, an obligation policy is also important, which describes that a specified subject is obliged to perform a specified action when a certain event occurs under a certain condition.

In section 3.2, we introduce a simple but useful policy specification language which can specify not only permission and prohibition policies but also obligation policies. Proposed language is suitable for distributed policy control and is used in policy controlled system (PCS) which is proposed in section 3.2. PCS is a system in which each object has its own security policy and objects' behaviors are autonomously controlled based on those policies when they interact with one another. Operational semantics of PCS is formally defined.

In section 3.3 and 3.4, using a model checking technique for pushdown system (PDS), we propose a method for verifying a safety property of a PCS. We define the (safety) verification problem for PCS as the problem to decide for a given PCS S and a goal (called *safety property*) Ψ, whether every reachable state of P satisfies Ψ. As defined later, Ψ is represented as a regular language.

In section 3.3, first, a pushdown system (PDS) and Esparza et al.'s results on model checking for PDS are reviewed. Next, we propose an efficient algorithm which constructs an NFA accepting the set of all reachable states of a given PDS.

Section 3.4 is devoted to presenting our verification method; we provide an abstraction of PDS from PCS. Example verification results obtained by our verification tool are also illustrated.

Section 3.5 concludes the chapter.

3.2. Policy Controlled System

We introduce a policy controlled multi-object system, and describe its formal semantics.

A *policy controlled system* (PCS) is a tuple $S = (O, Prog, Policy)$ where O is a finite set of objects, *Prog* is a finite set of bodies of all methods in O, and *Policy* is a finite set of policies. In section 3.2.1, we propose a simple policy specification language. In section 3.2.2, we formally define the structure and the behavior of a PCS.

3.2.1 Policy Specification Language

In a traditional access control model, 3-tuple (s, t, a) means that "subject s performs action a on target t." In an object-oriented model, (s, t, a) corresponds to "subject s executes method a on target t." There are four kinds of basic access control policy for (s, t, a) as follows.

- positive authorization (or permission or right) : s is permitted to perform a on t.
- negative authorization (or prohibition) : s is forbidden to perform a on t by the target's policy.
- refrainment : s is forbidden to perform a on t by the subject's policy.
- obligation : s is obliged to perform a on t (when a specific event has occurred).

We write $t.a \leftarrow s$ to denote (s, t, a). Furthermore, we write $t.a(p_1, \ldots, p_n) \leftarrow s$ to denote that s performs a with actual arguments $p_1, \ldots, p_n$ on t. In the following, auth+, auth−, oblg and refrain stand for positive authorization policy, negative authorization policy, obligation policy and refrainment policy, respectively.

In our model, each object has its own policy. When more than one object interact with one another, the execution of every method should meet all the policies of the objects which participate in the method execution. For example, object A can play the

⟨policy⟩ := 'policy' ⟨mode1⟩ ⟨policy name⟩
['var' ⟨variable declaration⟩+]
⟨operation unit1⟩+ ['if' ⟨condition⟩]
% This rule defines the syntax of auth± and refrain policies.
| 'policy' ⟨mode2⟩ ⟨policy name⟩
['var' ⟨variable declaration⟩+]
⟨operation unit2⟩+ 'on' ⟨event⟩ ['if' ⟨condition⟩]
%This rule defines the syntax of an obligation policy.
⟨condition⟩ := ⟨expression⟩
% The type of ⟨expression⟩ should be boolean.
⟨mode1⟩ := 'auth+' | 'auth−' | 'refrain'
⟨mode2⟩ := 'oblg'
⟨operation unit1⟩ := ⟨object⟩'.'⟨action⟩ ['←' ⟨object⟩]
⟨operation unit2⟩ := ⟨object⟩'.'⟨action⟩ '(' ⟨expression⟩∗ ')' ['←' ⟨object⟩]
% See Table 3.1 for the microsyntax of ⟨operation unit1⟩ and ⟨operation unit2⟩.
⟨object⟩ := ⟨identifier⟩ | 'this' % 'this' means the self object.
⟨expression⟩ := '(' ⟨expression⟩ ')' | ⟨object⟩'.'⟨attribute⟩ |
⟨expression⟩ ⟨binary operator⟩ ⟨expression⟩ |
⟨unary operator⟩ ⟨expression⟩ | ⟨constant⟩ | ⟨variable⟩
⟨event⟩ := 'beginning of' ⟨operation unit3⟩ | 'end of' ⟨operation unit3⟩
⟨operation unit3⟩ := ⟨operation unit1⟩
⟨variable declaration⟩ := ⟨variable⟩+ ':' ⟨type⟩
⟨policy name⟩,⟨action⟩,⟨attribute⟩,⟨variable⟩ := ⟨identifier⟩

Figure 3.1. Syntax of a policy specification language

movie contained in object B by executing method $B.play$ only when both the policies of A and B permit A to execute $B.play$. We use the reserved word "this" to denote the self object, namely, the object which has that policy.

We define the syntax of a policy specification language using BNF notation as Figure 3.1. The policy specification language is a set of ⟨policy⟩.

Note that:

- ⟨...⟩ are nonterminal symbols, A | B is a choice of A and B, [A] means A is an option, A∗ stands for 0 or more repetition of A, A+ stands for 1 or more repetition of A and 'α' stands for α itself as terminal symbols.
- The microsyntax of ⟨binary operator⟩, ⟨unary operator⟩, ⟨constant⟩, ⟨type⟩ and ⟨identifier⟩ are omitted.
- In Table 3.1, x and y stand for any objects other than 'this.'

Table 3.1. Form of ⟨operation unit1⟩ and ⟨operation unit2⟩

	this.m←this	$x.m$←this	this.m←y	$x.m$←y
auth+	√		√	
auth−			√	
refrain	√	√		
oblg	√	√		

- As defined in the above BNF rules, ⟨operation unit3⟩ is used only in the event clause of obligation mode, and is allowed to have the form this.m←this, $x.m$←this or this.m←y where x and y stand for any objects other than 'this.'

Using the policy specification language, basic access control policies can be written as follows.

(1) positive authorization
policy auth+ *policy_name* this.m←B if *Cond*
"If *Cond* holds, then object B is permitted to execute method m on this object."

(2) negative authorization
policy auth− *policy_name* this.m←B if *Cond*
"If *Cond* holds, then object B is forbidden to execute m on this object."

(3) refrainment
policy refrain *policy_name* B.m←this if *Cond*
"If *Cond* holds, then this object is forbidden to execute m on object B."

(4) obligation
policy oblg *policy_name* B.m←this on *Event* if *Cond*
"If *Cond* holds when *Event* occurs, then this object is obliged to execute m on object B."
Event should be a time instant (without duration). In the above policy specification,

- if *Event* ="beginning of $D.m' \leftarrow F$" then this object must perform m on B just before F performs m' on D.
- if *Event* ="end of $D.m' \leftarrow F$" then this object must perform m on B just after F performs m' on D.

Example 3.2.1.

Policy of digital contents (Playing contents)**:** Consider an object with media contents such as digital audio and video. This object may specify the following policy. Let x be an arbitrary user object.

"If x is the owner of this object and if x is older than or equal to 20 years, then x may play the contents involved in this object."

"Just after the execution of *play* method, x must execute *pay* method with actual arguments x, B and \$10.00. That is, when x has played the contents, then x must pay \$10.00 to B."

```
policy auth+ Play_1
  var x: user
  this.play←x if this.owner==x, x.age>=20
policy oblg Play_2
  var x: user
  this.pay(x,B,$10.00)←this on end of this.play←x
```

Policy of a user (Refrainment from playing contents)**:** A user object (or a personal computer of the user) may have the following policy. Let z be an arbitrary object which has a movie as contents.

"If the current user is under 18 years old and if the type of the contents is 'v', then this user object cannot play z."

```
policy refrain Age_Check
  var z: content
  z.play←this if this.age<18, z.type==v
```

□

3.2.2 Formal Semantics of PCS

In this section, we formally define a multi-object system in which the behavior of each object is controlled by specified policies. An object in the system calls a method of another object (or itself) along a given program, and the invocation of the called method is permitted or forbidden complying with both the caller's and callee's policies. Moreover, an invocation and an ending of a method may cause other obligatory method calls specified by the policies.

In subsection 3.2.2, we define a simple model of objects and programs that is a behavior of PCS without policies. In subsection 3.2.2, we introduce several concepts about policies and define the behavior of a system with policies. In subsection 3.2.2, the

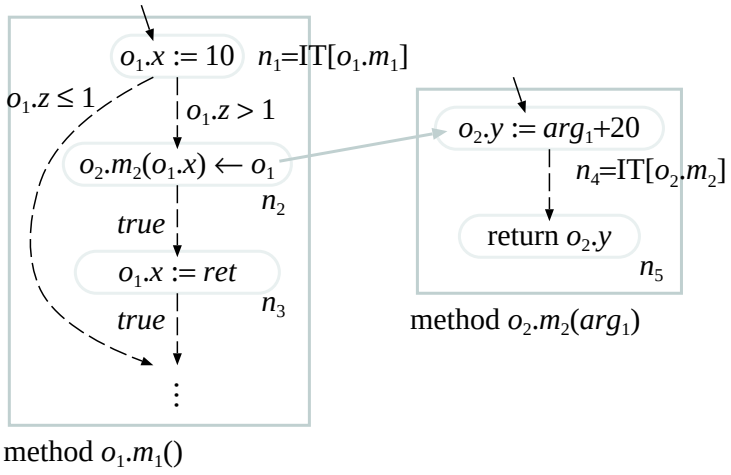

Figure 3.2. A sample program

system is extended by introducing an exception handling function. In our model, an exception occurs when a forbidden method call is requested, and thus policy violations are handled by the uniform exception handling function.

Behavior of PCS without Policies

Objects Each object has a finite number of attributes and methods defined by its class. Assume that an object o has attributes $a_1, a_2, \ldots, a_k$. When the value of a_i is v_i for $1 \leq i \leq k$, the state of o is represented by the tuple $state = (v_1, v_2, \ldots, v_k)$. We may write $o.a$ and $o.m$ to represent an attribute a and a method m of an object o, respectively.

Let O be a finite set of objects. Assuming a total order of objects in O, we let O be an ordered set $(o_1, o_2, \ldots, o_n)$. A global state of O is an n-tuple $\sigma = (state_1, state_2, \ldots, state_n)$, where $state_j$ is a state of o_j for $1 \leq j \leq n$. Let $\sigma(e)$ denote the value of an expression e at a global state σ and let $\sigma[o.a := v]$ denote the global state which is the same as σ except that the value of the attribute $o.a$ is v.

The body of a method is a program described as follows.

Programs The body of a method $o.m$ is a program which is represented by a directed graph as shown in Figure 3.2. A program is a tuple $(\mathrm{NO}, \mathrm{TG}, \mathrm{IS}, \mathrm{IT}, \mathrm{VAR})$. In the following we write NO$[o.m]$, TG$[o.m]$, and so on to represent each of the five components of the body of a method $o.m$. Let EXP$[o.m]$ be the set of expressions which consist of

built-in functions, attributes of o, and variables in VAR[$o.m$] (defined below).

- NO[$o.m$] is a set of nodes which represent program points. We assume that for any objects o_1, o_2, and methods m_1, m_2, NO[$o_1.m_1$] and NO[$o_2.m_2$] are disjoint unless $o_1 = o_2$ and $m_1 = m_2$.

- TG[$o.m$] $\subseteq$ NO[$o.m$] $\times$ EXP[$o.m$] $\times$ NO[$o.m$] is a set of edges called transfer edges. For any $n_1, n_2 \in$ NO[$o.m$], $n_1 \xrightarrow{e} n_2$ denotes $(n_1, e, n_2) \in$ TG[$o.m$], which represents that the control can move to n_2 just after the execution of n_1 if the value of e is *true*.

- IT[$o.m$] $\in$ NO[$o.m$] is the entry point of the program and is called the initial node.

- IS[$o.m$] is a mapping from a node to its label. The label of a node represents an atomic action and is one of the following forms.

  - $o_2.m_2(e_1, \ldots, e_k) \leftarrow o$ Invoke $o_2.m_2$ with the arguments $e_1, \ldots, e_k \in$ EXP[$o.m$]; move the control to IT[$o_2.m_2$] and assign the values of expressions $e_1, \ldots, e_k$ to the parameters $arg_1, \ldots, arg_k$ in VAR[$o_2.m_2$], respectively.
  - return e Return to the caller method and move the control to the next node. The value of $e \in$ EXP[$o.m$] is returned and is assigned to the special local variable *ret* of the caller method.
  - $o.a := e$ Assign the value of $e \in$ EXP[$o.m$] to the attribute a of o itself.
  - $r := e$ Assign the value of $e \in$ EXP[$o.m$] to the local variable $r \in$ VAR[$o.m$].

  In the following, let IS be the mapping which is the union of IS[$o.m$] for every method m of every object o.

- VAR[$o.m$] is a set of local variables. Assuming that VAR[$o.m$] is an ordered set $(r_1, \ldots, r_k)$, a state of the local variables of $o.m$ is represented by a tuple $\mu = (v_1, \ldots, v_k)$ of their values. We define $\mu(e)$ and $\mu[r := v]$ in the same way as the global state of objects. The value of an expression $e \in$ EXP[$o.m$] at a global state σ and a state μ of local variables is $\sigma \circ \mu(e) = \sigma(\mu(e))$. The state of local variables in which the values of all variables are undefined is denoted by $\perp$.

Transition System The multi-object system consists of a set O of objects and a control stack. The control stack is a sequence of an arbitrary number of stack frames.

$$\text{(GASSIGN)}\quad \frac{\mathrm{IS}(n) = \text{“}o.a := e\text{”} \qquad n \xrightarrow{e_2} n_2 \qquad \sigma' = \sigma[o.a := \sigma \circ \mu(e)] \qquad \sigma' \circ \mu(e_2) = \mathit{true}}{(\sigma,\ \ (n, \mu) : \xi) \to (\sigma',\ \ (n_2, \mu) : \xi)}$$

$$\text{(LASSIGN)}\quad \frac{\mathrm{IS}(n) = \text{“}r := e\text{”} \qquad n \xrightarrow{e_2} n_2 \qquad \mu' = \mu[r := \sigma \circ \mu(e)] \qquad \sigma \circ \mu'(e_2) = \mathit{true}}{(\sigma,\ \ (n, \mu) : \xi) \to (\sigma,\ \ (n_2, \mu') : \xi)}$$

$$\text{(CALL)}\quad \frac{\mathrm{IS}(n) = \text{“}o_2.m_2(e_1, \ldots, e_k) \leftarrow o_1\text{”} \qquad f_2 = (\mathrm{IT}[o_2.m_2], \bot[arg_1 := \sigma \circ \mu(e_1)] \cdots [arg_k := \sigma \circ \mu(e_k)])}{(\sigma,\ \ (n, \mu) : \xi) \to (\sigma,\ \ f_2 : (n, \mu) : \xi)}$$

$$\text{(RETURN)}\quad \frac{\mathrm{IS}(m) = \text{“return } e\text{”} \qquad \mathrm{IS}(n_1) = \text{“}o_2.m_2(e_1, \ldots, e_k) \leftarrow o_1\text{”} \qquad f_2 = (n_1, \mu[\mathit{ret} := \sigma \circ \mu_2(e)], \mathit{true})}{(\sigma,\ \ (m, \mu_2) : (n_1, \mu) : \xi) \to (\sigma,\ \ f_2 : \xi)}$$

$$\text{(FINISH)}\quad \frac{n_1 \xrightarrow{e} n_2 \qquad \sigma \circ \mu(e) = \mathit{true}}{(\sigma,\ \ (\not{n}_1, \mu) : \xi) \to (\sigma,\ \ (n_2, \mu) : \xi)}$$

Figure 3.3. Inference rules which define the transition relation (1)

Each stack frame (or simply, frame) is a triple (n, μ, ef), where $n \in \mathrm{NO}[o.m]$ for a method $o.m$, μ a state of the local variables of $o.m$, and ef a truth value. The frame represents that the control is at n in method $o.m$ with the state μ of the local variables, and if $ef = \mathit{true}$, then it also represents that the label of n is $o_2.m_2(e_1, \ldots, e_k) \leftarrow o$ and the control has already reached a return node in the callee method $o_2.m_2$. We may abbreviate a frame (n, μ, false) to (n, μ) and (n, μ, true) to $(\not{n}, \mu)$. The concatenation of two sequences ξ and ν is denoted by $\xi : \nu$. The empty sequence is denoted by ϵ. The leftmost symbol in a sequence of frames corresponds to the topmost symbol of the stack.

The system is represented by a transition system *Sys* defined as follows. A state of *Sys* is a pair (σ, ξ) where σ is a global state of O and ξ is the contents of the control stack. We define the transition relation $\to$ of *Sys* by inference rules in Figure 3.3 (also see Figure 3.4).

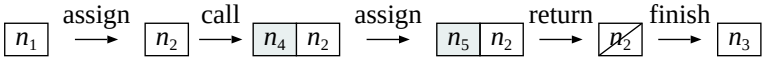

Figure 3.4. Behavior of the system with the program in Figure 3.2

Behavior of PCS with Policies

Policies Now we extend the system by introducing policies. Consider an ordered set $O = (o_1, o_2, \ldots, o_n)$ of objects. Each object in O has a finite set of policies as well as the attributes and the methods. Assuming that each object o_j has a set P_j of policies for $1 \leq j \leq n$, let $Policy(O) = \bigcup_{1 \leq j \leq n} P'_j$ where P'_j equals P_j except that "this" in P_j is replaced with o_j.

Let σ be a global state of the set O of objects, s and t be objects in O, m be a method of t, and mod be any of auth+, auth− and refrain. We define a relation $\sigma \models mod(s, t, m)$ as follows, which represents that the target t permits (if mod = auth+) or forbids (mod = auth−) the subject s to call the method m of t, or s refrains (if mod = refrain) from calling m of t when the global state is σ.

$$\begin{aligned}
&\sigma \models mod(s, t, m) \\
&\quad \text{if and only if } \exists p \in Policy(O),\ \exists \theta\text{: a substitution for the variables in } p, \\
&\qquad\qquad p = \text{“policy } mod \ldots B.m \leftarrow A \text{ if } Cond\text{”}, \\
&\qquad\qquad \sigma \circ \theta(Cond) = true,\ \theta(A) = s,\ \theta(B) = t.
\end{aligned}$$

For a global state σ and an event ev, we also define the set $\mathrm{oblg}(\sigma, ev)$ of obligatory method calls which become effective when the global state is σ and the event ev has just occurred.

$$\begin{aligned}
\mathrm{oblg}(\sigma, ev) = \{\ & t.m(v_1, \ldots, v_{k_m}) \leftarrow s \mid \\
& \exists p \in \mathrm{Policy}(O),\ \exists \theta\text{: a substitution for the variables in } p, \\
& p = \text{“policy oblg } \ldots B.m(E_1, \ldots, E_{k_m}) \leftarrow A \text{ on } Ev \text{ if } Cond\text{”}, \\
& \sigma \circ \theta(Ev) = ev,\ \sigma \circ \theta(Cond) = true,\ \theta(A) = s,\ \theta(B) = t, \\
& \sigma \circ \theta(E_i) = v_i \text{ for } 1 \leq i \leq k_m\ \}
\end{aligned}$$

Conflict between Policies If there exist two modes mod_1 and mod_2 such that

$$\sigma \models mod_1(s, t, m) \ \text{ and } \ \sigma \models mod_2(s, t, m)$$

and $(mod_1, mod_2) = (\text{auth+}, \text{auth−})$, then we say that the global state σ causes a conflict between policies upon the operation (s, t, m). Note that a conflict does not

Table 3.2. The truth of $\mathrm{CAN}(\sigma, t.m \leftarrow s)$

		refrain	
auth+	auth−	not marked	marked
not marked	not marked	depending on implementation	*false*
not marked	marked	*false*	*false*
marked	not marked	*true*	*false*
marked	marked	don't care	don't care

occur if $(mod_1, mod_2) = (\text{auth+}, \text{refrain})$, since the subject s refrains from performing (s, t, m) no matter whether t permits it or not.

There are two approaches to resolving a conflict: one is to obey a fixed rule, and the other is to specify a way to resolve using meta policies, which are policies to control other policies. Examples of the former approach are:

- Always give priority to auth− for safety.
- Using the class hierarchy among the objects, give priority to a policy defined in the most specialized class.

Using an arbitrary conflict resolution method (cf. [JSS97]), we define $\mathrm{CAN}(\sigma, t.m \leftarrow s)$ as a predicate which is true if the operation $t.m \leftarrow s$ is permitted when the global state is σ.

At a global state σ, we can compute $\mathrm{CAN}(\sigma, t.m \leftarrow s)$ as follows.

(1) For each $mod \in \{\text{auth+}, \text{auth−}, \text{refrain}\}$, test whether there exists a policy $p \in Policy(O)$ which satisfy the following.

- Assume that $p =$ "policy mod ... $B.m \leftarrow A$ if $Cond$". There exists a substitution θ such that $\theta(A) = s$, $\theta(B) = t$, and $\sigma \circ \theta(Cond) = true$.

If there exists such a policy p, then mark mod to represent that $\sigma \models mod(s, t, m)$ holds.

(2) If there exists a conflict, that is, both auth+ and auth− are marked, then resolve the conflict and erase the mark of either of them.

(3) The truth of $\mathrm{CAN}(\sigma, t.m \leftarrow s)$ is given by Table 3.2.

Order of Obligations If the set $\mathrm{oblg}(\sigma, ev)$ of obligations has more than one elements, then we assume a total order $\prec$ over $\mathrm{oblg}(\sigma, ev)$ which represents the order of performing the obligations. Thus

$$\mathrm{oblg}(\sigma, ev) = \{op_1, \ldots, op_l\} \quad \text{and} \quad op_1 \prec op_2 \prec \cdots \prec op_l$$

where $op_i =$ "$t_i.m_i(v_{i1}, \ldots, v_{ik_i}) \leftarrow s_i$" for $1 \leq i \leq l$. $op_i \prec op_j$ represents that op_i should be performed before op_j. For the above-mentioned $\mathrm{oblg}(\sigma, ev)$, we define a sequence $\mathrm{Foblg}(\sigma, ev)$ of stack frames (defined later) as follows.

$$\mathrm{Foblg}(\sigma, ev) = (n_{\mathrm{oblg}}[op_1], \bot) : (n_{\mathrm{oblg}}[op_2], \bot) : \cdots : (n_{\mathrm{oblg}}[op_l], \bot),$$

where $n_{\mathrm{oblg}}[op]$ is a special node newly introduced here for any obligatory operation op. Note that $n_{\mathrm{oblg}}[op]$ does not belong to any method and $\mathrm{IS}(n_{\mathrm{oblg}}[op])$ is defined as $\mathrm{IS}(n_{\mathrm{oblg}}[op]) = op$. Let $\mathrm{NO}_{\mathrm{oblg}}$ be the set of all $n_{\mathrm{oblg}}[op]$'s.

Transition System with Policies The transition system *Sys* defined by Figure 3.3 is modified substituting the rules (CALL′), (RETURN′), (FINISH′) and (OFINISH) in Figure 3.5 for the rules (CALL), (RETURN) and (FINISH) in Figure 3.3. After the modification, a method invocation $o_2.m_2 \leftarrow o_1$ is performed only when $\mathrm{CAN}(\sigma, o_2.m_2 \leftarrow o_1) = true$. Moreover, when a method has been just invoked or has just finished, a sequence of stack frames which will accomplish the obligations caused by the beginning or the end of the method is pushed into the control stack (see Figures 3.6).

Exception Handling

We extend our model by a function to handle exceptions, that is,

(1) Extend the program model so that we can specify an action to be performed when an exception has just occurred.

(2) Extend the transition system so that an exception occurs when a forbidden method call is requested.

Consider a situation in which an obligation causes a policy violation. In the definition of the policy specification language, we said that the main clause of an obligation policy is a method call, and thus we cannot specify any action for the violation exception. However, we extend the specification language as follows without changing the model of obligations. Figure 3.7 shows a specification of an obligation policy written

$$\text{(CALL}')\quad \frac{\begin{array}{l}\mathrm{IS}(n) = \text{“}o_2.m_2(e_1,\ldots,e_k) \leftarrow o_1\text{”} \\ \mathrm{CAN}(\sigma, o_2.m_2 \leftarrow o_1) = \mathit{true} \\ f_2 = (\mathrm{IT}[o_2.m_2], \bot[\mathit{arg}_1 := \sigma \circ \mu(e_1)] \cdots [\mathit{arg}_k := \sigma \circ \mu(e_k)]) \\ \mathrm{Foblg}(\sigma, \text{beginning of } o_2.m_2 \leftarrow o_1) = f_{\mathrm{x},1} : f_{\mathrm{x},2} : \cdots : f_{\mathrm{x},l}\end{array}}{(\sigma,\ \ (n,\mu) : \xi) \rightarrow (\sigma,\ \ f_{\mathrm{x},1} : f_{\mathrm{x},2} : \cdots : f_{\mathrm{x},l} : f_2 : (n,\mu) : \xi)}$$

$$\text{(RETURN}')\quad \frac{\begin{array}{l}\mathrm{IS}(m) = \text{“return } e\text{”} \qquad \mathrm{IS}(n_1) = \text{“}o_2.m_2(e_1,\ldots,e_k) \leftarrow o_1\text{”} \\ f_2 = (n_1, \mu[\mathit{ret} := \sigma \circ \mu_2(e)], \mathit{true}) \\ \mathrm{Foblg}(\sigma, \text{end of } o_2.m_2 \leftarrow o_1) = f_{\mathrm{x},1} : f_{\mathrm{x},2} : \cdots : f_{\mathrm{x},l}\end{array}}{(\sigma,\ \ (m,\mu_2) : (n_1,\mu) : \xi) \rightarrow (\sigma,\ \ f_{\mathrm{x},1} : f_{\mathrm{x},2} : \cdots : f_{\mathrm{x},l} : f_2 : \xi)}$$

$$\text{(FINISH}')\quad \frac{n_1 \notin \mathrm{NO}_{\mathrm{oblg}} \qquad n_1 \xrightarrow{e} n_2 \qquad \sigma \circ \mu(e) = \mathit{true}}{(\sigma,\ \ (\not{n}_1,\mu) : \xi) \rightarrow (\sigma,\ \ (n_2,\mu) : \xi)}$$

$$\text{(OFINISH)}\quad \frac{n_1 \in \mathrm{NO}_{\mathrm{oblg}}}{(\sigma,\ \ (\not{n}_1,\mu) : \xi) \rightarrow (\sigma,\ \ \xi)}$$

Figure 3.5. Inference rules which define the transition relation (2)

in the extended language. In the language we can write an arbitrary program code in the main clause (part (a) of Figure 3.7). We assume that the part (a) is the body of a method which has no name and when this obligation becomes effective, the method is called.

Program with Exception Handling A program is a tuple (NO, TG, EG, IS, IT, VAR), where EG is a set of edges called exception edges. An element in EG is a tuple (n_1, r, ty, n_2) where $n_1, n_2 \in$ NO, $r \in$ VAR, and ty a type of an exception such as OperationFailed (see Figure 3.8). Note that in our model an exception occurs only at a method call and thus n_1 should be a method call. $n_1 \xrightarrow{r:ty}_{\mathrm{EG}} n_2$ denotes $(n_1, r, ty, n_2) \in$ EG and represents that if the control is at n_1 and an exception ex of the type ty occurs, it can move to n_2 assigning ex to r (i.e., the exception is caught). When an exception ex of a type ty occurs at a node n and $n \xrightarrow{r:ty}_{\mathrm{EG}} n_2$ does not hold for any n_2 and r, ex is delivered to the method which called the method which n belongs to (i.e., the exception is thrown).

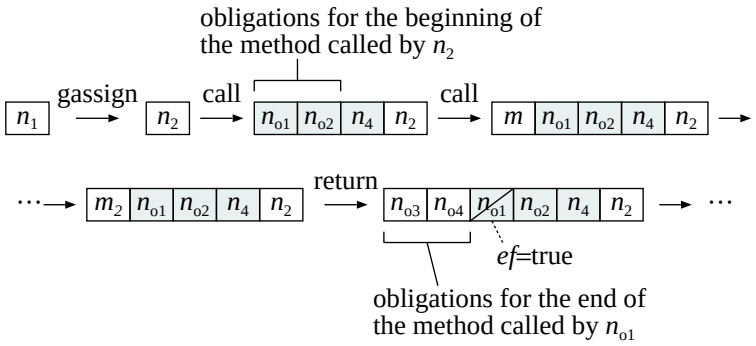

Figure 3.6. Behavior of the system with obligations

```
policy oblg DELIVERY_FEE
  var sender:Channel
(a)  try
       (this.casher).pay((this.cert).fee, sender)
     except
       on e:OperationFailed do this.throw(e)
  on beginning of this.receive←sender
```

Figure 3.7. A sample obligation

Transition System with Exception Handling Let $\mathrm{ex}_{\mathrm{policy}}$ be a constant which represents the policy violation exception, and typeof(ex) be the type of an exception ex.

Let $f_1 = (n_1, \mu_1, ef_1)$ and $f_2 = (n_2, \mu_2, ef_2)$ be arbitrary frames. We define a mapping CASTOFF from and to a sequence of frames as

$$\mathrm{CASTOFF}(f_1) = \epsilon$$

$$\mathrm{CASTOFF}(f_2 : f_1 : \xi) = \begin{cases} f_1 : \xi & \text{if } ef_1 = \mathit{true} \text{ or } n_2 \notin \mathrm{NO}_{\mathrm{oblg}} \\ \mathrm{CASTOFF}(f_1 : \xi) & \text{otherwise.} \end{cases}$$

We extend the transition system Sys as follows. A stack frame can be either the above-mentioned tuple (n, μ, ef) or an exception ex. If the topmost element of the control stack is an exception ex, it represents that ex has occurred and is to be processed. We add the inference rules in Figure 3.9 to the set of the rules in Figure 3.5. We also add "$m_2 \neq$ throw" to the premise of the rule (CALL′) in Figure 3.5.

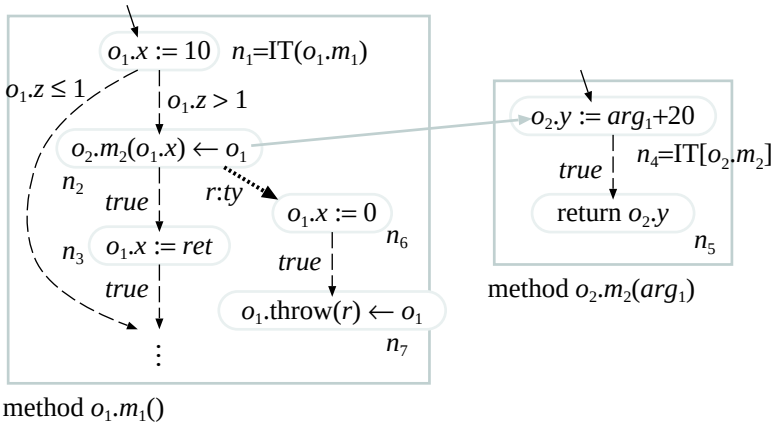

Figure 3.8. A sample program with an exception edge

3.3. Pushdown System

To verify a policy controlled system introduced in section 3.2, we use *pushdown system* (PDS) and its model checking method. In this section, we first review the definition of PDS and the model checking method for PDS in [EHRS00]. We then propose a new algorithm for computing the reachable configurations of a PDS. This algorithm works faster than the algorithm in [EHRS00] and matches the algorithms in [AEY01, BGR01] when we use it for the verification of a policy controlled system.

3.3.1 Definitions

A pushdown system is a tuple $M = (P, \Gamma, \Delta, q_0, \omega)$ where P is the finite set of *control locations*, Γ is the finite *stack alphabet*, $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is the finite set of *transition rules*, $q_0 \in P$ is the *initial control location*, and $\omega \in \Gamma$ is the bottom stack symbol. For simplicity, we can write $\langle p, a\rangle \hookrightarrow \langle q, w\rangle$ instead of $((p, a), (q, w)) \in \Delta$. Without loss of generality, we assume that for any $p, q \in P$, if $\langle p, \omega\rangle \hookrightarrow \langle q, w\rangle$, then w is of the form $\alpha\omega$. A *configuration* of M is a pair $\langle q, w\rangle$ where $q \in P$ and $w \in \Gamma^*$. The *initial configuration* is $\langle q_0, \omega\rangle$. The empty sequence of stack symbols is denoted by ϵ. The *transition relation* of M is the least relation $\Rightarrow$ satisfying the following condition: $\langle p, aw\rangle \Rightarrow \langle q, vw\rangle$ for every $w \in \Gamma^*$, if $\langle p, a\rangle \hookrightarrow \langle q, v\rangle$.

The reflexive and transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. For a given set $C \subseteq P \times \Gamma^*$ of configurations, the set of successors of C, which is $\{c' \in P \times \Gamma^* \mid \exists c \in C.\ c \Rightarrow c'\}$, is denoted by $post[M](C)$. The reflexive and transitive closure of $post[M](C)$, which is

$$\text{(THROW1)}\quad \frac{\mathrm{IS}(n) = \text{“}o_1.\mathrm{throw}(e_1) \leftarrow o_1\text{”} \qquad ex_1 = \sigma \circ \mu(e_1)}{(\sigma,\ \ (n,\mu) : \xi) \rightarrow (\sigma,\ \ ex_1 : (n,\mu) : \xi)}$$

$$\text{(POLICY\_EX)}\quad \frac{\begin{array}{l}\mathrm{IS}(n) = \text{“}o_2.m_2(e_1,\ldots,e_k) \leftarrow o_1\text{”} \\ \mathrm{CAN}(\sigma, o_2.m_2 \leftarrow o_1) = \mathit{false}\end{array}}{(\sigma,\ (n,\mu) : \xi) \rightarrow (\sigma,\ \mathrm{ex}_{\mathrm{policy}} : (n,\mu) : \xi)}$$

$$\text{(CATCH)}\quad \frac{n \overset{r:\mathrm{typeof}(ex)}{\rightarrow_{\mathrm{EG}}} n_2}{(\sigma,\ \ ex : (n,\mu,ef) : \xi) \rightarrow (\sigma,\ \ (n_2, \mu[r := ex]) : \xi)}$$

$$\text{(THROW2)}\quad \frac{n \overset{r:\mathrm{typeof}(ex)}{\rightarrow_{\mathrm{EG}}} n_2 \text{ does not hold for any } n_2 \text{ and } r.}{(\sigma,\ \ ex : (n,\mu,ef) : \xi) \rightarrow (\sigma,\ \ ex : \mathrm{CASTOFF}((n,\mu,ef) : \xi))}$$

Figure 3.9. Inference rules for exception handling

$\{c' \in P \times \Gamma^* \mid \exists c \in C.\ c \Rightarrow^* c'\}$, is denoted by $post^*[M](C)$. We say that a configuration c is *reachable* if $\langle q_0, \omega\rangle \Rightarrow^* c$ (or equivalently $c \in post^*[M](\{\langle q_0, \omega\rangle\})$).

We say a PDS M is in *normal form* if M satisfies $|w| \leq 2$ for every transition rule $\langle p, a\rangle \hookrightarrow \langle p', w\rangle$, where $|w|$ is the length of w. Any PDS can easily be converted into a normal form PDS by adding new control locations, the number of which is not more than the size of Δ.

3.3.2 Model Checking Pushdown Systems

As we are interested in security properties of policy control, we concentrate on the safety verification problem (the verification problem for short), which is one of the most important model checking problems. The (safety) verification problem for PDS is defined as follows:

- **Inputs**: A PDS M and a verification property $\Psi \subseteq P \times \Gamma^*$.
- **Output**: Does every reachable configuration belong to Ψ? (or equivalently, $post^*[M](\{\langle q_0, \omega\rangle\}) \subseteq \Psi$?)

In [EKS01], Esparza et al. show that if a set C of configurations is regular, then $post^*[M](C)$ is also regular and they present an algorithm for calculating $post^*[M](C)$. Using their results, we can solve the verification problem. To represent a regular set of configurations, we define $\mathcal{P}$-*automata* which accept configurations of M.

Definition 3.3.1 ($\mathcal{P}$-automata). Let $M = (P, \Gamma, \Delta, q_0, \omega)$ be a pushdown system. A *$\mathcal{P}$-automaton* is a tuple $\mathcal{A} = (Q, \Gamma, \delta, P, F)$ where $Q \supseteq P$ is the finite set of states, Γ is the tape alphabet (which equals the stack alphabet of M), $\delta : Q \times \Gamma \to 2^Q$ is the transition function, P is the set of initial states (which equals the set of control locations of M) and $F \subseteq Q$ is the set of final states. We extend δ to $\hat{\delta} : Q \times \Gamma^* \to 2^Q$ in the usual way. A configuration $\langle q, w \rangle$ of M is *accepted* by $\mathcal{A}$ if and only if $F \cap \hat{\delta}(q, w) \neq \emptyset$. The set of configurations accepted by $\mathcal{A}$ is denoted by $Conf(\mathcal{A})$. We say a set C of configurations is *regular* if there exists a $\mathcal{P}$-automaton $\mathcal{A}$ such that $C = Conf(\mathcal{A})$. □

For a set A, let $|A|$ denote the cardinality of A.

Theorem 3.3.2. [EKS01] For any pushdown system $M = (P, \Gamma, \Delta, q_0, \omega)$ and any $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \delta, P, F)$, there effectively exists a $\mathcal{P}$-automaton $\mathcal{A}_{post^*}$ such that $Conf(\mathcal{A}_{post^*}) = post^*[M](Conf(\mathcal{A}))$. For a normal form PDS M, $\mathcal{A}_{post^*}$ can be constructed in $O(|P| \cdot |\Delta| \cdot (|Q| + |\Delta|) + |P| \cdot |\delta|)$ time and space. □

If a verification property Ψ is given by a $\mathcal{P}$-automaton, then we can solve the verification problem for Ψ using the algorithm mentioned in Theorem 3.3.2. We describe the decision algorithm in the next theorem for referential convenience to the following sections.

Theorem 3.3.3. Let $M = (P, \Gamma, \Delta, q_0, \omega)$ be a PDS and $\mathcal{A} = (Q, \Gamma, \delta, P, F)$ be a $\mathcal{P}$-automaton. The verification problem for M and verification property $\Psi = Conf(\mathcal{A})$ is solvable.

Proof. Recall that the verification problem is the problem which decides whether $post^*[M](\{\langle q_0, \omega \rangle\}) \subseteq \Psi$. By Theorem 3.3.2, we can also construct a $\mathcal{P}$-automaton $\mathcal{A}_{post^*}$ which accepts $post^*[M](\{\langle q_0, \omega \rangle\})$. Since the inclusion problem of regular languages is decidable, we can decide whether $post^*[M](\{\langle q_0, \omega \rangle\}) \subseteq \Psi$. □

3.3.3 Computing Reachable Configurations

The proof of theorem 3.3.3 shown above is based on Esparza's $\mathcal{A}_{post^*}$ algorithm. The algorithm can construct a $\mathcal{P}$-automaton $\mathcal{A}_{post^*}$ such that $Conf(\mathcal{A}_{post^*}) = post^*[M](C)$ for any regular set C of configurations. To solve the verification problem, however, we need only $\mathcal{A}_{post^*}$ for $C = \{\langle q_0, \omega \rangle\}$, which can be constructed more efficiently than by using Esparza's algorithm. In the following, we show an algorithm for constructing $\mathcal{A}_{post^*}$ for $C = \{\langle q_0, \omega \rangle\}$, which is an extension of the algorithm in [NTS01b]. Instead

$$\frac{\langle p, a\rangle \hookrightarrow \langle q, \epsilon\rangle}{(p, a, q) \in R_M} \qquad \frac{\begin{array}{c}\langle p, a\rangle \hookrightarrow \langle q_1, b_1 b_2 \ldots b_k\rangle \\ (q_1, b_1, q_2) \in R_M \\ \cdots \\ (q_{k-1}, b_{k-1}, q_k) \in R_M \\ (q_k, b_k, q) \in R_M\end{array}}{(p, a, q) \in R_M}$$

Figure 3.10. Inference rules for R_M

$$\frac{}{X_{p,a} \to pa \in D} \qquad \frac{\langle p, a\rangle \hookrightarrow \langle q, \epsilon\rangle}{X_{p,a} \to q \in D} \qquad \frac{\begin{array}{c}\langle p, a\rangle \hookrightarrow \langle q_1, b_1 b_2 \ldots b_k\rangle \\ (q_1, b_1, q_2) \in R_M \\ (q_2, b_2, q_3) \in R_M \\ \cdots \\ (q_{j-1}, b_{j-1}, q_j) \in R_M\end{array}}{X_{p,a} \to X_{q_j, b_j} b_{j+1} b_{j+2} \ldots b_k \in D}$$

Figure 3.11. Inference rules for G_M

of constructing a $\mathcal{P}$-automaton, we construct a left linear grammar G_M from a given PDS $M = (P, \Gamma, \Delta, q_0, \omega)$, which satisfies $w \in L(G_M)$ if and only if $w = q\gamma$ and $\langle q_0, \omega\rangle \Rightarrow^* \langle q, \gamma\rangle$. $\mathcal{A}_{post^*}$ can easily be obtained from G_M.

First, we define a relation $R_M \subseteq P \times \Gamma \times P$ as the least relation which satisfies the rules in Figure 3.10. The following lemma shows the meaning of R_M.

Lemma 3.3.4. $(p, a, q) \in R_M$ if and only if $\langle p, a\rangle \Rightarrow^* \langle q, \epsilon\rangle$.

Proof Sketch. (Only-if) By induction on the number of rules used for deriving $(p, a, q) \in R_M$.
(If) By induction on the number of transitions between $\langle p, a\rangle$ and $\langle q, \epsilon\rangle$. □

Using R_M, we define the right linear grammar $G_M = (V, T, D, I)$ for the given PDS $M = (P, \Gamma, \Delta, q_0, \omega)$ where $V = P \times \Gamma$ is the set of variables (for readability, we write $X_{p,a}$ to represent $(p, a) \in V$), $T = P \cup \Gamma$ is the set of terminal symbols, D is the set of productions and is defined as the smallest set which satisfies the rules in Figure 3.11, and $I = X_{q_0,\omega}$ is the start symbol.

Lemma 3.3.5. *$w \in L(G_M)$ if and only if $w = q\gamma$ and $\langle q_0, \omega\rangle \Rightarrow^* \langle q, \gamma\rangle$.*

Proof Sketch. This lemma is implied by the following stronger proposition:

w can be derived from $X_{p,a}$ if and only if $w = q\gamma$ and $\langle p, a\rangle \Rightarrow^* \langle q, \gamma\rangle$.

(Only-if) By induction on the length of the derivation of w.
(If) By induction on the number of transitions between $\langle p, a\rangle$ and $\langle q, \gamma\rangle$. □

Lemma 3.3.6. *For a normal form PDS $M = (P, \Gamma, \Delta, q_0, \omega)$, G_M can be constructed in $O(|P_\epsilon|^2 \cdot |\Delta|)$ where $P_\epsilon = \{q \in P \mid \langle p, a\rangle \hookrightarrow \langle q, \epsilon\rangle$ for some p and $a\}$.* □

Obviously, $|P_\epsilon| \leq \min(|P|, |\Delta|)$. Note that a PDS which is not in normal form can be converted into a normal form PDS without increasing $|P_\epsilon|$.

In the verification of a policy controlled system (PCS) described in section 3.4, we construct $\mathcal{A}_{post^*}$ for a PDS $S^\sharp = (P, \Gamma, \Delta, p, n_0)$ with $|P| = 2$. When we use either Esparza's algorithm or ours to construct $\mathcal{A}_{post^*}$, $S^\sharp$ has to be converted into a normal form PDS $M' = (P', \Gamma, \Delta', p, n_0)$ with $|P'| = O(|P| + \|\Delta\|)$ and $|\Delta'| = O(\|\Delta\|)$, where $\|\Delta\|$ is the size of Δ. Therefore, Esparza's algorithm takes $O(\|\Delta\|^3)$ time to construct $\mathcal{A}_{post^*}$ while ours takes only $O(\|\Delta\|)$ time.

3.4. Model Checking Policy Controlled Systems

In this section, we present a method for verifying a safety property of a policy controlled system (PCS). When every reachable configuration of a PCS (or PDS) S belongs to a property Ψ (see subsection 3.3.2), we simply say that S satisfies Ψ. Similar to the case of pushdown system (PDS), the (safety) verification problem for PCS is defined as the problem to decide, for a PCS S and a verification property Ψ, whether S satisfies Ψ. We first present an abstraction of a PDS $S^\sharp$ from a given PCS S. We can verify whether $S^\sharp$ satisfies a given property Ψ by using the method described in section 3. By the soundness of the abstraction (Theorem 3.4.2), if $S^\sharp$ is known to satisfy Ψ then S is also guaranteed to satisfy Ψ.

3.4.1 Abstracting PDS from PCS

We define a transformation which abstracts PDS from PCS. Remember that a configuration of a PCS is a pair of global variable values and a stack (of which frames may involve local variable values), while a configuration of a PDS is a pair of a control location and a stack. Hence, program variables in the PCS must be abstracted. For a given PCS $S = (O, Prog, Policy)$, the abstract PDS $S^\sharp = (P, \Gamma, \Delta, p, n_0)$ is defined as $P = \{p, q\}$ where p and q are new symbols, $\Gamma = \{n, \not\Rightarrow \mid n$ is a node in $Prog\}$, the

$$\text{(GASSIGN)}\quad \frac{\mathrm{IS}(n) = \text{“}o.a := e\text{”} \qquad n \xrightarrow{e_3} n_2}{\langle p, n\rangle \hookrightarrow \langle p, n_2\rangle}$$

$$\text{(LASSIGN)}\quad \frac{\mathrm{IS}(n) = \text{“}r := e\text{”} \qquad n \xrightarrow{e_3} n_2}{\langle p, n\rangle \hookrightarrow \langle p, n_2\rangle}$$

$$\text{(CALL)}\quad \frac{\begin{array}{l}\mathrm{IS}(n) = \text{“}o_2.m_2(e_1, \ldots, e_k) \leftarrow o_1\text{”} \\ \mathrm{CAN}^\sharp(o_2.m_2 \leftarrow o_1) = \mathit{true} \qquad f_2 = \mathrm{IT}[o_2.m_2] \\ \mathrm{Foblg}^\sharp(\text{beginning of } o_2.m_2 \leftarrow o_1) \ni f_{\mathrm{x},1} : f_{\mathrm{x},2} : \cdots : f_{\mathrm{x},l}\end{array}}{\langle p, n\rangle \hookrightarrow \langle p, f_{\mathrm{x},1} f_{\mathrm{x},2} \cdots f_{\mathrm{x},l}\, f_2\, n\rangle}$$

$$\text{(RETURN)}\quad \frac{\begin{array}{l}\mathrm{IS}(m) = \text{“return } e\text{”} \qquad \mathrm{IS}(n_1) = \text{“}o_2.m_2(e_1, \ldots, e_k) \leftarrow o_1\text{”} \\ \mathrm{Foblg}^\sharp(\text{end of } o_2.m_2 \leftarrow o_1) \ni f_{\mathrm{x},1} : f_{\mathrm{x},2} : \cdots : f_{\mathrm{x},l}\end{array}}{\begin{array}{l}\langle p, m\rangle \hookrightarrow \langle q, \epsilon\rangle \\ \langle q, n_1\rangle \hookrightarrow \langle p, f_{\mathrm{x},1} f_{\mathrm{x},2} \cdots f_{\mathrm{x},l}\, \not{n}_1\rangle\end{array}}$$

$$\text{(FINISH)}\quad \frac{n_1 \notin \mathrm{NO}_{\mathrm{oblg}} \qquad n_1 \xrightarrow{e} n_2}{\langle p, \not{n}_1\rangle \hookrightarrow \langle p, n_2\rangle}$$

$$\text{(OFINISH)}\quad \frac{n_1 \in \mathrm{NO}_{\mathrm{oblg}}}{\langle p, \not{n}_1\rangle \hookrightarrow \langle p, \epsilon\rangle}$$

Figure 3.12. Abstraction rules

initial control location is p, the bottom stack symbol n_0 is a node in $Prog$, which represents the initial program point of $Prog$ and Δ is defined in Figure 3.12. Each rule in Figure 3.12 has its counterpart in Figure 3.5. In fact, rules in Figure 3.12 are obtained from Figure 3.5 by discarding values of global and local variables. Note that rule (RETURN) in Figure 3.5 replaces the topmost two frames. To simulate this rule, we use the location q and define two rules which are consecutively applied due to q.

For a sequence of frames $\xi = (n_1, \mu_1) : (n_2, \mu_2) : \ldots : (n_k, \mu_k)$ of S, define $\xi^\sharp = n_1 n_2 \ldots n_k$. That is, $\xi^\sharp$ is obtained from ξ by discarding values of all local variables. $\mathrm{CAN}^\sharp$ and $\mathrm{Foblg}^\sharp$ in Figure 3.12 are the abstract versions of CAN and Foblg, respectively. $\mathrm{CAN}^\sharp$ is any predicate such that $\mathrm{CAN}^\sharp(o_2.m \leftarrow o_1) = \mathit{true}$ whenever $\mathrm{CAN}(\sigma, o_2.m \leftarrow o_1) = \mathit{true}$ for some σ. Specifically, we define $\mathrm{CAN}^\sharp$ as follows: delete all conditional negative authorization policies (i.e., which have the if-clauses), and replace the if-clauses of all positive authorization policies with *true*. Next, define $\mathrm{CAN}^\sharp$ in the same way as CAN. Likewise, $\mathrm{Foblg}^\sharp$ is any set of sequences of frames which

satisfies $\mathrm{Foblg}^\sharp(ev) \ni (\mathrm{Foblg}(\sigma, ev))^\sharp$ for every σ.

For a configuration $c = (\sigma, \xi)$, let us define $c^\sharp = \langle p, \xi^\sharp \rangle$. Also, for a given property Ψ, which is a subset of configurations of S, let us define the abstract property $\Psi^\sharp$ as $\Psi^\sharp = \{c' \mid \text{every } c \text{ such that } c^\sharp = c' \text{ satisfies } c \in \Psi\}$. We can obtain the following soundness property of the abstraction.

Lemma 3.4.1. Let S be a PCS and $S^\sharp$ be the corresponding PDS abstracted from S. For any configurations c_1 and c_2 in S, if $c_1 \to c_2$ then $c_1^\sharp \Rightarrow^* c_2^\sharp$.

Proof. By the correspondence of rules in Figure 3.5 and rules in Figure 3.12. □

Theorem 3.4.2 (Soundness of the abstraction). Let S be a PCS and $S^\sharp$ be the corresponding abstract PDS. Also let Ψ be a property in S and $\Psi^\sharp$ be the corresponding abstract property in $S^\sharp$. If $S^\sharp$ satisfies $\Psi^\sharp$, then S also satisfies Ψ.

Proof. Assume that every reachable configuration of $S^\sharp$ belongs to $\Psi^\sharp$. Let c be an arbitrary reachable configuration of S. By Lemma 3.4.1, $c^\sharp$ is reachable in $S^\sharp$. By assumption, $c^\sharp \in \Psi^\sharp$. By the definition of $\Psi^\sharp$, we know $c \in \Psi$. □

By Theorem 3.4.2, if we can find a $\mathcal{P}$-automaton $\mathcal{A}_\Psi$ such that $Conf(\mathcal{A}_\Psi) \subseteq \Psi^\sharp$ and know that the answer to the verification problem for $S^\sharp$ and $Conf(\mathcal{A}_\Psi)$ is affirmative by the method in section 3, then we can conclude that the answer to the original problem for S and Ψ is also affirmative.

3.4.2 Verification Example

In this section, we briefly introduce our verification tool and show verification results on example policies. For simplicity, the current version of the verification tool assumes that for a PCS S and a verification property Ψ, authorization policies and program variables have been abstracted from S according to the rules in Figure 3.12 and a deterministic $\mathcal{P}$-automaton $\mathcal{A}_\Psi$ such that $Conf(\mathcal{A}_\Psi) = \Psi$ is given. That is, inputs to the verification tool are a PCS $S = (O, Prog, Policy)$ where $Prog$ is without variables and $Policy$ is a set of obligation policies, and a deterministic $\mathcal{P}$-automaton $\mathcal{A}_\Psi$. The verification tool performs the following procedure.

(Step1) From a given PCS S, construct a $\mathcal{P}$-automaton which accepts the set of all reachable configurations of $S^\sharp$ based on the algorithms in sections 3.3.3 and 3.4.1. That is, construct a $\mathcal{P}$-automaton $\mathcal{A}_{post^*}$ such that $Conf(\mathcal{A}_{post^*}) = post^*[S^\sharp](\{\langle p, n_0 \rangle\})$ where $\langle p, n_0 \rangle$ is the initial configuration of $S^\sharp$.

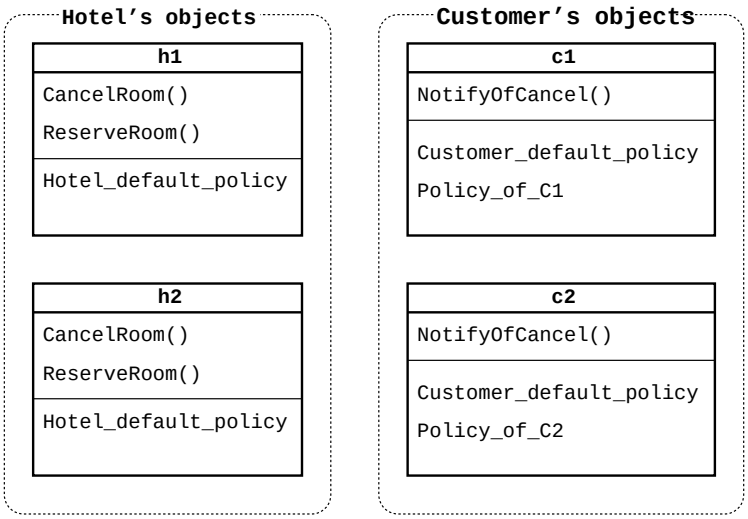

Figure 3.13. A hotel reservation system

(Step2) Decide whether $Conf(\mathcal{A}_{post^*}) \subseteq Conf(\mathcal{A}_\Psi)$.

The answer issued in (Step2) is "yes" if and only if $post^*[S^\sharp](\{\langle p, n_0\rangle\}) \subseteq Conf(\mathcal{A}_\Psi) = \Psi$, implying S satisfies Ψ. The verification tool is implemented by Java.

Example 3.4.3. Consider a simple online hotel reservation system which provides reservation management services for several hotels and their customers. In the system, every hotel and every customer have their own object, and each of them can specify its own policy in its object. Every hotel's object must prepare *ReserveRoom()* and *CancelRoom()* methods for their customers, and every customer's object is required to have *NotifyOfCancel()* method to receive a notice of cancellation of a room. Also, every hotel's object and every customer's object must have the following policies.

"If a reservation at a hotel is canceled by a customer, then the hotel must give notice of the cancellation to all other customers."

"Every customer must cancel his/her reservation of a hotel before he/she makes a new reservation of another hotel."

For simplicity, we assume that two hotels h_1 and h_2 are registered in the system, and their customers are c_1 and c_2 only (Figure 3.13). They must have the following policies.

Table 3.3. Verification profiles of example 3.4.3

# of customers	PCS S		$\mathcal{A}_{post^*}$		computation time† (sec)
	# of nodes	# of edges	$\|P\|$	$\|\Delta\|$	
10	190	830	791	1309	7.5
40	700	9290	9101	11120	45.9
70	1210	26750	26411	29930	198.3
100	1720	53210	52721	57740	312.8

† JVM build J2SDK.v.1.4.1, on Windows XP (Pentium 4 (2GHz), 1GB RAM)

```
policy oblg Hotel_default_policy
  c2.NotifyOfCancel()←this on end of this.CancelRoom()←c1
  c1.NotifyOfCancel()←this on end of this.CancelRoom()←c2
policy oblg Customer_default_policy
  h1.CancelRoom()←this on beginning of h2.ReserveRoom()←this
  h2.CancelRoom()←this on beginning of h1.ReserveRoom()←this
```

Furthermore, assume that c_1 wants to reserve at hotel h_1 and c_2 wants to reserve at hotel h_2. Thus they independently specify the following policies.

```
policy oblg Policy_of_C1
  h1.ReserveRoom()←this on end of this.NotifyOfCancel()←h1
policy oblg Policy_of_C2
  h2.ReserveRoom()←this on end of this.NotifyOfCancel()←h2
```

Consider a situation that for some reason, c_1 has reserved at hotel h_2 and c_2 has reserved at hotel h_1. In this situation, if c_2 cancels his/her reservation at h_1, then the system immediately enters an infinite chain of obligation method calls.

This undesirable behavior could be detected by our verification tool as follows. First, we specified the verification property as "the control stack is always shorter than a certain length." We will call this length the *threshold length*. Actually, we verified this property for a few different threshold lengths, and for any lengths, our verification tool answered "no" (the hotel reservation system did not satisfy the property) and showed the following error trace:

```
h1.CancelRoom()←c2, c1.NotifyOfCancel()←h1, h1.ReserveRoom()←c1, h2.CancelRoom()←c1,
c2.NotifyOfCancel()←h2, h2.ReserveRoom()←c2, h1.CancelRoom()←c2, ....
```

To measure the computation time to verify a larger PCS, we extended the hotel reservation system to have five hotels and an arbitrary number of customers. The profiles of the verification are summarized in Table 3.3.

In this example, the computation time needed for the verification hardly increased

as the threshold length grew. That is, the computation time is affected mainly by the size of the input PCS. Since the size of the PCS in this example is determined by the number n_c of customers (namely, $O(n_c)$ of nodes and $O(n_c^2)$ of edges), we fixed the threshold length at 1000 and measured the computation time for different numbers of customers. From Table 3.3, we can see that the computation time is within five and a half minutes when the number of customers is not more than 100. Note that in this example, $\mathcal{A}_{post^*}$ contains many unreachable transitions since most nodes of the PCS are unreachable because of the infinite chain of obligations. For example, there are only 456 transitions reachable from the initial state when $n_c = 100$. If we avoid generating unreachable transitions when constructing $\mathcal{A}_{post^*}$, the construction time is reduced and the total computation time becomes about the half of that in Table 3.3. □

3.5. Conclusion of Chapter 3

In this chapter, an automatic verification method for a policy controlled system (PCS) using PDS model checking has been proposed. Examples conducted on the verification tool have also been demonstrated.

Verification of a security goal other than a safety property, e.g., a liveness property is left as a future study. The proposed abstraction of PDS from PCS is such that the data part of the PCS is discarded and a conditional, which depends on the data part, is replaced with a nondeterministic choice. However, in some cases, more sophisticated abstraction is required to succeed in model checking [CGP00]. Such abstraction techniques include abstract interpretation, program slicing [CDHLPRZ00] and predicate abstraction [GS97]. We would like to integrate these techniques into our verification method.

Chapter 4

Conclusion

In this thesis, for MAC based system and DAC based system, we proposed static analysis methods which check whether the whole system behavior determined by security policies satisfies security goals of the system.

In Chapter 2, an information flow analysis algorithm for a procedural program is proposed. In our model, SC of data can be formalized as an arbitrary finite lattice. The proposed algorithm adopt abstract interpretation as an analysis method which makes information flow analysis of an arbitrary recursive program possible. That is, the algorithm constructs equations from statements in the program and computes the least fixpoint of those equations. We proved the soundness of the algorithm and showed the algorithm can be executed in $O(N^3)$ time where N is the total size of given program. Furthermore, the algorithm has been extended so that operations which hide information of their arguments can be appropriately modeled by using congruence relation.

Analysis result of the proposed algorithm can be used for providing flexibility to a MAC database system as shown in Figure 2.1.

In [OYF03], the information flow analysis algorithm proposed in Chapter 2 is extended so that not only one-way functions but also trapdoor functions can be properly modeled.

In Chapter 3, first, a simple but useful policy description language which can specify not only permission and prohibition policies but also obligation policies is proposed. Secondly, using the proposed language, we have defined a policy controlled system (PCS). PCS is a system in which each object has its own security policy (specified by the proposed language) and objects' behaviors are autonomously controlled based on those policies when they interact with one anther. Operational semantics of PCS is formally defined.

We defined the (safety) verification problem for PCS as the problem to decide for a given PCS S and a goal (called *safety property*) Ψ, whether every reachable state of P satisfies Ψ where Ψ is represented by a regular language. Using a model checking technique for pushdown system (PDS), we proposed a method for solving the (safety) verification problem for PCS.

An obligation policy is similar to an aspect in aspect-oriented programming (AOP) [K03] and an active rule (or an event-condition-action rule, ECA rule) in active database [PD99]. An aspect in AOP specifies actions which must be performed when the program control reaches a certain point called join point (e.g., "Call method A when method B is called" where an invocation of method B is the join point of this aspect.) An active rule in active database specifies actions which must be performed when a certain action is performed to the database (e.g., "Take a log when data is written to the database.") How to apply the verification method for PCS to the verification of AOP and active database system is a future work.

Note that, the security of all access control systems cannot be protected by using the security assurance methods shown in this paper. However, for MAC based system and DAC based system which are general access control systems, we showed that using abstract interpretation and model checking as the analysis method is useful.

There is an access control model called Role-Based Access Control (RBAC) [PHS03] as well as MAC and DAC. The RBAC model streamlines the access control by first defining *roles* which are given access permissions to objects and later assigning roles to subjects (users). That is, the access permissions do not depend on who a subject is but rather which role a subject belongs to. The RBAC model allows us to define relations among roles and users. When considering a security assurance method for RBAC, the relation between a user and a role will be important. Therefore, if we are going to use model checking as analysis method, the necessity of modeling the relation between a user and a role will may come out, which is also an interesting future work.

References

[AFG99] M. Abadi, C. Fournet and G. Gonthier: Secure communications processing for distributed languages, 1999 IEEE Symp. on Security and Privacy, 74–88.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman: *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[AEY01] R. Alur, K. Etessami and M. Yannakakis: Analysis of recursive state machines, 13th Conference on Computer Aided Verification (CAV '01), LNCS 2102, 207–220.

[A98] A. W. Appel: *Modern Compiler Implementation in Java*, Cambridge University Press, 1998.

[BBM94] J. Banâtre, C. Bryce and D. Le Métayer: Compile-time detection of information flow in sequential programs, 3rd European Symp. on Research in Computer Security (ESORICS), LNCS 875, 55–73, 1994.

[BGR01] M. Benedikt, P. Godefroid and T. Reps: Model checking of unrestricted hierarchical state machines, 28th International Colloquium on Automata, Languages and Programming (ICALP '01), LNCS 2076, 652–666.

[B76] R. Bird: *Programs and Machines*, John Wiley & Sons, 1976.

[CGP00] E. M. Clarke, O. Grumberg and D. Peled: *Model Checking,* MIT Press, 2000.

[CDHLPRZ00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby and H. Zheng: Bandera: Extracting finite-state models from Java source code, IEEE Int'l Conf. on Software Engineering (ICSE), 439–448, 2000.

[Da02] N. C. Damianou: *A Policy Framework for Management of Distributed Systems.* PhD thesis, Imperial College of Science, Technology and Medicine, 2002. http://www-dse.doc.ic.ac.uk/Research/policies/ponder/thesis-ncd.pdf

[DDLS01] N. C. Damianou, N. Dulay E. Lupu and M. Sloman: The Ponder policy specification language, Workshop on Policies for Distributed Systems and Networks (POLICY '01), LNCS 1995, 18–38.

[De76] D. E. Denning: A lattice model of secure information flow, Communications of the ACM, 19(5), 236–243, 1976.

[DD77] D. E. Denning and P. J. Denning: Certification of programs for secure information flow, Communications of the ACM, 20(7), 504–513, 1977.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon: Efficient algorithms for model-checking pushdown systems, 12th Conference on Computer Aided Verification (CAV '00), LNCS 1855, 232–247, 2000.

[EKS01] J. Esparza, A. Kučera, and S. Schwoon: Model-checking LTL with regular variations for pushdown systems, 4th International Symposium on Theoretical Aspects of Computer Software (TACS '01), LNCS 2215, 316–339, 2001.

[ES01] J. Esparza and S. Schwoon: A BDD-based model checker for recursive programs, 13th Conference on Computer Aided Verification (CAV '01), LNCS 2102, 324–336, 2001.

[GS97] S. Graf and H. Saïdi: Construction of abstract state graphs with PVS, 9th Conference on Computer Aided Verification (CAV '97), LNCS 1254, 72–83, 1997.

[HR98] N. Heintze and J. G. Riecke: The SLam calculus: Programming with secrecy and integrity, 25th ACM Symp. on Principles of Programming Languages (POPL), 365–377, 1998.

[JSS97] S. Jajodia, P. Samarati and V. S. Subrahmanian: A logical language for expressing authorizations, IEEE Symp. on Security and Privacy, 31–42, 1997.

[JMT99] T. Jensen, D. Le Métayer and T. Thorn: Verification of control flow based security properties, IEEE Symp. on Security and Privacy, 89–103, 1999.

[JR02] S. Jha and T. Reps: Analysis of SPKI/SDSI certificates using model checking, IEEE Computer Security Foundations Workshop, 129–144, 2002.

[KS02] G. Karjoh and M. Schunter: A privacy policy model for enterprises, IEEE Computer Security Foundations Workshop, 271–281, 2002.

[K03] I. Kiselev: *Aspect-Oriented Programming with AspectJ*, SAMS, 2003.

[LR98] X. Leroy and F. Rouaix: Security properties of typed applets, 25th ACM Symp. on Principles of Programming Languages (POPL), 391–403, 1998.

[Mi96] J. Mitchell: *Foundations of Programming Languages*, The MIT Press, 1996.

[My99] A. C. Myers: JFLOW: Practical mostly-static information flow control, 26th ACM Symp. on Principles of Programming Languages (POPL), 228–241, 1999.

[ML98] A. C. Myers and B. Liskov: Complete, safe information flow with decentralized labels, 1998 IEEE Symp. on Security and Privacy, 186–197.

[NTS01a] N. Nitta, Y. Takata and H. Seki: Security verification of programs with stack inspection, 6th ACM Symp. on Access Control Models and Technologies (SACMAT), 31–40, 2001.

[NTS01b] N. Nitta, Y. Takata and H. Seki: An efficient security verification method for programs with stack inspection, 8th ACM Conf. on Computer and Communication Security (CCS), 68–77, 2001.

[OYF03] H. Ohmura, M. Yoshida and T. Fujiwara: An information flow analysis of programs with cryptographic function, Computer Security Symp., 277–282, 2003 (in Japanese).

[O95] P. Ørbæk: Can you trust your data? 6th International Joint Conference on the Theory and Practice of Software Development (TAPSOFT '95) , LNCS 915, 575–589.

[PD99] N. W. Paton and O. Díaz: Active Database Systems, ACM Computing Surveys, Vol.31, No.1, pp.63–103, 1999.

[PHS03] J. Pieprzyk, T. Hardjono and J. Seberry: *Fundamentals of Computer Security*, Springer, 2003.

[P94] G. Purnul: *Database Security*, Advances in Computers (M. Yovits Ed.), Vol.38, pp.1–72, 1994.

[RWW94] A. W. Roscoe, J. C. Woodcock and L. Wulf: Non-interference through determinism, 3rd European Symp. on Research in Computer Security (ESORICS), LNCS 875, 33–53, 1994.

[RS99] P. Y. A. Ryan and S. A. Shneider, Process algebra and non-interference, 12th IEEE Computer Security Foundations Workshop, 214–227, 1999.

[SV98] G. Smith and D. Volpano: Secure information flow in a muti-threaded imperative language, 25th ACM Symp. on Principles of Programming Languages (POPL), 355–364, 1998.

[T01] T. Tonouchi: An operational semantics of a 'small' ponder, July 2001. http://www.doc.ic.ac.uk/~tton/Semantics.pdf

[VS97] D. Volpano and G. Smith: A type-based approach to program security, 7th International Joint Conference on the Theory and Practice of Software Development (TAPSOFT '97), LNCS 1214, 607–621.

[Y01] R. Yokomori: Security Analysis Algorithm for Object-Oriented Programs, Master's Thesis, Osaka University, 2001.