PAPER

Policy Controlled System and Its Model Checking***

Shigeta KUNINOBU[†*a)], **Yoshiaki TAKATA**[†b)], **Naoya NITTA**[†**], ***and*** **Hiroyuki SEKI**[†c)], ***Members***

SUMMARY A policy is an execution rule (or constraint) for objects in a system to retain security and integrity of the system. We introduce a simple policy specification language and define its operational semantics. A new NFA construction algorithm that works in linear time is proposed and a model checking method for policy controlled system (PCS) is presented. We conducted verification of a sample PCS for hotel reservation by our automatic verification tool and the experimental results showed the efficiency of the proposed method.

key words: *policy control, policy controlled system, verification, model checking, pushdown system*

1. Introduction

Recently, much attention has been paid to an integrated approach to computer system management, called *policy control*. A policy is a rule describing when and on which condition a specified subject can (or cannot or must) perform a specified action on a specified target. A few specification languages have been proposed which have rich functions to describe policies in a concise way (see related works for details).

A policy controlled system (abbreviated as PCS) should have its own goal. For example, consider a digital contents service. A user who watches a movie should pay for it by executing a certain method that transfers the charge.

Model checking [3] is a well-known technique of automatically verifying whether a system satisfies such goals. Most of the existing model checking methods and tools assume that a system to be verified has finite state space. Hence, if a system has infinite state space, then the system should be transformed into an abstract system with finite state space. However, the abstract system does not always retain the desirable property of the original system, in which case the verification fails. 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 that involves recursive procedure calls. Recently, efficient model checking algorithms for PDS and an equivalent model (namely, recursive state machines (RSM)) have been proposed in [1], [2], [7], [8].

In this paper, we propose a formal method for verifying whether a given PCS satisfies a given property such as the security goals mentioned above, by using model checking for PDS. We first introduce a simple policy specification language like Ponder [5] as a referential model. The language has a structure sufficient for describing positive authorization (permission), negative authorization (prohibition) and obligation. We formally define the operational semantics of a policy controlled system (PCS) based on this language. Next, 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. We have implemented a verification tool for PCS. Our verification tool works as follows. First, a PDS is abstracted from a given PCS and a nondeterministic finite automaton (NFA) that accepts the set of all reachable states of the PDS is constructed. As described in Sect. 3.3, the NFA construction algorithm in this paper works in linear time which matches the algorithms in [1] and [2] 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 the paper is that the proposed method is one of the first attempts to verify a safety property of a policy controlled system. Also, this paper presents an application of model checking for PDS to a real world verification problem and shows verification results conducted on an automatic tool.

The rest of this paper is organized as follows. In Sect. 2, we introduce a simple policy specification language suitable for distributed policy control and define the operational semantics of PCS. In Sect. 3, a pushdown system (PDS) and Esparza et al.'s results on model checking for PDS are reviewed. Next, we propose an efficient algorithm that constructs an NFA accepting the set of all reachable states of a given PDS. Section 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 5 concludes the paper.

Manuscript received August 27, 2003.
Manuscript revised November 5, 2004.

[†]The authors are with the Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan.

[*]Presently, with the Research and Development Center, Toshiba Corporation.

[**]Presently, with the Faculty of Science and Engineering, Konan University.

[***]Section 2 of this paper is partially based on [16].

a) E-mail: shigeta.kuninobu@toshiba.co.jp
b) E-mail: y-takata@is.naist.jp
c) E-mail: seki@is.naist.jp

DOI: 10.1093/ietisy/e88–d.7.1685

Related Works Recently, much effort has been devoted to applying model checking to security verification. Model checking for finite state systems has a long history [3], and a number of model checkers have been developed. In [19], a computer network configuration together with a set of vulnerabilities of each computer node is represented as a state, and a possible attack to the network can be obtained as a counterexample against a safety property by using model checker SMV. In [14], Jha et al. take a similar approach to [19] to security verification problems of an intrusion detection system (IDS) by using model checker NuSMV.

Efficient algorithms and complexity of LTL and CTL* model checking for PDS are extensively studied in [7], [8]. Verification results using an automatic verification tool are reported in [9]. Model checking algorithms for RSM, which is equivalent to PDS, are studied in [1], [2]. Their algorithms work in the same complexity as [7]'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 [12]. 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. [17], [18] improved the result of [12] 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 [8]. 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) [8], [18]. In [18], 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 [13].

A number of policy specification languages have been proposed (for example, [5], [11], [15]). In [11] 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 [15], the model of [11] is extended so that one can express an obligatory policy.

Ponder [5], [6] is a general purpose policy specification language in which one can specify obligatory, conditional, and data-dependent policies. Also the formal semantics of a subset of the language is defined in [6], [20]. However, they define only the execution order of methods under given policies, leaving the rest of the system as a blackbox. In this paper, the entire behavior of a system under given policies is formally defined by explicitly describing the change of runtime stack configuration.

2. Policy Controlled System

We introduce a policy controlled multi-object system, and describe its formal semantics. Our policy controlled system models a system in which objects (data and program code) provided by more than one participants run on a single virtual machine. In other words, the system is a single-threaded multi-object system. A typical target of our model is a digital content service in which a digital video and audio is encapsuled in an object and distributed to users, though the model is not restricted to it. In the digital content service, both the object (say B) containing digital video and a user's agent (say A) run on the user's machine, and when the user wants to watch the video in B, A would request B to replay the video by calling B's method. Moreover, the request is issued only when A's policy permits, and the request is accepted by B only when B's policy permits.

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 Sect. 2.1, we propose a simple policy specification language. In Sect. 2.2, we formally define the structure and the behavior of a PCS.

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 that participate in the method execution. For example, object A can play the movie contained in object B by executing method $B.play$ only when both the policies of A and B

```
⟨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 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⟩
```

Fig. 1 Syntax of a policy specification language.

Table 1 Form of ⟨operation unit1⟩ and ⟨operation unit2⟩.

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

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 Fig. 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 1, x and y stand for any objects other than 'this.'
- 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 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 that 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
```

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 Sect. 2.2.1, we define a simple model of objects and programs. In Sect. 2.2.2, we introduce several concepts about policies and define the behavior of a system with policies. In Sect. 2.2.3, the 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.

2.2.1 Objects and Programs

Preliminaries In the following, let V, V_1, and V_2 be arbitrary sets of variables. A *substitution* for V is a mapping which maps a variable in V to its value. For a substitution σ for V and an expression e with variables in V, let $\sigma(e)$ denote the value of e obtained by replacing each occurrence of a variable x in e with $\sigma(x)$. For a substitution σ for V and a variable $x \in V$, let $\sigma[x := v]$ denote the substitution which is the same as σ except that the value of x is v. For substitutions σ for V_1 and μ for V_2 such that $V_1 \cap V_2 = \emptyset$, let $\sigma \circ \mu$ denote the union substitution for $V_1 \cup V_2$ such that $\sigma \circ \mu(x) = \sigma(x)$ for all $x \in V_1$ and $\sigma \circ \mu(y) = \mu(y)$ for all $y \in V_2$.

Objects Each object has a finite number of attributes and methods defined by its class. 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. A global state σ of O is a substitution for $\mathrm{Att}_O = \{o.a \mid o \in O \text{ and } a \text{ is an attribute of } o\}$.

Programs The body of a method $o.m$ is a program which is represented by a directed graph as shown in Fig. 2. A program is a tuple (NO, TG, IS, IT, 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 which maps a node to its label.

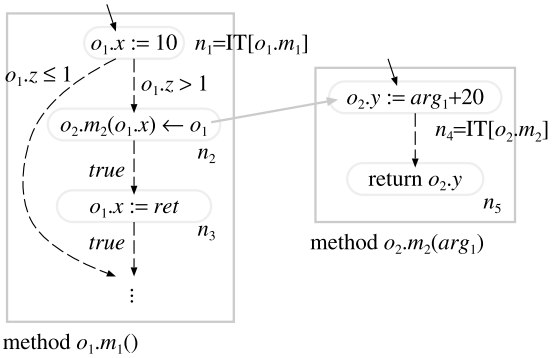

Fig. 2 A sample program.

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. A state μ of the local variables of $o.m$ is a substitution for VAR[$o.m$]. Note that the value of an expression $e \in$ EXP[$o.m$] at a global state σ and a state μ of local variables is $\sigma \circ \mu(e)$. The state of local variables in which the values of all variables are undefined is denoted by $\perp$.

2.2.2 Behavior of PCS

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 $1 \le j \le n$, let $Policy(O) = \bigcup_{1 \le j \le n} P'_j$ 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.

$\sigma \models mod(s, t, m)$ if and only if $\exists p \in Policy(O)$,
$\exists \theta$: a substitution for the variables in p,

p = "policy *mod* ... $B.m \leftarrow A$ if *Cond*",
$\sigma(\theta(Cond)) = true,\ \theta(A) = s,\ \theta(B) = t.$

Note that the substitution θ is a mapping which maps a variable in a policy p to an object, and $\theta(Cond)$ is the expression which is the same as *Cond* except that every variable x declared in p is replaced with $\theta(x)$.

If $\sigma \models \text{auth+}(s,t,m)$ and $\sigma \models \text{auth}-(s,t,m)$, then we say that σ causes a conflict between policies upon the operation (s,t,m). Using an arbitrary conflict resolution method (cf. [11]), we define $\text{CAN}(\sigma, t.m \leftarrow s)$ as a predicate which is true if the operation $t.m \leftarrow s$ is permitted when the global state is σ.

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

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

Order of Obligations We assume a total order $\prec$ over the set $\text{oblg}(\sigma, ev)$ of obligations which represents the order of performing the obligations in our model, so that a programmer can know and control the order. In a real system, the order would be decided according to a criterion such as the order of appearing in the policy specification. Thus

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

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

$$\text{Foblg}(\sigma, ev) = (n_{\text{oblg}}[op_1], \bot) : \ldots : (n_{\text{oblg}}[op_l], \bot),$$

where $n_{\text{oblg}}[op]$ is a special node newly introduced here for any obligatory operation op. Note that $n_{\text{oblg}}[op]$ does not belong to any method and $\text{IS}(n_{\text{oblg}}[op])$ is defined as $\text{IS}(n_{\text{oblg}}[op]) = op$. Let NO_{oblg} be the set of all $n_{\text{oblg}}[op]$'s.

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. Each stack frame (or simply, frame) is a triple (n, μ, ef), where $n \in \text{NO}[o.m]$ 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 $ef = true$, 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 $(n, \mu, false)$ to (n, μ) and $(n, \mu, true)$ 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.

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

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

$$\text{(CALL)}\quad \frac{\begin{array}{l}\text{IS}(n) = \text{"}o_2.m_2(e_1,\ldots,e_k) \leftarrow o_1\text{"} \\ \text{CAN}(\sigma, o_2.m_2 \leftarrow o_1) = true \\ f_2 = (\text{IT}[o_2.m_2], \bot[arg_1 := \sigma \circ \mu(e_1)] \ldots [arg_k := \sigma \circ \mu(e_k)]) \\ \text{Foblg}(\sigma, \text{beginning of } o_2.m_2 \leftarrow o_1) = f_{\text{x},1} : f_{\text{x},2} : \ldots : f_{\text{x},\ell}\end{array}}{\begin{array}{l}(\sigma,\ (n,\mu) : \xi) \to \\ \quad (\sigma,\ f_{\text{x},1} : f_{\text{x},2} : \ldots : f_{\text{x},\ell} : f_2 : (n,\mu) : \xi)\end{array}}$$

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

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

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

Fig. 3 Inference rules which define the transition relation.

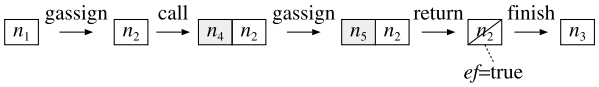

Fig. 4 Behavior of the system with the program in Fig. 2 and no obligation.

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 Fig. 3. Note that a method invocation $o_2.m_2 \leftarrow o_1$ is performed only when $\text{CAN}(\sigma, o_2.m_2 \leftarrow o_1) = true$. 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 Figs. 4 and 5).

2.2.3 Exception Handling

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

[†]In the case that we cannot assume such an order over $\text{oblg}(\sigma, ev)$, we can define $\text{Foblg}(\sigma, ev)$ as the set of sequences of stack frames which are the permutations of $(n_{\text{oblg}}[op_1], \bot) : \ldots : (n_{\text{oblg}}[op_l], \bot)$, though it may increase the complexity of verification.

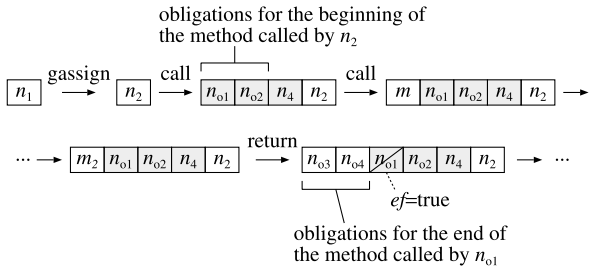

Fig. 5 Behavior of the system with obligations.

```
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
```

Fig. 6 A sample obligation.

(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. In the extended language, we can write an arbitrary program code in the main clause of a specification of an obligation policy. Figure 6 shows a sample specification and part (a) of the figure is the main clause of the specification. Note that part (a) of Fig. 6 is a text version of a program with an exception edge described below. We assume that the main clause 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 Fig. 7). Note that in our model an exception occurs only at a method call and thus IS(n_1) should be a method call. $n_1 \overset{r:ty}{\rightarrow}_{\mathrm{EG}} n_2$ 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 $n \overset{r:ty}{\rightarrow}_{\mathrm{EG}} n_2$ 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).

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.

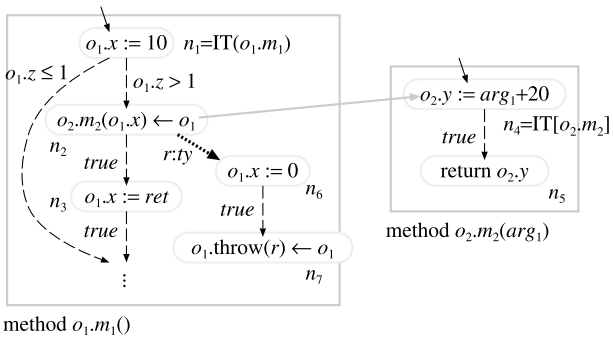

Fig. 7 A sample program with an exception edge.

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

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

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

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

Fig. 8 Inference rules for exception handling.

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 the set of sequences 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 Fig. 8 to the set of the rules in Fig. 3. We also add "$m_2 \neq$ throw" to the premise of the rule (CALL) in Fig. 3.

3. Pushdown System

To verify a policy controlled system introduced in Sect. 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 [7]. We then propose a new algorithm for computing the reachable

configurations of a PDS. This algorithm works faster than the algorithm in [7] and matches the algorithms in [1], [2] when we use it for the verification of a policy controlled system.

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 $\{c' \in P \times \Gamma^* \mid \exists c \in C.\ c \Rightarrow^* c'\}$, 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.

3.2 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 [8], 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 3.1 ($\mathcal{P}$-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 3.1: [8] 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 3.1. We describe the decision algorithm in the next theorem for referential convenience to the following sections.

Theorem 3.2: 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 3.1, 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 Computing Reachable Configurations

The proof of theorem 3.2 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 [18]. Instead 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 Fig. 9. The following lemma shows the meaning of R_M.

Lemma 3.3: $(p, a, q) \in R_M$ 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 $\langle q, \epsilon\rangle$. □

$$\frac{\langle p,a\rangle \hookrightarrow \langle q,\varepsilon\rangle}{(p,a,q)\in R_M} \qquad \frac{\begin{array}{c}\langle p,a\rangle \hookrightarrow \langle q_1,b_1b_2\ldots b_k\rangle \\ (q_1,b_1,q_2)\in R_M \\ \cdots \\ (q_{k-1},b_{k-1},q_k)\in R_M \\ (q_k,b_k,q)\in R_M\end{array}}{(p,a,q)\in R_M}$$

Fig. 9 Inference rules for R_M.

$$\frac{}{X_{p,a}\to pa\in D} \qquad \frac{\langle p,a\rangle \hookrightarrow \langle q,\varepsilon\rangle}{X_{p,a}\to q\in D}$$

$$\frac{\begin{array}{c}\langle p,a\rangle \hookrightarrow \langle q_1,b_1b_2\ldots b_k\rangle \\ (q_1,b_1,q_2)\in R_M \\ (q_2,b_2,q_3)\in R_M \\ \cdots \\ (q_{j-1},b_{j-1},q_j)\in R_M\end{array}}{X_{p,a}\to X_{q_j,b_j}b_{j+1}b_{j+2}\ldots b_k\in D}$$

Fig. 10 Inference rules for G_M.

Using R_M, we define the left linear grammar $G_M = (V,T,D,I)$ for the given PDS $M = (P,\Gamma,\Delta,q_0,\omega)$ where $V = Q\times\Gamma$ is the set of variables (for readability, we write $X_{p,a}$ to represent $(p,a)\in V$), $T = Q\cup\Gamma$ is the set of terminal symbols, D is the set of productions and is defined as the smallest set which satisfies the rules in Fig. 10, and $I = X_{q_0,\omega}$ is the start symbol.

Lemma 3.4: $w\in L(G_M)$ 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 Lemma 3.3 and induction on the length of the derivation of w.
(If) By Lemma 3.3 and induction on the number of transitions between $\langle p,a\rangle$ and $\langle q,\gamma\rangle$. □

The following theorem can be proved by Lemma 3.4 and the form of the inference rules in Figs. 9 and 10.

Theorem 3.5: For a normal form PDS $M = (P,\Gamma,\Delta,q_0,\omega)$, a left linear grammer G_M such that $L(G_M) = \{q\gamma \mid \langle q_0,\omega\rangle \Rightarrow^* \langle q,\gamma\rangle\}$ 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, $|P_\epsilon|\le \min(|P|,|\Delta|)$. 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 Sect. 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.

4. Model Checking Policy Controlled Systems

In this section, we present a method for verifying a safety property of a PCS. When every reachable configuration of a PCS (or PDS) S belongs to a property Ψ (see Sect. 3.2), we simply say that S satisfies Ψ. Similar to the case of 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 Sect. 3. By the soundness of the abstraction (Theorem 4.2), if $S^\sharp$ is known to satisfy Ψ then S is also guaranteed to satisfy Ψ.

4.1 Abstracting PDS from PCS without Exception Handling

We define a transformation which abstracts PDS from PCS without exception handling. Abstraction from PCS with exception handling is described in Sect. 4.2. 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, $\Gamma = \{n, \not\!\!\downarrow \mid n$ is a node in $Prog\}$, the 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 Fig. 11. Each rule in Fig. 11 has its counterpart in Fig. 3. In fact, rules in Fig. 11 are obtained from Fig. 3 by discarding values of global and local variables. Note that rule (RETURN) in Fig. 3 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, we define $\xi^\sharp = n_1n_2\ldots n_k$. That is, $\xi^\sharp$ is obtained from ξ by discarding values of all local variables. $\mathrm{CAN}^\sharp$ and $\mathrm{Foblg}^\sharp$ in Fig. 11 are the abstract versions of CAN and Foblg, respectively. $\mathrm{CAN}^\sharp$ is any predicate such that $\mathrm{CAN}^\sharp(o_2.m \leftarrow o_1) = true$ whenever $\mathrm{CAN}(\sigma, o_2.m \leftarrow o_1) = true$ 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 $\Psi^\sharp = \{c' \mid$ every c such that $c^\sharp = c'$ satisfies $c\in\Psi\}$. We can obtain the following soundness property of the abstraction.

$$\text{(GASSIGN)}\quad \frac{\mathrm{IS}(n) = \text{“}o.a := e\text{”} \qquad n \xrightarrow{e_2} n_2}{\langle p, n\rangle \hookrightarrow \langle p, n_2\rangle}$$

$$\text{(LASSIGN)}\quad \frac{\mathrm{IS}(n) = \text{“}r := e\text{”} \qquad n \xrightarrow{e_2} n_2}{\langle p, n\rangle \hookrightarrow \langle p, n_2\rangle}$$

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

$$\text{(RETURN)}\quad \frac{\begin{array}{l}\mathrm{IS}(m) = \text{“return } e\text{”}\\ \mathrm{IS}(n_1) = \text{“}o_2.m_2(e_1,\ldots,e_k) \leftarrow o_1\text{”}\\ \mathrm{Foblg}^\sharp(\text{end of } o_2.m_2 \leftarrow o_1) \ni f_{\mathrm{x},1} : f_{\mathrm{x},2} : \ldots : f_{\mathrm{x},\ell}\end{array}}{\begin{array}{l}\langle p, m\rangle \hookrightarrow \langle q, \varepsilon\rangle\\ \langle q, n_1\rangle \hookrightarrow \langle p, f_{\mathrm{x},1} f_{\mathrm{x},2} \cdots f_{\mathrm{x},\ell}\, n_1'\rangle\end{array}}$$

$$\text{(FINISH)}\quad \frac{n_1 \notin \mathrm{NO}_{\mathrm{oblg}} \qquad n_1 \xrightarrow{e} n_2}{\langle p, n_1'\rangle \hookrightarrow \langle p, n_2\rangle}$$

$$\text{(OFINISH)}\quad \frac{n_1 \in \mathrm{NO}_{\mathrm{oblg}}}{\langle p, n_1'\rangle \hookrightarrow \langle p, \varepsilon\rangle}$$

Fig. 11 Abstraction rules.

$$\text{(THROW1)}\quad \frac{\begin{array}{l}\mathrm{IS}(n) = \text{“}o_1.\mathrm{throw}(e) \leftarrow o_1\text{”}\\ et \in \mathrm{EVAL}(e)\end{array}}{\langle p, n\rangle \hookrightarrow \langle et, n\rangle}$$

$$\text{(POLICY\_EX)}\quad \frac{\begin{array}{l}\mathrm{IS}(n) = \text{“}o_2.m_2(e_1,\ldots,e_k) \leftarrow o_1\text{”}\\ \mathrm{CAN}^\flat(o_2.m_2 \leftarrow o_1) = false\end{array}}{\langle p, n\rangle \hookrightarrow \langle \mathrm{et}_{\mathrm{policy}}, n\rangle}$$

$$\text{(CATCH)}\quad \frac{n \xrightarrow{r:et}_{\mathrm{EG}} n_2}{\begin{array}{l}\langle et, n\rangle \hookrightarrow \langle p, n_2\rangle\\ \langle et, n_1'\rangle \hookrightarrow \langle p, n_2\rangle\end{array}}$$

$$\text{(THROW2)}\quad \frac{n \xrightarrow{r:et}_{\mathrm{EG}} n_2 \text{ does not hold for any } n_2 \text{ and } r.}{\begin{array}{l}\langle et, n\rangle \hookrightarrow \langle \hat{et}, n\rangle\\ \langle et, n_1'\rangle \hookrightarrow \langle \hat{et}, n_1'\rangle\end{array}}$$

$$\text{(CASTOFF1)}\quad \frac{n_2 \notin \mathrm{NO}_{\mathrm{oblg}}}{\begin{array}{l}\langle \hat{et}, n_2\rangle \hookrightarrow \langle et, \varepsilon\rangle\\ \langle \hat{et}, n_2'\rangle \hookrightarrow \langle et, \varepsilon\rangle\end{array}}$$

$$\text{(CASTOFF2)}\quad \frac{n_2 \in \mathrm{NO}_{\mathrm{oblg}}}{\begin{array}{l}\langle \hat{et}, n_2\rangle \hookrightarrow \langle \tilde{et}, \varepsilon\rangle\\ \langle \hat{et}, n_2'\rangle \hookrightarrow \langle \tilde{et}, \varepsilon\rangle\end{array}}$$

$$\text{(CASTOFF3)}\quad \frac{true}{\begin{array}{l}\langle \tilde{et}, n_1\rangle \hookrightarrow \langle \hat{et}, n_1\rangle\\ \langle \tilde{et}, n_1'\rangle \hookrightarrow \langle et, n_1'\rangle\end{array}}$$

Fig. 12 Abstraction rules for exception handling.

Lemma 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 Fig. 3 and rules in Fig. 11. □

Theorem 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 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 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 Sect. 3, then we can conclude that the answer to the original problem for S and Ψ is also affirmative.

4.2 Abstracting PDS from PCS with Exception Handling

We define $\mathrm{CAN}^\flat$ as any predicate such that $\mathrm{CAN}^\flat(o_2.m \leftarrow o_1) = false$ whenever $\mathrm{CAN}(\sigma, o_2.m \leftarrow o_1) = false$ for some σ. While $CAN^\sharp$ in Sect. 4.1 is an over-estimation of CAN (i.e., the set of events that make $\mathrm{CAN}^\sharp$ true is larger than or equal to that of CAN), $\mathrm{CAN}^\flat$ is an under-estimation of CAN. We also define $\mathrm{et}_{\mathrm{policy}} = \mathrm{typeof}(\mathrm{ex}_{\mathrm{policy}})$ and $\mathrm{EVAL}(e) = \{et \mid et = \mathrm{typeof}(\sigma \circ \mu(e))$ for $\exists\sigma$ and $\exists\mu\}$ for each expression e appearing as the actual argument of an invocation of throw.

The PDS $S^\sharp$ abstracted from a PCS S with exception handling is defined as follows. The set P of control locations of $S^\sharp$ is defined as $P = \{p, q\} \cup \{et, \hat{et}, \tilde{et} \mid et \in Ex\}$ where Ex is the set of types of exceptions used in S. The set Δ of transition rules is defined by Fig. 11 and Fig. 12. The first four rules in Fig. 12 correspond to the exception handling rules in Fig. 8. The other three rules in Fig. 12 are for simulating CASTOFF in Sect. 2.2.3.

For each configuration $c = (\sigma, \xi)$ of S, let us define

$$c^\sharp = \begin{cases} \langle \mathrm{typeof}(ex), \nu\rangle & \xi = ex : \nu \text{ and} \\ & ex \text{ is an exception,} \\ \langle p, \xi^\sharp\rangle & \text{otherwise.} \end{cases}$$

Then, the soundness property (Theorem 4.2) of the abstraction from PCS with exception handling also holds, though the time complexity increases from $O(\|\Delta\|)$ to $O(|Ex|^2 \cdot \|\Delta\|)$.

4.3 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 Fig. 11 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

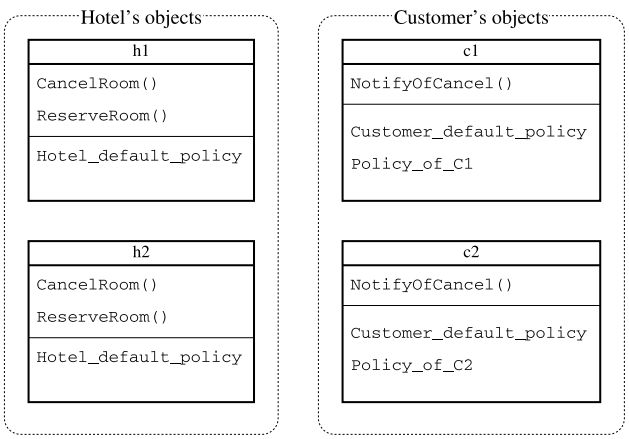

Fig. 13 A hotel reservation system.

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 Sects. 3.3 and 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 $S^\sharp$.

(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 4.1: Consider a simple online hotel reservation system which is executed on a server computer and 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. If the system receives a request from a hotel or a customer, then it invokes a corresponding method of his/her object in a sequential manner. Every hotel's object and every customer's object must also 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 (Fig. 13). They must have the following policies.

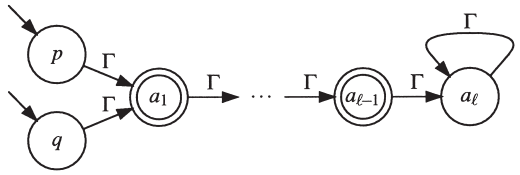

Fig. 14 A verification property.

Table 2 Verification profiles of example 4.1.

# 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 (2 GHz), 1 GB 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*. Let $S^\sharp = (\{p, q\}, \Gamma, \Delta, p, n_0)$ be the abstracted PDS from the hotel reservation system and l the threshold length. The verification property can be represented as a deterministic $\mathcal{P}$-automaton $\mathcal{A}_\Psi = (Q, \Gamma, \delta, \{p, q\}, F)$ shown in Fig. 14. 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 2.

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 2, 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 2.

5. Conclusion

In this paper, a simple policy specification language was defined together with its operational semantics. An automatic verification method for PCS using PDS model checking was also proposed. Experimental results on verification of a sample PCS for hotel reservation showed the efficiency of the proposed method.

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 [3]. Such abstraction techniques include abstract interpretation, program slicing [4] and predicate abstraction [10]. We would like to integrate these techniques into our verification method.

Acknowledgment

The authors would like to thank Mr. Daigo Taguchi and Mr. Masayuki Nakae of NEC Corporation for their valuable discussions on the design of policy specification language.

References

[1] R. Alur, K. Etessami, and M. Yannakakis, "Analysis of recursive state machines," CAV 2001, LNCS 2102, pp.207–220, 2001.
[2] M. Benedikt, P. Godefroid, and T. Reps, "Model checking of unrestricted hierarchical state," ICALP 2001, LNCS 2076, pp.652–666, 2001.
[3] E.M. Clarke, O. Grumberg, and D. Peled, Model Checking, MIT Press, 1999.
[4] 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," Int'l Conf. on Software Engineering, pp.439–448, 2000.
[5] N.C. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," POLICY 2001, LNCS 1995, pp.18–38, 2001.
[6] N.C. Damianou, A Policy Framework for Management of Distributed Systems, Ph.D. Thesis, Imperial College of Science, Technology and Medicine, 2002. http://www-dse.doc.ic.ac.uk/Research/policies/ponder/thesis-ncd.pdf
[7] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon, "Efficient algorithms for model-checking pushdown systems," CAV 2000, LNCS 1855, pp.232–247, 2000.
[8] J. Esparza, A. Kučera, and S. Schwoon, "Model-checking LTL with regular variations for pushdown systems," TACS01, LNCS 2215, pp.316–339, 2001.
[9] J. Esparza and S. Schwoon, "A BDD-based model checker for recursive programs," CAV 2001, LNCS 2102, pp.324–336, 2001.
[10] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," CAV 97, LNCS 1254, pp.72–83, 1997.
[11] S. Jajodia, P. Samarati, and V.S. Subrahmanian, "A logical language for expressing authorizations," IEEE Symp. on Security and Privacy, pp.31–42, 1997.
[12] T. Jensen, D. Le Métayer, and T. Thorn, "Verification of control flow based security properties," IEEE Symp. on Security and Privacy, pp.89–103, 1999.
[13] S. Jha and T. Reps, "Analysis of SPKI/SDSI certificates using model checking," IEEE Computer Security Foundations Workshop, pp.129–144, 2002.
[14] S. Jha, O. Sheyner, and J. Wing, "Two formal analyses of attack graphs," IEEE Computer Security Foundations Workshop, pp.49–63, 2002.
[15] G. Karjoh and M. Schunter, "A privacy policy model for enterprises," IEEE Computer Security Foundations Workshop, pp.271–281, 2002.
[16] S. Kuninobu, Y. Takata, D. Taguchi, M. Nakae, and H. Seki, "A specification language for distributed policy control," 4th Int'l Conf. on Information and Communications Security, LNCS 2513, pp.386–398, 2002.
[17] N. Nitta, Y. Takata, and H. Seki, "Security verification of programs with stack inspection," 6th ACM Symp. on Access Control Models and Technologies, pp.31–40, 2001.
[18] 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, pp.68–77, 2001.
[19] R.W. Ritchey and P. Ammann, "Using model checking to analyze network vulnerabilities," IEEE Symp. on Security and Privacy, pp.156–165, 2000.
[20] T. Tonouchi, An operational semantics of a 'small' ponder, July 2001. http://www.doc.ic.ac.uk/~tton/Semantics.pdf

Shigeta Kuninobu received the B.E. degree in electronics from Doshisha University, Japan, in 1999, and received the M.E. and Ph.D. degree in information science from Nara Institute of Science and Technology, Japan, in 2001 and 2004, respectively. He joined Research and Development Center, Toshiba Corporation, Japan, in 2004, where he has been working on the research and development of reliable software.

Yoshiaki Takata received the Ph.D. degree in information and computer sciences from Osaka University, Osaka, Japan, in 1997. He has been an Assistant Professor at Nara Institute of Science and Technology since 1997. His current research interests include formal specification and verification of software systems.

Naoya Nitta has received his Ph. D. degree in information science from Nara Institute of Science and Technology, Japan, in 2002. Since 2002 he has been an assistant professor in Nara Institute of Science and Technology. His current research interest is verification of software and secure systems.

Hiroyuki Seki received the Ph.D. degree in information and computer sciences from Osaka University, Japan, in 1987. He was with Osaka University as an Assistant Professor in 1990–1992 and an Associate Professor in 1992–1994. In 1994, He joined the faculty of Nara Institute of Science and Technology, where he has been a Professor since 1996. His research interest includes formal language theory and formal approach to software development.