An Algebraic Basis for Specifying and Enforcing Access Control in Security Systems

Claus Pahl
School of Computer Applications, Dublin City University
Dublin, Ireland
cpahl@compapp.dcu.ie

Abstract

Security services in a multi-user environment are often based on access control mechanisms. Static aspects of an access control policy can be formalised using abstract algebraic models. We integrate these static aspects into a dynamic framework considering requesting access to resources as a process aiming at the prevention of access control violations when a program is executed. We use another algebraic technique, monads, as a meta-language to integrate access control operations into a functional programming language. The integration of monads and concepts from a denotational model for process algebras provides a framework for programming of access control in security systems.

1 Introduction

Confidentiality and integrity are important security aspects [Pfl97, Gol99]. Guaranteeing confidentiality and integrity of information is the objective in deploying an access control mechanism. Confidentiality denotes the denial of access to sensitive information to unauthorised users. Information must not be disclosed to anyone who is not authorised to access it. The system must maintain the continuing integrity of the information, i.e. it must not be corrupted by unauthorised modification. Controlling access to resources will be the issue under consideration in this paper. We will present a formal description of access control mechanisms using algebraic operations, similar to [Amo94]. We define the static aspects and add, additionally to existing algebraic access control models, concepts to describe the dynamics of requesting access to resources as abstract processes in the style of [Hoa85, Mil89]. We will present an abstract model of the dynamics, which is adaptable to different static models and which can easily be integrated into a functional specification or programming language using monads as a meta-language [Mog91] for integration. The static model formalises an access control policy. The dynamic part of the abstract model describes an enforcement mechanism for an access control policy aiming at the prevention of security violations.

Algebraic techniques will be used to describe principles of access control formally. This makes the implementation of these mechanisms in a functional style straightforward. We see that requesting access to resources can be modeled as a process. This is modelled using a standard algebraic model for process algebras. Static and dynamic aspects of accessing resources can be modularly integrated into a functional programming language, again using algebraic techniques, in particular using a monad-style approach [Wad92, Mog91]. The result is a formal framework which provides all advantages of algebraic techniques

as a specification notation, such as abstractness, preciseness, conciseness, and reasoning. All concepts used are easy to implement in a functional programming language such as Haskell [HJW+92]. A kernel of a security system realising access control can be implemented based on a security policy. This can be seen as a first step, a mathematical kernel, towards concurrent and distributed programming with a functional language such as Haskell that can be web-based.

We will start formalising security concepts in Section 2. This results in a formal description of simple static access control. The dynamics of processes requesting access to objects shall then be described. This starts with the presentation of a domain representing the processes in Section 3. The following Section 4 introduces an access control monad which allows us to integrate dynamic descriptions into a functional context, i.e. into a programming or specification language. Section 5 implements the concepts in Haskell. We end with related work and conclusions.

2 Static Aspects of Access Control

Amoroso [Amo94] describes the formal specification of access control properties. The specification of these properties forms a security policy. Predicates, such as

$$checkCapability : Subject \times Object \times AccessOperation \to Boolean$$

are suggested[1]. Subjects are active entities in the system such as users or processes. Subjects request operations on the available resources in the system. For example, a user can request to read a file. These predicates can be defined by $checkCapability(s, o, a) \Leftrightarrow P$ where P is some condition.

Before we are going to specify security elements, we shall make a few brief remarks on our notation. We will specify an *abstract model*, specifying properties in an algebraic specification style. We will use the usual domain constructors such as $\times$ (Cartesian product), $\to$ (function space), or $\mathcal{P}$ (powerset). The underlying model in which the elements are interpreted is the category of sets and total functions (later on we will add more structure to the category).

A permission is an instance of type $Object \to AccessOperation$, e.g. *permission(fn)* = *read* in a file server specification says that file *fn* can be read.

$$Capability = Object \to \mathcal{P}AccessOperation$$

A capability is a function from objects into the set of access operations. For a particular object, it would list all permitted forms of access.

Alternatively, instead of focussing on individual permissions, we could have modelled an *access control matrix*, giving the permitted access operations for each pair of subject and object:

$$Matrix = Subject \times Object \to \mathcal{P}AccessOperation$$

This is the most general form. Operating systems, such as Unix, administer access rights in form of access control lists - called capabilities here. Capabilities are usually associated to users in a system,

[1] Some authors, e.g. Amoroso, use agent for subject, action for access operation, resource for object. We stick to the terminology introduced e.g. in the Bell-LaPadula security model - see recent text books on security such as [Gol99].

which can be modelled by $Subject \rightarrow Capability$ – which can be expanded to $Subject \rightarrow Object \rightarrow \mathcal{P}AccessOperation$. In the category Set this is isomorphic to our $Matrix$ model. This shows that access control lists, access control matrices and capabilities are essentially only different representations of the same security policy.

Specialisations of the model might impose constraints on the domains. For instance for files, an object can be denoted by any syntactically correct path expression (e.g. a file name, directory name, or a wildcard); operations can be read, write, delete, and execute. To point out the importance of modelling capabilities precisely, we will give an example: a write permission on a whole file system allows us to substitute capability definitions, which would allows us to do everything then. This dangerous permission has to be prevented.

Subjects in security systems need permission to perform operations. The security policy of a running system expresses these capabilities of subjects:

$$Policy = Subject \rightarrow Capability$$

A security policy shall now be specified using an algebraic specification style.

SecurityPolicy =
  domains
    Policy = *Subject* → *Capability*
  constants
    ACL_trusted : Capability,
    ACL_non_trusted : Capability,
    implications : AccessOperation → $\mathcal{P}$ *AccessOperation,*
  operations
    isTrusted : Subject → *Boolean,*
    checkCapability : Subject × *Object* × *Access Operation* → *Boolean,*
    implies : Capability × *Capability* → *Boolean*
  equations
    checkCapability(s,o,a) = if *isTrusted(s)* then
        a ∈ *ACL_trusted(o)*
      else
        a ∈ *ACL_non_trusted(o)*

The definition of the constants is dependent on a particular security policy implementation. The operation *checkCapability* checks a capability for a subject. An operation *implies* relates two capabilities. *implies* calculates the closure of *implications*, which should be considered in the specification of *checkCapability*. We distinguish between two access control lists, i.e. two sets of capabilities. $ACL\_trusted$ defines capabilities for trusted subjects, and $ACL\_non\_trusted$ for non-trusted ones. Definitions for $ACL\_trusted$, $ACL\_non\_trusted$, the access control lists, and for $implications$ are mere type definitions, they have to be customised for each concrete security policy. Possible definitions for the access control lists for a file system could look like $ACL\_trusted(fn1) = \{read\}$, $ACL\_trusted(fn2) = \{read, write\}$ or $ACL\_non\_trusted(fn) = \{read\}$. Implications could include $implications(write) = read$. Implications typically represent a reflexive and transitive relation. Antisymmetry (which in combination with reflexivity and transitivity forms a partial ordering) is not a necessary condition.

The model could be extended and made more flexible, for example by operations to create, modify and delete capabilities in a modifiable security policy. Our model provides rudimentary architectural support for more advanced models such as the Bell-LaPadula model [Gol99]. Our model covers still, despite its simplicity, the access control concepts of the Java 1.0 programming language [MF99, Fla99]. Since our focus is on dynamic issues, we do not pursue these directions.

3 Dynamic Aspects – The Access Control Domain

The previous section described a static model of a security policy which defines the capabilities of subjects, i.e. what are subjects permitted to do, or, which actions are they permitted to execute on which objects.

Now assume a running program in a security-aware context. During the execution of a sequence of computations, some of these computations might involve access operations subject to permission. This process of subjecting each step to permission shall be modelled on an abstract level. Without looking at any implementation-relevant issues, we just want to consider computations requesting some kind of operation and an authority, or a guard, which grants or denies permission to perform the request. Therefore, the *abstract model* of the *dynamics of access control* which we are seeking should only consist of the described process itself, requests and a guard.

In this section, we will introduce an algebraic model for the dynamics of requests and access control. We will show how this abstract model can be used to model the processing of requests to access an object with some operation in a security system. We will relate the model to a common denotational model for process algebras. In the following section, we will accommodate this model in a monad in order to integrate access control into a functional framework.

3.1 Abstract Requests

The behaviour of a process p requesting access to objects via operations on objects shall be expressed by an element from an abstract recursive process domain P. We are going to formulate the process domain in the category of posets and continous functions which adds more structure to our original framework. We suggest the following domain P to describe the dynamics of access control:

$$P = ((R \to P) \times G)_\perp$$

The domain R denotes requests. Applied to our concrete model from the previous section a request contains an object and the access operation which is supposed to be performed on the object. The set G models a guard - which would, if applied to the concrete model, represent the current security policy. Here, we assume $G = \mathcal{P}R$ meaning that a guard is a set of requests that are permitted. In the case that a request is granted – determined by a guard $g \in G$ – the computation proceeds by *performing* the request and the next request can be looked at. Technically, that means that the current process state is transformed into the next state. The new process state is determined by $\mathit{pfm}(r)$ for a current process state $(\mathit{pfm}, g) : P$ and a request $r : R$. The mapping *pfm* is an element of type $R \to P$. It models the operation performing the request which transforms the process into its next state. A process can behave chaotically, expressed by $\perp$. This idea could be refined by providing a set of exceptions representing e.g. different security violations instead of a single symbol $\perp$. R and G might be internally structured domains. This is the case if this abstract model is applied

to the concrete static model of Section 2. We will discuss this later based on the basic access control entities described in the previous section. For the purpose of looking at access control dynamics, we do not need to model objects or operations explicitly. This has led us to a more abstract model.

In order to allow only processes to proceed whose requests are permitted, we introduce a constraint on the recursive domain P, called the *security constraint*:

$$\forall\, \mathit{pfm} : (R \to P), g : G \,.\, dom(\mathit{pfm}) \in g$$

for requests $dom(\mathit{pfm}) : R$ and guards $g : G$. This constraint expresses that requests have to be granted for a process in order to transfer into a new state – remember that g is a set of permitted requests. This can be illustrated by defining $R = Capability$ and $G = \mathcal{P}Capability$. A process requests some operation on an object which can be represented as a capability – a function mapping an object to the operations which are requested on it.

The idea of process state transformation is modelled by the reflexivity of the domain definition. We are not interested in the concrete structure and state of objects or operations. The notion of state is implicit and abstract. This is an essential property of the model. The state of the system, i.e. the state of the objects, is not relevant for the access control model. Only the process of granting or denying access is relevant. Only the state change is important, but not the internals of the state. This objective was the motivation to choose a reflexive and abstract domain as the foundation of the dynamic model (this compares to continuations from denotational semantics [Sch86] where also a reflexive domain is used to model the remainder of a computation).

3.2 Mathematical Properties of the Domain

Mathematical properties of the access control domain shall now be addressed. The essential one is that a solution for the domain equation used to model our access control processes exists. In order to start this investigation, we need to add more structure to the underlying category. Reflexive domain equations cannot be solved in the category of sets and total functions. We need to consider sets S as posets $(S, \leq)$ in which every chain has a supremum. This forms the category CPO of ω-complete partial orders as objects and ω-continuous functions as arrows, see [Pie91] Chapter 3.4 or [BW96] Chapter 2.4 for details.

Theorem 3.1 *The recursive domain (a domain is an ω-complete partial ordering – cpo)*

$$P = ((R \to P) \times G)_\perp$$

has a solution.

Proof: Bolignano and Debabi show in [BD92] a proof for the existence of $P = ((E \to P) \times \mathcal{P}\mathcal{P}E)_\perp$ with the constraint $dom(m) = \bigcup S$ for all $(m, S) \in P$. In our case, we have the domain $P = ((R \to P) \times G)_\perp$ and the condition $dom(\mathit{pfm}) \in g$ for $dom(\mathit{pfm}) : R$ and $g : G$ with $R = Capability$ and $G = \mathcal{P}Capability$. The proof idea of [BD92] Section 5 can be applied: the subset relation between E and $\mathcal{P}\mathcal{P}E$ is not used in the proof and our constraint is less restricted than that of [BD92]. In order to show the existence of a solution, we need to show that a suitable P_∞ is isomorphic to $F(P_\infty)$. We set up a hierarchy of domains $P_n (n \geq 0)$ where P_0 is the trivial domain $\{\perp\}$ and $P_{n+1} = F(P_n)$ with $P_{n+1} = ((R \to P_n) \times G)$ in this

case. We define an order $\sqsubseteq$ on P_i: $\forall P \in P_i \,.\, \bot \sqsubseteq_{P_i} P$ and $((R \to P) \times G) \sqsubseteq_{P_i} (R' \to P') \times G')$ if $R \subseteq R'$ and $G \subseteq G'$. Bolignano and Debabi show that $(P, \sqsubseteq)$ is an algebraic cpo. □

More details about properties of the domain can be found in [Hen85, BD92]. The domain is a Scott-domain, if we assume that the posets R and G are countable. As mentioned in the proof, Bolignano and Debabi used a domain construction similar to ours as a denotational model for a process algebra. Their approach will be discussed in Section 6.

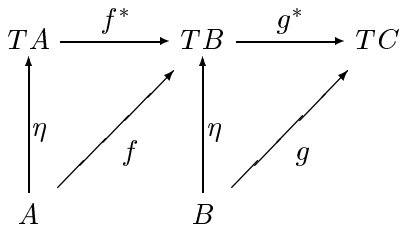
4 Dynamic Aspects – The Access Control Monad

A monad shall be used to accommodate the access control domain. Monads are algebraic constructs which allow the integration of new language features into a functional specification or programming language. Here, our aim is to integrate access control into such a language.

4.1 Monads

Monads have been suggested in denotational language semantics as a construct to extend language definitions in a modular way [Mog91]. The potential of using monads also for practical issues in language design has been recognised [Wad92, Wad98, LH96, CS97]. Monads allow us to integrate new language features into a given language. Monads are defined in terms of category theory. We will present here monads in the category CPO of posets and continous functions[2].

We introduce two classical monads to illustrate how monads work before we present our own access control monad. Monads are triples $(T, \eta, \_^*)$ consisting of a type constructor T on sets, an embedding $\eta : A \to TA$ for a set A and a function lifting which lifts a function $f : A \to TB$ to $f^* : TA \to TB$.



The embedding η and the function lifting $\_^*$ allow us to have function composition on the level of lifted functions. The elements of a monad are constrained such that η is a unit of composition and that function composition on the level of liftings is associative. These constraints are called Kleisli laws.

The monad elements shall be illustrated by looking at how monads are used in language semantics. The fundamental difference between standard denotational semantics [Sch86] and the monadic approach [Mog91] is, that in the monadic approach functions mapping to so-called computations, $A \to TB$, are considered as the denotation of programs, instead of functions $A \to B$ on simple value domains. The denotation of a program, which maps values from a domain A into a domain B and which has side-effects on a state S, is a

[2] We use the term 'monad', but technically we work with Kleisli triples. Moggi [Mog91] shows the equivalence of both structures. Monads are purely defined in terms of functors and natural transformations, categorically the more elegant form. We see Kleisli triples as the less abstract algebraic notation, as more suitable for our purpose of integrating different algebraic techniques. We will use this notation throughout the paper, also implementing it later on in Haskell.

state transformer $(S \times B)^S$, i.e. $f : A \to TB$ with $TB = (S \times B)^S$. Exponentiation $(S \times B)^S$ denotes the space of total functions $S \to (S \times B)$. The denotation of such a program is a function which takes a given state from S as an argument and which produces a new state element from S and a value element from B. Two examples of monads in Kleisli notation, using Moggi's formulation [Mog91], are:

1. Partiality:
   - $TA = A_\perp$ where $\perp$ is the diverging computation, the undefinedness symbol $\perp$ is added to A.
   - η_A is the inclusion of A into $A_\perp$: elements from A are embedded into $A_\perp$.
   - If $f : A \to TB$, then $f^*(\perp) = \perp$ and $f^*(a) = f(a)$ for $a \in A$.
2. Side-effects:
   - $TA = (A \times S)^S$.
   - η_A maps an element a from A to a constant state transformer $\lambda s : s.\langle a, s\rangle$.
   - If $f : A \to TB$ and $c \in TA$, then $f^*(c) = \lambda s : S.(\text{let } \langle a, s'\rangle = c(s) \text{ in } f(a)(s'))$. The construct 'let ... in ...' is a binding construct which binds values to variables (a and s' in the above example) which can be used in the expression on the right ($f(a)(s')$ in the example). The lifted function takes a computation c – a state transformer – applies it to a given state and performs f in the intermediate state s'.

In the next subsection, we are going to accommodate the dynamic access control model, presented in the previous section, into a monad.

4.2 Accommodating Access Control

A Request Language. We are going to accommodate the access control mechanisms in a variant of the state transformer monad. Our monad definition shall be illustrated using a simple *request language*. The syntax of our language is:

$$Expr ::= \ 'Con'\ Int \mid 'Incr'\ Expr \mid 'Request'\ Op\ Obj\ Expr$$

Each expression represents one of the following elements:

- a constant value - prefixed by the token Con,
- a simple increment operation - prefixed by the token $Incr$ - which has been chosen as a sample internal computation not requesting access to some object,
- a request to an external object - prefixed by the token $Request$ - which is subject to permission.

The two operations of the language are an increment operation, which is not subject to any form of access control, and a request requesting an operation $op : Op$ on an object $obj : Obj$.

In Section 5, we will present a Haskell program which interprets expressions in this simple language based on the monadic construction.

Standard Interpretation. In order to illustrate the difference between a standard denotational semantics and monad-based semantics, we will firstly interpret the language in the static algebraic security model using a classical semantics. This also serves as a more detailled introduction to the request language.

Sets of objects $Object$ and integer values Int shall be assumed. Access operations shall not be modelled explicitly. The state of our system shall be described by a pair consisting of an object component and a value component. The object changes by performing access operations on it, the value component contains the last incremented integer value. Thus, the execution of expressions shall be interpreted as functions mapping to the domain $Object \times Int$. Projections $\pi_1 : Object \times Int \to Object$ and $\pi_2 : Object \times Int \to Int$ are used to project onto the first or second component, respectively.

Let **P** be the semantic function

$$\mathbf{P} : Expr \to (Object \times Int)$$

which maps syntactic phrases from $Expr$ to the pair of objects and values. Values can be defined by

$$\mathbf{P}[\![' Con' \; val]\!] := \langle \bot, val \rangle$$

The undefinedness symbol $\bot$ expresses that no object is defined yet. This expression only initialises the an integer value.

$$\mathbf{P}[\![' Incr' \; t]\!] := \langle \pi_1 \mathbf{P}[\![t]\!], \pi_2 \mathbf{P}[\![t]\!] + 1 \rangle$$

This adds 1 to the value component and leaves the object component unchanged.

$$\mathbf{P}[\![' Request' \; obj \; op \; t]\!] := \langle op(obj), \pi_2 \mathbf{P}[\![t]\!] \rangle$$

With the expression $op(obj)$ we express that the operation op is performed on object obj. The value component remains unchanged. We can refine the definitions by including the operation $checkCapability$ in order to check permissions:

$$\mathbf{P}[\![' Request' \; obj \; op \; t]\!] := \text{if } checkCapability(\textit{self}, obj, op) \text{ then} \langle op(obj), \pi_2 \mathbf{P}[\![t]\!] \rangle$$

The operation op is only performed on object obj, if the permission is given. *self* is a pointer to the requesting subject itself.

The Access Control Monad. The *access control monad* M is an embedding of the process domain $P = ((R \to P) \times G)_\bot$ into the state transformer monad. The state transformer monad has already been introduced. Thus, we define the triple $M = (T, \eta, \_^*)$ by :

- $TA = (P \times A)^P$ with $P = ((R \to P) \times G)_\bot$. Values A are mapped to process transformers $(P \times A)^P$.
- η_A is defined as the mapping $a \mapsto (\lambda p : P.\langle p, a \rangle)$. The identity on processes is the basis of the definition of the embedding η.

- if $f : A \to TB$ and $c \in TA$, then $f^*(c) = \lambda p : P.(\text{let } \langle p', a\rangle = c(p) \text{ in } f(a)(p'))$. The current state of the system is computed on the level of computations by $c(p)$. This results in a value and a process description. Based on value a and process p', a new pair of value and process is computed by $f(a)(p')$.

One important question has to be addressed: does our definition for M form a monad? The question is easy to answer. The access control monad is obviously a monad, since it is directly based on the state-transformer monad: the abstract state domain is instantiated by the process domain, the rest is renaming.

Details of the static access control model are not relevant for the monad specification itself. It provides an abstract framework. The domains R and G can be instantiated in different ways in order to integrate different access control models. We can connect the abstract access control monad, which allows us to integrate security aspects into a functional framework, more closely to the static access control structures which were modelled in Section 2. The domain G represents guards which guard access to the requested object, modelled by a request $r \in R$. The guard G implements the specified security policy. G and R can be made concrete by defining $G := \mathcal{P}Capability$ and $R := Capability$ with $Capability := Object \to AccessOperation$. We can now reformulate $P = ((Capability \to P) \times \mathcal{P}Capability)_\perp$.

Monad-based Interpretation. Now, we have to connect the request language and the access control monad. We define the phrases of the language based on the given semantical model of access control monads.

In a first step, there shall be no guarding and no explicit objects. We only look at the semantics side – the semantic function **P** mapping syntax to semantics will be used later on. Let $a : A$ and $f : A \to TB$ with $TB = (P \times B)^P$ an element of set A and a function whose range is a computation, respectively. A function application can be defined in general as follows:

$$f(a) = \lambda p : P \,.\, \langle p, g(a)\rangle$$

i.e. $f = \eta \circ g$ holds. The function g is some computation. An example is the increment operator:

$$incr(a) = \lambda p : P \,.\, \langle p, a + 1\rangle$$

where a is an integer. Requests $r(a)(p)$ update the process description according to the access operation on the object. Values are not affected. The request r maps from A to TB. The projection $\pi_1(p)(r)$ denotes the subsequent process state.

$$r(a) := \lambda p : P \,.\, \langle\langle \lambda r' : R.\pi_1(p)(r), \pi_2(p)\rangle, a\rangle$$

The dependency on permissions for requests shall now be introduced. A simple request $'Request'\ obj\ op$ can be interpreted by the semantic function **P** as a mapping from A to TB.

$$\mathbf{P}[\![{}'Request'\ obj\ op\ t]\!] = \\ \text{if } checkCapability(\mathit{self}, obj, op) \text{ then } \lambda p : P \,.\, \langle\langle \lambda r : R \,.\, \pi_1(p)(obj, op), \pi_2(p)\rangle, \mathbf{P}[\![t]\!]\rangle$$

In the concrete access control model, requests $r : R$ are pairs $r = (obj, op)$ of objects and operations. The security constraint based on the domains G and R is satisfied by this interpretation: for $g \in G$ and

pfm $\in R \to P$, we have defined the security constraint as $dom(pfm) \in g$, saying that requests $dom(pfm)$ have to be permitted by the guard g. The guard realised here is based on capabilities (or an access control list) which maps objects to the access operation on them which are permitted. The operation $checkCapability$ implements a specific form of an access control list which distinguishes between trusted and non-trusted requesting subjects, referred to by the *self*-pointer.

This was only a sketch how the request language can be interpreted in the monadic structure. A full Haskell implementation of an interpreter for the language which interprets in the monadic structure is presented in the next section.

5 The Access Control Monad in Haskell

In the previous sections, we have shown how static and dynamic access control features can be modelled, and how monads can be used to integrate these features into a functional language. An implementation of the security monad in the functional programming language Haskell shall now be presented. We will follow Wadler [Wad92] and implement an interpreter for our simple request language from Section 4 based on the access control monad. The source code given in this section in executable in Haskell. We have chosen to follow Wadler's classical paper (in which he presents several interpreter implementation) in our notation and not to use the Haskell monad type class.

We are going to look at the language, the monad, and the interpreter in separate subsections.

5.1 Implementing the Language.

The language has already been introduced in detail in the standard interpretation part of Section 4.2. The syntax can be represented as follows:

```
data Expr = Con Int | Incr Expr | Request Obj Op Expr
```

`Con, Incr` and `Request` are syntactical keywords. `Int` denotes integer literals. `Obj` and `Op` denote objects and operations, both will be represented as strings. Expressions of this language are interpreted by executing them using the security monad.

Sample phrases of the language, which could be executed, are:

```
execute(Incr(Request "o1" "a2" (Request "o2" "a1" (Incr(Con 3)))))
```

or

```
execute(Incr(Request "o1" "a1" (Incr(Con 1))))
```

The execution of both depends on the current security policy.

5.2 Implementing the Monad.

The implementation of monad shall now be described.

```
type Obj = String
type Op = String
type Req = (Obj,Op)
type Guard = [(Obj,Op)]
```

These are basic type definitions. The next definition concerns the implementation of the monad type constructor. The first definition implements the reflexive process domain.

```
data R a  = BOT | R ( (Req -> R a) , Guard)
type RR a = R Int -> (R Int , a)
```

The type constructor `RR` maps a value type `a` to the process transformer type. Each process description is characterised by the type constructor `R`. `BOT` is a parameterless data constructor. `BOT` denotes the undefined process. The mapping η from the monad shall be implemented by a unit-operator.

```
unitR a = \p -> (p,a)
```

We have not implemented a bind-operator explicitly which corresponds to the lifting operator $\_^*$. Instead, we have expanded a definition whenever this bind-operator was supposed to be applied. The monad definition is now implemented, and we can define the interpreter. The syntax, in form of expression definitions, has already been described.

```
data Value = Num Int | securityError
```

The connection to our access control specification can now be seen.

```
checkCapability :: (Req,Guard) -> Bool
checkCapability(r,[])  = False
checkCapability(r,x:g) =
         if r==x then True else checkCapability(x,g)
```

A guard `g` implementing a sample security policy shall be provided.

```
g :: Guard
g = [("o1","a1"),("o2","a2")]
```

Object `o1` and be accessed through operation `a1` and object `o2` can be accessed through operation `a2`. The function `showval` implements an output routine which prints the computed value.

```
showval :: Value -> Int
showval (Num i) = i
```

We use `add` to realise the increment operation of the language.

```
add :: Value -> Value -> Value
add (Num i) (Num j) = (Num (i+j))
```

5.3 Implementing the Interpreter

The main interpreter function shall now be defined.

```
interp :: Expr ->  RR Value
interp (Con i)  = unitR (Num i)
interp (Incr t) = \p0 -> let (p1,a) = (interp t) p0
                         in  (p1,(add a (Num 1)))
```

The process description `p1` is returned without modifications. We increment by adding `1` to the value in `a`. In the definitions of `interp (Incr t)` and `interp (Request obj op t)` we have used the bind-operator implicitly. The correspondence is still visible.

```
interp (Request obj op t) = \p0 ->
          let (R(p1,p2),a) = (interp t) p0
          in  if checkCapability((obj,op),g) then
                        (p1(obj,op),a)
              else
                        (R(p1,p2),securityError)
```

The value of the previous state is just passed through. The subsequent process state is determined by the result of the expression `p1(obj,op)`. `p1` is the implementation of an element of $R \to P$. This mapping is then applied to the actual request `(obj,op)`. In case the request is denied, a `securityError` will be produced. There is no exception handling implemented in this simple version, the program will react to security errors with a run-time error in the next operation.

5.4 Implementing the Interface

The interface is implemented by the `execute` function. It will print out the final value. The function needs an initial process description which is supposed to be executed.

```
execute :: Expr -> Int
execute (t) = let (p,a) = (interp t)  (R((\s -> BOT),g))
              in  showval a
```

The initial process `R((\s -> BOT),g)` terminates after the first process state transition. An alternative call of `execute` would be:

```
execute (t) = let (p,a) = (interp t)  BOT
              in  showval a
```

This, viz. `BOT`, terminates immediately.

5.5 Correctness of the Implementation

We will not prove formally that our Haskell implementation indeed implements the static and dynamic access control specifications correctly. It is relatively easy to see the correspondence between specification and implementation.

6 Related Work

At the heart of our security model is the dynamic access control model and its embedding in a monad. The domain underlying the dynamic access control model, which we have used here, is similar to a denotational model for process algebras such as CCS [Mil89] or CSP [Hoa85]. In [BD92] the denotational model was used to define the concurrency sublanguage of the RAISE specification language. We have based our model on this idea since one of our objectives is to model an abstract space of processes – the requests and their execution, which compares to abstract processes and events. The standard model for processes based on acceptance sets is $P = ((E \to P) \times \mathcal{P}\mathcal{P}E)_\perp$ (see [BD92]). The domain of process descriptions P is recursively defined. E denotes a set of events. Acceptable events are determined by acceptance sets of type $\mathcal{P}\mathcal{P}E$. Matching between acceptable and actual events is therefore evaluated based on set membership. In our security application, the matching is determined by the guard and the security constraint.

Monad constructions to describe the semantics of languages and to integrate features into a language in a modular way have been widely used [Mog91, Wad92, Wad98]. Another application of Moggi's monadic type calculus is presented in [ABHR99]. Abadi et.al. present the integration of several dependency analysis features in a monadic lambda calculus. Their calculus, called the Dependency Core Calculus DCC, is an extension of Moggi monadic calculus [Mog91, Mog97]. The calculus is presented with a denotational semantics. Typed calculi for secure information flow, partial evaluation, program slicing and call-tracking are translated into DCC. These different aspects are subsumed under the notion of *program dependency*. The idea of integration the various forms of program dependency analyses is essentially motivated by the security information flow analysis (as described in [HR98]).

Heintze and Riecke [HR98] present a typed lambda calculus that deals with type and security information. The security aspects addressed are confidentiality (no unintentional disclosure of sensitive data) and integrity (no unintentional modification of sensitive data) with the aim of preventing security violations. There mechanism work on tracing dependencies between input and output of operations in order to guarantee secure information flow. Information flow analysis is the basic tool. The security level of data is monitored as it flows through a system. This corresponds to our approach. We distinguish two security levels: trusted and untrusted. Requests of untrusted subjects might not be granted and the flow of sensitive data is prevented. We take another viewpoint in that we focus on access control and not the objects (or data) themselves. Whereas Heintze and Riecke only allow a security classification of data, we offer a more fine-grained model considering the form of access and the subject's requests as well in the security policy.

Smith and Volpano [SV98] analyse secure information flow in imperative while-programs. Data is labelled as either *high security* or *low security*. Mechanisms based on the type system are offered which guarantee that an operation demanding secure input results in a secure output and that low-security output does not depend on high-security input. This is called non-interference property. It expresses that high-security data does not interfere with the calculation of low-security data.

7 Conclusions

Algebra is a suitable notation for formulating concepts of access control. We have presented an abstract model of an access control system consisting of a security policy and a policy enforcement part. We have especially looked at dynamic aspects, i.e. seeing subjects in the security system as processes which request operations on objects. We have used a monadic meta-language to accommodate an abstract domain of these processes. The monad allows us to integrate access control into a functional language in a modular way. Moggi's monadic type calculus has proved itself as a suitable framework to accommodate various computational effects such as side effects [Mog91, Wad92]. Now, it also appears to be suitable to incorporate program dependency analyses such as security analyses, which should prevent security violations, but do not have computational effects [Mog97, ABHR99].

The process domain and the monad based on this domain possess useful mathematical properties. In addition, the concepts are of practical relevance since they can easily be integrated into a specification or programming language. The approach of using algebra to model static security features and monads to integrate language features into languages is well known. We have presented a new abstract model for the dynamics of access control based on a reflexive domain, in which statics can be integrated. The accommodation of this dynamics model into a monad allows us to integrate this model into different contexts.

The model can be used for the development of security systems and for the development of languages with security features. Monads and in particular the access control monad can be used to implement algebraically specified security services in a functional language. In the area of language design, our concepts can in particular be used to support extension and evolution of languages. The access control monad is generic. Different concrete access control mechanisms can be realised within the framework given by the monad. In the design of a language dialect, a number of possible realisations of a particular feature can be explored and experimented with.

We have used ideas from denotational models for process algebras: we have adapted the domain and realised one typical operation, the event-based process state transition. The event in our model is the request. Using such a model raises the question whether process combinators such as the parallel composition or non-deterministic choice operators from process algebras can be realised in this framework. This would allow us to implement concurrent processes requesting objects in a multi-user environment. Requests could be directed to a policy process which guards the objects by checking the capabilities of requesting subjects. The answer to this question is based on the ability to implement independent processes in our implementation language Haskell. The Haskell monad classes could be used for this purpose. Such a solution would allow us to implement a flexible multi-user security system in which permissions can be passed between processes. Another direction in which future research can be directed is to investigate type-based calculi for access control and analysis techniques for the detection and prevention of security violations.

We have used the term *algebra* to describe our framework for specifying, integrating and reasoning about security systems. To be more precise, we have partly used co-algebraic methods instead of classical algebraic ones [JR97]. The use of monads based on functions $X \to TX$ indicates this. The difference between algebras and co-algebras is the difference between construction and observation, respectively. A function of type $X \to TX$ maps from X into some observation domain on X. During the last decade, dynamic state-based systems, such as or security systems, have more and more been described using co-algebraic methods.

Acknowledgements.

Many thanks to go the anonymous referees for their constructive and helpful comments.

References

[ABHR99] M. Abadi, A. Banerjee, N. Heintze, and J.G. Riecke. A Core Calculus of Dependency. In ACM, editor, *26th Symposium on Principles of Programming Languages POPL'99*. ACM Press, 1999.

[Amo94] E. Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall, 1994.

[BD92] D. Bolignano and M. Debabi. On the Foundations of the RAISE Specification Language Semantics. Technical report, ESPRIT project LACOS, 1992.

[BW96] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1996. (second edition).

[CS97] P. Cenciarelli and E. Saaman. Using Monads in Algebraic Specification. In *2th Workshop on Algebraic Development Technology (WADT97), Tarquinia*, 1997.

[Fla99] D. Flanagan. *Java in a Nutshell - 3rd Edition*. O'Reilly & Associates, 1999.

[Gol99] D. Gollmann. *Computer Security*. John Wiley and Sons, 1999.

[Hen85] M. Hennessy. Acceptance Trees. *Journal of the ACM*, 32(4):896–928, October 1985.

[HJW+92] P. Hudak, S.L. Peyton Jones, P.L. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, M. Guzman, K. Hammond, J. Hughes, T. Johnsson, R. Kieburtz, R.S. Nikhil, W. Partain, and J. Peterson. *Report on the Functional Programming Language Haskell*. ACM SIGPLAN Notices, 1992.

[Hoa85] C.A.R Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HR98] N. Heintze and J.G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *25th ACM Symposium on Principles of Programming Languages POPL'98*. ACM Press, 1998.

[JR97] B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin*, 62:222–259, 1997.

[LH96] S. Liang and P. Hudak. Modular Denotational Semantics for Compiler Construction. In H.R. Nielson, editor, *Proceedings European Symposium on Programming ESOP'96*. Springer-Verlag, LNCS 1058, 1996.

[MF99] G. McGraw and E. Felten. *Securing Java*. John Wiley and Sons, 1999.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mog91] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93:55–92, 1991.

[Mog97] E. Moggi. A Categorical Account of Two-level Languages. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Proceedings Mathematical Foundations of Programming Semantics MFPS'97*. Electronic Notes in Theoretical Computer Science, Elsevier, The Netherlands, 1997.

[Pfl97] C.P. Pfleeger. *Security in Computing - 2nd Edition*. Prentice Hall, 1997.

[Pie91] B.C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[Sch86] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Wm.C. Brown Publishers, 1986.

[SV98] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *25th ACM Symposium on Principles of Programming Languages POPL'98*. ACM Press, 1998.

[Wad92] P. Wadler. The Essence of Functional Programming. In *Proc. 19th ACM Symp. on Principles of Programming Languages, Austin, Texas*, 1992. (invited talk).

[Wad98] P. Wadler. The Marriage of Effects and Monads. In ACM, editor, *Proc. of the 1998 ACM Conference on Functional Programming*, pages 63–74. ACM Press, 1998.