21

Developing Secure Applications: A Systematic Approach

C. Eckert and D. Marek
Munich University of Technology, Department of Computer Science
D–80290 Munich, Germany, eckertc@informatik.tu-muenchen.de

Abstract

This paper presents parts of the SECREDS project which aims to bridge the gap between system modeling and implementation using a high-level programming language. Within SECREDS secure applications are developed top down starting with a top-level specification. Top-level specifications are given by our computational model and application-specific security policies are specified using our security requirement logic. To implement a top-level specification we developed a high-level programming language called INSEL$^+$ offering language concepts well adapted to our underlying model. We will present main features of INSEL$^+$ focusing on access control aspects and we will outline some guidelines to support the systematic implementation of a given top-level specification preserving specified security properties.

Keywords

Access Control, Security Policy, Programming Language

1 INTRODUCTION

The issue of developing secure applications is still a great challenge. Secure applications should be developed top-down starting with a formal top-level specification given by a security model comprising the security policy of the application. A lot of security models have been proposed in the literature focusing on confidentiality (e.g. Bell, 1975) or integrity (e.g. Clark, 1987) aspects. Besides their individual shortcomings existing approaches lack appropriate support to bridge the gap between system modeling and implementation using a high-level programming language. Hence, a framework is required offering features to model the behavior of a distributed secure application on a high level of abstraction as well as features to specify access properties as well as information flow properties adapted to the specific needs of the application. In addition, a high-level programming language is required offering language concepts adapted to the formalism used for top-level specifications.

This adaption enables to systematically transform a top-level specification into an executable program.

The paper presents parts of the SECREDS project which aims to bridge the gap between the formal specification of secure parallel and distributed applications and their implementation and execution in a distributed environment.

The rest of the paper will be organized as follows. In section 2 we give a short overview over the SECREDS project. Section 3 briefly introduces our computational model and the logic to specify application-specific security policies. Section 4 presents main features of the language INSEL$^+$. The development of INSEL$^+$ programs starting with a top-level specification is explained by means of an example. Section 5 concludes the paper.

2 SECREDS – AN OVERVIEW

Within SECREDS a framework to design and implement secure distributed applications is elaborated. An overview over the working areas of the project is sketched in figure 1. The paper focuses on the dotted parts.

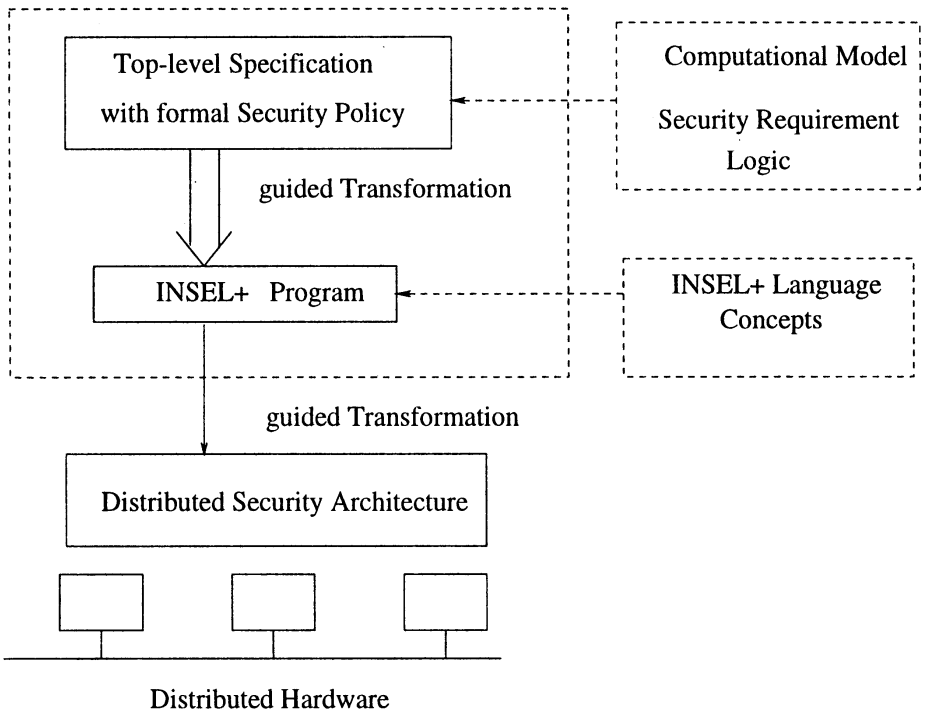

Figure 1 Overview over the SECREDS approach

In SECREDS the development of a secure distributed application starts with a top-level specification comprising a semantic model of application's behavior together with those attributes needed to capture security properties.

These properties (the security policy) are specified by allowed or disallowed information flows between users, as well as, access restrictions for users defined with application-specific granularity. The security properties are given by means of formulas of our security requirement logic.

A top-level specification is implemented by stepwise refinement. First, the specification is transformed into a program using the language concepts of our high-level programming language INSEL$^+$. Though desirable we are not able to perform a verified transformation from top-level specification to implementation within SECREDS. Formal verification requires great efforts (e.g. EHDM (Rushby, 1991)). Within SECREDS we pursue a less sound but more pragmatic approach. Instead of elaborating formal transformation rules we developed a programming language which offers language concepts that allow to express security properties in a declarative way. As the language concepts and the formalism to specify security policies are very closely related, major parts of the specification can be directly implemented using the adapted language concepts. In addition, we have elaborated guidelines which aid the application programmer in transforming a top-level specification into an INSEL$^+$ program.

INSEL$^+$ applications are executed in a distributed environment based on a tailored security architecture. The security architecture is part of the MoDiS distributed operating system (cf Eckert, 1996). Describing this architecture lies beyond the scope of this paper. The key feature of MoDiS is its distributed manager architecture. All resource management tasks including security services are cooperatively accomplished by a distributed reflective manager architecture. As INSEL$^+$ programs are realized by stepwise refinement it is the task of these managers to perform security services like access controls, authentication and encrypting messages sent across a network.

3 TOP-LEVEL SPECIFICATION

3.1 Computational Model

In this subsection we introduce the main features of our computational model to describe system behavior. A more detailed and formal description of the model can be found in (Eckert, 1995).

A distributed application is modeled by a set of **objects** and **subjects**. Objects are the protected entities which can be specified with fine-grained granularity. Objects are only accessible via well-formed **operations** comparable with well-formed transactions in the Clark-Wilson model (Clark, 1987). Operations define access rights for objects. Each operation invocation creates a distinct **instance** of the operation. The computational steps of an instance are defined by a sequence of **atomic actions** called action refinement of the instance. An action refinement starts with an initial action. During the exe-

cution of the initial action the input parameter association is performed. The last action of the refinement is the termination action which associates the output parameters if necessary. Executing an atomic action results in state changes mirroring the **effect** of the action.

Each user is represented by a set of subjects the so called **user representatives**, which execute operations on behalf of users. **Concurrency** between user activities is modeled by the interleaved execution of sequences of actions belonging to different action refinements of operations executed by the users' representatives. The sequence of states associated with a sequence of actions is called a **computation** σ of the model CS of a distributed application.

We introduce a **generic set of labels** and a labeling function to capture important object properties within our model. For instance, labeling associates a **role** with each user representative describing the role its associated user is playing. Introducing role-attributes allows us to grant access rights to roles rather than to individual users if required.

The computational model can be used to define the formal semantics of a programming language. Simple read and write operations on data objects of predefined types, like integer or boolean objects, can be modeled as atomic actions.

3.2 Policy Specification

User representatives execute operations on behalf of their associated users. Each execution of an instance may cause information flows between different user representatives and, therefore, between users. To describe these information flows we introduce two properties: the **influence** property and the **observation** property. The semantics of these properties (cf Eckert, 1995) is defined based on our computational model.

Definition: Influence
We say, that there exists an **influence between user representatives** ur_1 and ur_2 with respect to operations op_1 and op_2 in a given computation σ if the action refinement of at least one instance of op_1 executed by ur_1 contains an action a which influences the execution of an action b contained in the action refinement of at least one instance of op_2 executed by ur_2.

The execution of an action a **influences the execution of an action** b in computation σ if either executing b after a in computation σ provides different values for at least one object y compared with the value of y in a computation σ' where b is executed before a or a influences b if b is not enabled until a has been executed. △

Restricting information flows based on allowed and forbidden influences turns out to be very restrictive as many applications just require that the information flows caused by influences may not be observable by other users.

Definition: Observation
We say, the activity of a user representative ur_1 executing instances of op_1 is **observable by a representative** ur_2 executing instances of operation op_2, if the value of at least one output parameter of at least one instance of op_2 executed by ur_2 depends on the effect of the execution of at least one instance of op_1 executed by ur_1. That is, observable information flows can be recognized via different values of output parameters of instances. $\triangle$

To specify disallowed influences and observations between users of an application we will define **conditional non-influencing** (*cninf*) requirements and **conditional non-observing** (*cnobs*) requirements. Disallowed and allowed accesses are specified by **access restriction** (*acc*) requirements.

Definition: Security requirement logic
Given a computational model CS of an application. The set of objects $\mathcal{K}$ of CS, the set of predicate symbols PS with $PS = PS' \cup \{cninf, cnobs, acc\}$, and the temporal operator $\Box$ expressing the usual 'henceforth' semantics form the syntactic basis of the logic. Let TFO be the set of all formulas over the basis. PS' contains predicates over $\mathcal{K}$ comprising logical and arithmetic operators and quantifiers.

Let $<$ define the linear ordering on states in a computation σ of CS.
$\omega : TFO \times \Sigma \longrightarrow \{true, false\}$ defines the semantics of formulas in a state.

Given two user representatives ur_1, ur_2, two operations op_1, op_2, an object k, a computation σ of CS and a predicate $Cond \in PS'$. The semantics of a formula $P \in TFO$ in state s of computation σ, denoted by $(\sigma, s) \models P$, is defined as follows*:

1. Let $P = cninf(ur_1, ur_2, op_1, op_2, Cond)$.
   $(\sigma, s) \models P$ iff $\omega(P, s) = true$ whereby the **conditional non-influencing predicate** *cninf* is *true* in state s of the given computation, if either predicate *Cond* is $false$ in state s or predicate *Cond* is *true* in state s and for each state s' with $s' < s$ holds, that no influence caused by user representative ur_1 executing instances of op_1 on user representative ur_2 executing instances of op_2 exists.
2. Let $P = cnobs(ur_2, ur_1, op_2, op_1, Cond)$.
   $(\sigma, s) \models P$ iff $\omega(P, s) = true$ whereby the **conditional non-observing predicate** *cnobs* is *true* in state s, if either predicate *Cond* is $false$ in state s or predicate *Cond* is *true* in state s and for each state s' with $s' < s$ holds, that no observation of the activities of user representative ur_2 executing instances of op_2 by user representative ur_1 executing instances of op_1 exists.
3. Let $P = acc(ur_1, op_1, k, Cond)$.
   $(\sigma, s) \models P$ iff $\omega(P, s) = true$ whereby the **access restriction predicate** *acc* is *true* in state s, if either predicate *Cond* is *true* in state s or predicate

*Due to space limitation we focus on the *cninf*, *cnobs* and *acc* formula.

Cond is *false* in state s and user representative ur_1 does not execute an instance of op_1 in state s.

4. Given a formula $P \in TFO$:
   $(\sigma, s) \models \Box P : \Longleftrightarrow \ \forall\, s' \in \sigma \ : \ s' \leq s \ : \ \omega(P, s') = true.$ $\triangle$

A conditional non-influencing predicate between two user representatives ur_1 and ur_2 with respect to two operations requires that, as long as the condition *Cond* within the requirement specification holds, no influence from ur_1 on ur_2 executing instances of the specified operations exists. The meaning of a conditional non-observing requirement can be stated in a similar way. With an access restriction predicate the execution of operation op_1 of the specified object k is forbidden for user representative ur_1 as long as the condition *Cond* within the requirement is *false*.

The condition *Cond* may, for instance, contain a boolean expression concerning access rights or it may contain an expression concerning other objects of the system, for instance a timer to restrict access for a user to a limited period of time.

Based on the security requirement logic introduced a **security policy** for a secure parallel application is specified by a formula $P \in TFO$. Usually, a security policy is specified by a conjunction of conditional non-influencing, conditional non-observing and of access restriction predicates which must hold in every state of a computation, hence by a conjunction of $\Box acc(...)$, $\Box cninf(...)$, $\Box cnobs(...)$ formulas. As a computation of the model is given by a sequence of states associated with action executions the security policy P must hold in every state of the computation.

3.3 Example: Bank Scenario

Consider for example a simplified bank scenario. The set of protected objects comprises the customer accounts. For each account the set of operations is given by $OP(account) \supset \{read\_OP, enter\_rights\_OP, delete\_rights\_OP\}$. To distinguish between the operation itself and the access right we introduce the set of rights $\{read, enter\_rights, delete\_rights\}$. Users can act in different roles. Depending on the role a user currently plays a different set of access rights is granted. To keep the example simple we introduce only two roles: `customer` and `clerk`. Subjects in this scenario are for instance user representatives for customers and clerks. With each account we associate a unique owner and a clerk responsible for managing the account.

To manage access rights we introduce an object `access_matrix` M. M is a two-dimensional array, where columns are given by bank accounts and rows are given by users and roles and where each entry describes the set of access rights a user or role possesses with respect to a specific account (usual access matrix approach). Notice, that in our model possession of an access right is

usually not a sufficient precondition to gain access permissions. M is itself a protected object which must be accessed in a protected and controlled way. $OP(M) = \{enter\_OP, delete\_OP, create\_account\_OP, delete\_account\_OP\}$. The access rights are given with $\{enter, delete, create\_account, delete\_account\}$.

Security requirements:

1. A user who possesses the *read* right and acts in the role customer is allowed to execute the operation *read_OP* on a specific account. A clerk may just read the balance of those accounts he is responsible for and this read access is restricted to office hours.
2. Each customer may only grant (*enter_rights_OP*) or revoke (*delete_rights_OP*) access rights concerning accounts he owns. Granting or revoking access rights concerning owned accounts requires the possession of the specific access right (*enter_rights, delete_rights*) by the customer. A clerk may only grant and revoke access rights concerning accounts he manages. As a clerk may revoke access rights a clerk can freeze an account by revoking all rights for the account owner.
3. The operation *enter_OP* of the access matrix to grant rights can only be executed in the context of the operation *enter_rights_OP* of account objects and the caller must possess the *enter* right. (For all other matrix operations similar context-dependent restrictions must hold.)
4. Activities of customers who do not share accounts may not be visible to each other.
5. Activities of a customer may not influence the activities of a clerk if the clerk is not responsible for managing the accounts of the customer.

To formalize the requirements using the logic formulas we first introduce the labels we need. A label l is given by a tuple (*user*, *role*, *account_owner*, *resp_clerk*). For each user representative ur, $l(ur).user$ denotes the user ur represents. $l(ur).role$ denotes the role of the associated user. The owner of an account k is given with $l(k).account\_owner$ and $l(k).resp\_clerk$ denotes the clerk being responsible for the account or for the user representative, respectively.

Security requirement specification in terms of logic formulas:

1. Access restriction formula for *read_OP* operations.
   □ $acc(ur, read\_OP, account, Cond)$ with

$$\begin{aligned} Cond = \; & (read \in M[l(ur).user, account] \wedge \; l(ur).role = customer) \vee \\ & (l(ur).role = clerk \; \wedge \; l(account).resp\_clerk = l(ur).user \; \wedge \\ & 8 \leq CurrentTime \leq 16) \end{aligned}$$

2. Access restriction formula for *delete_rights_OP* operations.
   □ $acc(ur, delete\_rights\_OP, account, Cond)$ with

$$Cond = \begin{array}{l}(delete\_rights \in M[l(ur).user, account] \ \wedge \\ (l(ur).role = customer \ \wedge \ l(account).account\_owner = l(ur).user) \\ \vee \ (l(ur).role = clerk \ \wedge \ l(account).resp\_clerk = l(ur).user)\end{array}$$

Access restriction formula for *enter_rights_OP* operations can be specified in an analogous way.

3. Restricting matrix accesses to specific execution contexts:

$$\Box \ acc(ur, enter\_OP(account, \ldots), M, in(enter\_rights\_OP(account, \ldots), ur))$$

The predicate `in(operation_name, user_representative)` describes context-dependencies. The predicate is *true* in a state s of a computation if user representative `user_representative` executes an instance of operation `operation_name` in state s.

4. Information flow restrictions are specified by conditional non-observable predicates. To identify users who share accounts we define:

$$set\_of\_accounts(ur) = \{account| \ \begin{array}{l}M[l(ur).user, account] \neq \emptyset \ \vee \\ M[l(ur).role, account] \neq \emptyset\}.\end{array}$$

For all $op1, op2 \in OP(account)$ the predicate $\Box \ cnobs(ur1, ur2, op1, op2, Cond)$ must hold with

$$Cond = \begin{array}{l}l(ur1).role = customer \ \wedge \ l(ur2).role = customer \ \wedge \\ set\_of\_accounts(ur1) \cap set\_of\_accounts(ur2) = \emptyset\end{array}$$

5. No information flows are allowed between clerks and customers not being correlated.
For all $op1, op2 \in OP(account)$ the predicate $\Box \ cninf(ur1, ur2, op1, op2, Cond)$ must hold with

$$Cond = \begin{array}{l}l(ur1).role = customer \ \wedge l(ur2).role = clerk \ \wedge \\ l(ur1).resp\_clerk \neq l(ur2).user\end{array}$$

The security policy can be strengthened or weakened systematically by adding or removing information flow or access restrictions or by modifying the conditions within the requirement formulas.

4 IMPLEMENTATION

4.1 INSEL$^+$ Programming Language

Given a top-level specification we have to implement the specification using a programming language. Unfortunately, existing languages provide no or only insufficient languages features to support the implementation of security policies. Hence, application programmers have to deal with low-level security

mechanisms (cf (Ancilotti, 1983, McGraw, 1979)) and security services offered by the underlying operating system like simple access control lists for files.

To bridge the gap between specification and its implementation programming language concepts are needed, that allow to specify security properties in a declarative fashion as far as possible and that are well adapted to the formal framework used for security policy specification. Our programming language INSEL$^+$ offers the desired features.

INSEL$^+$ is a strongly-typed, object-based language for programming secure parallel and distributed applications. INSEL$^+$ provides concepts for passive and active objects. An active object defines a separate thread of control. Each object is an instance of an INSEL$^+$-class. A class defines a set of operations that provide the only means for accessing objects of that class. One important feature of INSEL$^+$ is the principle of nesting. The principle of nesting enables to restrict the scope of objects according to visibility rules known from block-oriented programming languages like Ada 95 (cf Feldman, 1996). That is, access to objects can be a priori restricted. INSEL$^+$-objects may cooperate and communicate calling operations on other objects.

The object model of INSEL$^+$ is closely related to the object model of our formal framework. With the adapted object model, the principle of nesting and the well-known properties of object-based languages (e.g. information hiding by encapsulation) INSEL$^+$ provides an appropriate basis for implementing secure applications.

Security-related language features

Until now concepts for implementing access control policies that are specified with access restriction formulas *acc* have been elaborated and incorporated in INSEL$^+$. Language features to support the implementation of information flow restrictions (*cninf, cnobs* predicates) are still under investigation. As a lot of these flow restrictions can be expressed by access control restrictions as well, just needing appropriate object labeling, we are already able to implement a wide range of information flow policies. For instance, information flow restrictions comparable to the well-known multi-level security policy MLS (Bell, 1975) can easily be transformed into access control restrictions by introducing security classifications and clearances for objects. Within the *Cond* condition of an access control formula *acc* conditions expressing the 'no write down' and 'no read up' property of MLS must be specified to restrict access according to the MLS policy.

To support the implementation of *acc* predicates INSEL$^+$ provides the concept of **access-controlled objects** and **classes**. For each operation of an access-controlled entity we are able to formulate a boolean expression called **access restriction expression** that is evaluated at runtime on each operation call before operation execution. If the access restriction expression is true for the calling object the requested access is allowed and the operation is executed, otherwise access is denied. To handle denial of accesses we integrated a simple exception handling mechanism in INSEL$^+$.

Access Restriction Expression
An access restriction expression defined for an operation of an access-controlled entity is a boolean expression that may contain: (1) local objects of the access controlled entity including input and output parameters of the operation; (2) global objects, (3) special predicates `IN_ACL` and `ACCESSED` and (4) the attribute `Caller`.

The attribute `Caller` is defined for each operation call and describes the calling object. It contains the uid (`Caller.UserUid`) of the user who created the object, the identifier of the role this user currently plays (`Caller.Role`), the identifier of the calling object (`Caller.Actor`) and the execution context of the calling object given by the identifier (`Caller.Con`) of the operation.

With the components of the `Caller` attribute the application programmer is able to express access restrictions concerning specific users, roles, active objects or contexts, in which the active objects acts. Further components, for example containing a security label of the calling object to support the implementation of label based security policies, can be added to the `Caller` attribute.

`IN_ACL` Predicate
For each access-controlled INSEL$^+$-object an access control list (ACL) is implicitly defined which may contain a list of subject identifiers for each operation (right) of the object. A subject identifier may be either a user identifier or a role identifier. The list of subject identifiers in an ACL entry for an operation may contain positive and negative subject entries (= negative right). The ACL of an access-controlled object is initialized on object creation. The initialization of the ACL is specified in a special part of the class description of the object. To dynamically change the entries of an object's ACL, the operation `ChangeACL_OP(...)` is implicitly defined on each access-controlled INSEL$^+$-object. With `ChangeACL_OP(...)` the associated ACL of the object may be altered.

`ACCESSED` Predicate
The predicate `ACCESSED` allows to check if a subject has already accessed an object via a specific operation. With the `ACCESSED` predicate the application programmer is able to specify restrictions depending on the access history of subjects.

The INSEL$^+$ runtime system provides a range of mechanisms for implementing access-controlled entities in a distributed environment. ACLs are implemented and managed in a secure way using low-level mechanisms offered by the underlying security architecture. Objects must be authenticated by using appropriate authentication mechanisms. As sketched in section 2, SECREDS aims to provide a security architecture tailored to our language concepts for implementing application-specific security policies.

4.2 Guided Transformation

In this subsection we roughly explain some guidelines to systematically transform a model and its associated security policy into an INSEL$^+$-program. The guided transformation is explained based on the previously introduced bank example.

Given the set of objects defined in the top-level specification we identify classes of objects which have the same functionality and the same security requirements. For example the set of customer accounts in the bank scenario forms such a class. For each identified class the application programmer has to implement an INSEL$^+$-class which defines the set of operations (rights) specified for the objects. If users can act in different roles, the set of initial roles has to be defined in a special role part in the main program. Each role requires the implementation of an INSEL$^+$-class specifying user representatives for users acting in this role.

The labels of objects defined by the generic set of labels in the model have to be implemented. Some application-independent labels, like the user which is associated with an object and the role this user plays, are directly supported by the attribute `Caller`. Application-specific labels, for example the label *account_owner* of a customer account in the bank scenario, have to be implemented by local variables of the object and have to be managed explicitly.

To implement the access restrictions specified by *acc* predicates the *Cond* conditions of these predicates must be transformed into corresponding access restriction expressions. Consider for instance the implementation of the customer accounts. Each account is an instance of the access-controlled INSEL$^+$-class `AccountType`. The labels *account_owner* and *resp_clerk* defined for an account are implemented by the input parameters `AccountOwner` and `RespClerk`, i.e. these labels are initialized on creation of a new account. The `access_matrix` M is implemented by the ACLs implicitly associated with each account. The operations *enter_rights_OP* and *delete_rights_OP* defined in the model are implemented in INSEL$^+$ by the predefined operation `ChangeACL_OP`. As an account's ACL may only be accessed via the `ChangeACL_OP` operation the policy restrictions concerning the access matrix are implicitly implemented. The initialization of the ACL of an account is specified in the ACL part of the class `AccountType`. For each operation of an account an access restriction expression implementing the access restrictions for accounts specified by the *acc* predicates is given in the access restriction part of the class. The following program skeleton specifies the class `AccountType`.

```
PROTECTED DEPOT TYPE AccountType(AccountNumber : IN integer;
                                 AccountOwner  : IN UserUidType;
                                 RespClerk     : IN UserUidType)  IS
 PROCEDURE TYPE Read_OP (Amount: OUT Real);  -- operations
        ...
```

```
ACL                                                    -- ACL  part
 Read_OP :  AccountOwner;                              -- initialization of ACL
      ...
ACCESS RESTRICTIONS                                    -- access restriction part
 Read_OP : (IN_ACL(Caller.UserUid,THIS,THIS) AND Caller.Role = Customer)
           OR (Caller.Role = Clerk AND Caller.UserUid = RespClerk AND
                 8 <= CurrentTime.Hour <= 16);
 ChangeACL_OP : (IN_ACL(Caller.UserUid,THIS,THIS) AND
                Caller.Role = Customer AND Caller.UserUid = AccountOwner)
                OR (Caller.Role = Clerk AND Caller.UserUid = RespClerk );
      ...
END AccountType;
```

THIS is the keyword for self-reference of an object or an operation. It strikes the eye that the gap between the *Cond* conditions of the *acc* predicates and the corresponding access restriction expressions is very small. Look for example at the *acc* predicate specified for the operation *read_OP*. The condition that an entry in the access matrix, $(read \in M[l(ur).user, account])$, has to exist for the calling user is implemented in the access restriction expression for Read_OP by the IN_ACL predicate IN_ACL(Caller.UserUid,THIS,THIS) and the condition that this user has to act in role *customer* ($l(ur).role = customer$) is implemented by Caller.Role = Customer.

The *acc* predicates for the operations *enter_rights_OP* and *delete_rights_OP* are combined and transformed into one access restriction expression specified for operation ChangeACL_OP.

As access and information flow restrictions may be specified for individual objects we are faced with the problem of specifying contradicting requirements. We have elaborated criteria to analyze access restriction formulas with respect to specific consistency properties. This analysis is incorporated into our INSEL$^+$ compiler. Discovering an inconsistency the compiler shows the two access restriction expressions and the kind of inconsistency caused by these expressions. Contradicting access restriction expressions then must be fixed by the application programmer. Hence, the programmer is offered support to strengthen or weaken parts of the security policy to gain an overall statically consistent policy as far as possible.

5 CONCLUSION

We have presented parts of our SECREDS framework to design and implement secure distributed applications. With our computational model fine-grained protected entities with access rights given by operations and application-specific user roles can easily be modeled. Our security requirement logic allows to specify fine-grained information flow restrictions, as well as, access restrictions customized to the individual needs of applications. As the SECREDS

approach aims to bridge the gap between formal specification and implementation we presented main features of our programming language INSEL$^+$ offering well adapted language support to implement the top-level specification of a secure distributed application in a systematic way. Guidelines have been developed to implement access restriction formulas using the INSEL$^+$ features. Some of these guidelines have been demonstrated by means of an example. The development and implementation of our tailored security architecture to realize secure applications in a distributed environment is still on going. Future work is concerned with enhancing INSEL$^+$, and our compiler as well as our security architecture with features to implement a wider range of information flow policies, that is to implement information flow formulas which can not be transformed into access restriction formulas.

The SECREDS approach combining formal specification techniques and attuned programming language concepts and tools supports the application programmer in developing secure applications of high quality.

6 REFERENCES

Feldman, Michael B. (1996) *Software Construction and Data Structures with Ada 95.* Addison Wesley

Ancilotti, P. and Bowi, M. and Lejmaer, N. (1983) Language Features for Access Control. *IEEE Transactions on Software Engineering*, SE-9(1).

Bell, D.E. and LaPadula L. (1975) *Secure Computer Systems: Unified Exposition and MULTICS Interpretation.* Technical Report MTR - 2997.

Clark, D.D. and Wilson, D.R. (1987) A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*,184 – 194.

Eckert, C. (1995) Matching Security Policies to Application Needs. In *11th International Conference on Information Security*, 237 – 254.

Eckert, C. (1996) Issues in the Design of Modern Distributed Computing Environments. In *Eighth IASTED International Conference on Parallel and Distributed Computing and Systems*, 188 – 192

McGraw, J.R. and Andrews, G.R. (1979) Access Control in Parallel Programs. *IEEE Transactions on Software Engineering*, SE-5(1), 1 – 9

Rushby, J.M. and von Henke, F. W. and Owre, S. (1991) *An Introduction to Formal Specification and Verification Using EHDM.* Technical Report, SRI International, Menlo Park.