Supporting Confidentiality in UML: A Profile for the Decentralized Label Model

Rogardt Heldal[1], Steffen Schlager[2], and Jakob Bende[1]

[1] Chalmers University of Technology
SE-412 96 Göteborg, Sweden
heldal@cs.chalmers.se
[2] University of Karlsruhe
Institute for Logic, Complexity and Deduction Systems
D-76128 Karlsruhe, Germany
schlager@ira.uka.de

Abstract We present a way of incorporating a decentralized label model into the UML by defining a profile which is the built-in extension mechanism of the UML. Our profile permits specifying confidentiality of data in UML by offering annotations for classes, attributes, operations, values of objects, and parameters of operations. The profile also supports generation of Jif (Java information flow) code and the Jif compiler guarantees that the specified confidentiality constraints are not violated. Our approach is appealing in the sense that it offers the possibility to consider confidentiality in UML and that the obtained code is guaranteed to behave correctly.

1 Introduction

Our philosophy is that it should be convenient to consider security issues during the system development process, and that security should be automatically verifiable at code level. Addressing both of these aspects is important in order to build a secure software system.

The UML [25] has become the *de facto* standard for the development of object-oriented software systems in industry. There are several reasons for this: it is relatively easy to understand and to learn and it offers several views on a system giving a good overview on its architecture. We aim to make it possible to consider security issues[1] during the industrial development process—so UML is a suited starting point.

One of the main problems with UML is that the focus has been more on functionality than on constraints such as security. In this paper we adapt the UML by defining a profile offering security annotations for a seamless integration of security aspects into a UML-based development process. But how can the

[1] In this work we concentrate on *confidentiality*. "Security" has a lot of other aspects, like e.g. authenticity, data integrity, non-repudiation, access control, or availability which are not considered here.

required rigor for handling security be obtained? Here, language-based checkers play an important role where security information is derived from a program written in a high-level language during the compilation process and is included in the compiled object. This extra security information can take several forms, e.g. a formal proof or type annotations. There have been several overview papers in this area, e.g. [17,27,26].

Our UML profile (addressing the specification of secure systems) is intended to be used together with a language-based checker to validate that the code really satisfies the security constraints. We decided to use the Java information flow (Jif) system [21,22]. Jif handles a large subset of the object-oriented[2] language Java [11] but also contains some additional language constructs, e.g. to control the propagation of confidential data. The Jif type checker guarantees that confidential data can only leak in a controlled manner. What kind of data is allowed to leak should already be part of the specification[3]. A prerequisite for that is, of course, that the specification language supports such constraints. It is important to notice that UML diagrams cannot be validated on the same level as code. The code is needed to consider for example indirect information flow [8] and covert channels [18].

In standard security models, like the Bell-LaPadual model [2] and the Biba model [4], the security policy is separated from the code. In this respect Jif differs since the policy is incorporated into the code in form of *labels*, implementing a *decentralized label model* [23,21]. Data is annotated with labels that specify the ownership and read permissions. The Jif type checker guarantees that the confidentiality policies declared in the labels are not violated.

The decentralized label model gives fine-grained control of data based on decentralized labels. So, it can be guaranteed that a program working with confidential data propagates information only in a controlled manner. Other approaches, like e.g. access control [3], give you all or nothing: they help to prevent information release but do not control information propagation, i.e. do not control how a program distributes confidential data that it is allowed to read. Also not suited for many applications is the sandbox model which e.g. is used for the execution of Java applets that can be downloaded from the internet. It prevents access to data outside the sandbox which is often too restrictive.

Java is not adequate for developing programs which require tight control of confidentiality. That is why we use Jif instead. Similarly, UML is not suited for specifying such programs. That is why we have previously created an extended version of UML called UMLS [12] (UML for Security). UMLS is also based on the decentralized label model. The purpose of that work was to demonstrate that the combination of model-based and language-based security is compelling. However, we did not extend UML in the standard way by defining a profile. This has serious drawbacks: UMLS is not UML-compliant, general UML tools cannot

[2] Object-orientation is important because the UML is tailored to the development of object-oriented systems.

[3] In fact, this information should already be identified during the analysis phase since the customer usually knows exactly which information has to be kept confidential.

be used and the interchangeability of models is harmed. The aim of this paper is to offer support for confidentiality in UML by casting UMLS in a UML profile.

In order to obtain an implementation from the model that satisfies the security constraints, Jif code skeletons can be generated automatically. A type checker then can automatically verify that the (manually) added implementation does not violate the specified confidentiality constraints. In this paper, we concentrate on class diagrams but the profile can be extended to the other diagram types considered in UMLS (interaction diagrams, use cases, and activity diagrams).

Structure of the paper. In Sect. 2 we shortly present the decentralized label model. The built-in extension mechanism of UML is introduced in Sect. 3. Our security profile UMLsProfile is defined in Sect. 4 where we also show some examples and discuss the profile. In Sect. 5 we mention related research before we draw conclusions and point out future work in Sect. 6.

Due to space restrictions we do not give an introduction to Jif. Rather, we introduce Jif bit by bit when needed. For more information on Jif, the reader is referred to [21,22].

2 Decentralized Label Model

The decentralized label model [23] is a security model that improves existing models by allowing users to declassify information in a decentralized way and by supporting fine-grained data sharing. Its main elements and ideas—labels, constraints, and declassification—are shortly explained in the following.

2.1 Labels

The central element of the decentralized label model is the *label*. Labels are used to annotate data in order to guarantee confidentiality—they specify ownership and read permission of data helping to control the propagation of (confidential) data.

Jif is an extension of the Java language [11] implementing the decentralized label model. In this paper we will incorporate the decentralized label model into UML, so UML can be used for deriving Jif code skeletons.

A label consists of a possibly empty set of *policies* where a policy consists of a list of *principals* (e.g. users, groups, or roles). Each policy has a dedicated principal as its *owner*. Each owner controls a set of *readers* that are allowed to read the data. By definition, an owner is implicitly contained in its reader set. A principal is allowed to read data if and only if it is contained in the reader set of all policies of the label attached to the data.

Example 1. The label $\{Bob : Lise\}$ consists of only one policy where the owner is Bob and the readers are Bob (the owner is always a reader) and Lise. The label $\{Bob : Lise, Lars; Lise :\}$ consists of two policies. Only Lise is allowed to read the data. She is the only one contained in the reader set of both policies.

$$Label = \{Components \mid \epsilon\}$$

$$Components = Component \mid Components\textbf{;}\ Component$$

$$Component = \textbf{\textit{principal:}}\ Principals \mid \textbf{\textit{identifier}} \mid \textbf{\textit{*identifier}} \mid \textbf{\textit{this}}$$

$$Principals = \textbf{\textit{principal}} \mid Principals\textbf{,}\ \textbf{\textit{principal}} \mid \epsilon$$

Figure 1. Syntax of Labels in BNF.

In Example 1 labels and principals are *static*. The advantage of static labels is that they can be checked at compile-time. Working only with static labels is however sometimes too restrictive. Thus, there is a need for two new primitive types *label* and *principal* and first-class values of these types represent labels and principals, respectively. We will see examples of how to use these types later.

Fig. 1 shows the syntax of label expressions where ϵ denotes the empty word and symbols in bold represent literals. As can be seen, a label may be empty (meaning that the data is not confidential) or consist of different components which we explain in turn.

A component can be a variable denoted by an identifier. Let us consider the following Jif code:

$$int\{Bob:\}\ x;$$
$$int\{x\}\ y;$$

Here the variable x is owned by Bob. In the label for y we have variable x, meaning that y has the same label as x—in this case $\{Bob:\}$. A component can also be a reference to a label. Let us consider the Jif code:

$$label\{Bob:\}\ lb;$$
$$int\{*lb\}\ y;$$

The label $\{*lb\}$ denotes the label stored in lb rather than the label of lb (which is denoted by $\{lb\}$ and would be $\{Bob:\}$ here). Finally, the reserved label $\{this\}$ represents the label of an object of the class.

Principals can be arranged in hierarchies where a principal can act for another principal ("A can act for B" means that A can do anything that B can do assuming his power). Jif contains an *actsFor* clause which executes a statement only if a certain constraint on the principal hierarchy is satisfied. There is also an *actsFor* constraint on methods which guarantees that certain defined hierarchies hold in the method body. The *actsFor* constraint will be contained in our profile. For more information on principal hierarchies see [21,22].

In Sect. 4 we will need the *join* of two labels, which is the least restrictive label that maintains all restrictions expressed by the two labels. Due to lack of space we omit a formal definition here (it can be found in [23]), we just give the following example.[4]

[4] Intuitively, the joined label is built from the union of the owners and the intersection of their reader sets.

Example 2. The join of labels $\{Bob : Lise\}$ and $\{Bob : Lise, Lars; Lise :\}$ from Example 1 is $\{Bob : Lise; Lise :\}$.

2.2 Declassification

Labeling of data guarantees that information does not leak to users without appropriate authority. Having only labels at hand is however often not sufficient. Sometimes it is necessary to consciously weaken the confidentiality of data, e.g. when an operation processes confidential data but the result should be made less confidential to permit the caller to use it. The problem is solved by giving *authority* (which consists of a list of principals) to classes and operations using the Jif keyword *where* (see example in Fig. 2). An operation must not be given more authority than its owning class[5]. Giving the authority $(p_1, \ldots, p_n)$ to a method means that the method can act on behalf of the principals p_i (even if the caller of the method has lower authority than p_i). This can be used for the *declassification* of data if the owner of the data is one of the principals $p_1, \ldots, p_n$. So, any principal $p_1, \ldots, p_n$ is allowed to relax its own policy (e.g. add readers) without weakening policies of other principals. E.g. the Jif statement declassify(e,L) relabels the result of expression e with label L.

```
class PasswordFile {
  private String[] nameList;
  private String{root:}[] passwordList;
  public boolean check(String user, String password)
    where authority(root){
    boolean match=false;
    try {
      for (int i=0; i<nameList.length; i++) {
        if (nameList[i].equals(user) &&
            passwordList[i].equals(password)) {
            match=true; break;
        }
      }
    } catch (NullPointerException e) {}
      catch (IndexOutOfBoundsException e) {}
    return declassify(match, {user; password});
  }
}
```

Figure 2. Jif Implementation of Class PasswordFile.

Now, we will consider an example (taken from [22]) where declassification is needed. Fig. 2 shows a class *PasswordFile* having an operation *check* which

[5] The Jif checker verifies that this property of the *least privilege* is obeyed.

takes a login name (*user:String*) and a password (*password:String*) and returns a boolean depending on whether *user* and *password* is contained in the arrays *nameList* and *passwordList*, respectively. On the one hand the operation should return a boolean value (i.e. leaking the information whether the password was correct), but on the other hand one does not want to leak the whole content of the array *passwordList*. To ensure this the elements of the array are labeled with {*root:*}. Certainly, one does not want to give the authority *root* to a normal user. So, the return value of operation *check* has to be *declassified* not to contain root. This means that principal *root* is removed from the owners of the return value (for more information on declassification see [22]). Thus, we have permitted to leak some information about the array *passwordList*. Declassification should be used with care. E.g. the above method can in fact leak all information in *passwordList* if the user is given the opportunity to call it repeatedly. We will come back to this issue later.

3 UML Extension Mechanism

The UML is a general purpose specification language. It can be adapted to particular domains by defining a *profile*. A profile is a conservative extension in the sense that it is not allowed to modify the metamodel. The application of a profile always results in a model that is still compliant with the metamodel. Thus, problems concerning semantics and interchangeability between tools are avoided.

The means for defining a profile are *stereotypes*, *tag definitions*, and *constraints*. A stereotype is used for extending metaclasses (defined in the metamodel) or other stereotypes. Like classes, a stereotype may have properties (in that context called *tag*). When a stereotype is applied to a model element, the values of its defined tag may be referred to as tagged values.

Finally, constraints can be used to define or refine the semantics of model elements. Constraints can be stated informally (e.g. using natural language) or formally using an adequate language (e.g. using the Object Constraint Language [24] which is an integral part of the UML).

4 Profile for Decentralized Label Model

In this section we define our profile which we call UMLsProfile (profile for security in UML). Like our previous UML extension UMLS [12], it is built on the decentralized label model. It permits confidentiality aspects to be considered in class diagrams.

4.1 Stereotypes

The aim of our profile is to provide stereotypes for annotating classes, attributes, operations, parameters, and return types of operations with confidentiality labels

and constraints. Tab. 1 shows the metaclasses that are extended (first column) by stereotypes (second column). The tags of the stereotypes are defined in the third column. The meaning of the stereotypes is explained in the following.

Metaclass, **Stereotype**	**Stereotypes**	**Tags**
TypedElement	*confidential*	*l:label*
Class	*authorityConstraint*	*authority:principal[*]*
Operation	*authorityConstraint*	*authority:principal[*]*
	actsForConstraint	*actsFor:(principal,principal)[*]*
	callerConstraint	*caller:principal[*]*
	beginLabel	*l:label*
	endLabel	*l:label*
<< send >>	*sendConfidential*	*l:label*

Table 1. Metaclasses extended by Stereotypes.

By using stereotype *confidential*, labels can be attached to instances of *TypedElement* which are attributes, formal parameters and return type of operations, and the values of objects.

As described in Sect. 2.2 classes might be given authority to permit declassification. In Tab. 1 we can see that this is achieved by extending the metaclass *Class* with stereotype *authorityConstraint*.

In addition to *authorityConstraint*, the profile (and Jif) offer the *callerConstraint* and *actsForConstraint* which extend metaclass *Operation*. The *callerConstraint* allows a caller to dynamically grant authority to the invoked operation. An operation with a *callerConstraint* may be called only if the caller possesses the required static authority.

In a hierarchy of principals, some principals can act for some other principals. This can be specified using the stereotype *actsForConstraint* which can be attached to operations. Then the operation can only be invoked if the specified constraint holds at the call site.

To prevent information leaks through implicit flows, the compiler associates a program-counter label with every statement and expression, representing the information that might be learned by their evaluation. A *beginLabel* can be specified to restrict the program-counter label at the point of invocation of a method—preventing a method from causing side-effects that have higher security than the value of *beginLabel*. Stereotype *endLabel* specifies what information can be learned from the fact that the method terminates normally. We will see an example on how to use them later.

It is also possible to give labels to individual exceptions which an operation might throw. In UML there already exists a stereotype *send* which can be applied to dependencies whose source is an operation and whose target is a signal,

specifying that the source sends the target signal. In Tab. 1, stereotype *sendConfidential* extends stereotype *send*. This permits us to attach labels to exceptions that are potentially sent by an operation.

The whole profile UMLsProfile is depicted in Fig. 3. The figure also shows the tags (and their types) of the stereotypes.

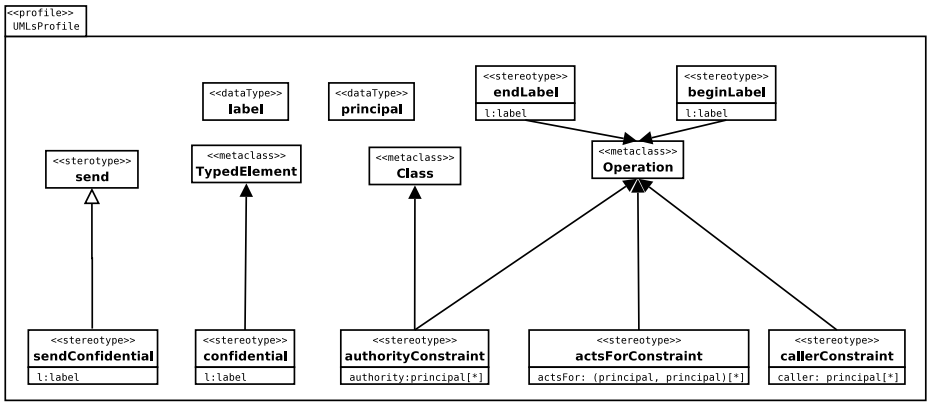

Figure 3. The Profile UMLsProfile.

The tags defined for the stereotypes above are of type *label*, *principal*, or are arrays of the respective type. The syntax of values of these types is defined in Fig. 1.

4.2 Examples and Default Values

In this section we first consider an example which involves labels and explain which default labels apply if no label is given. Thereafter, we consider declassification and shortly discuss the profile.

Labels. Fig. 4 shows an example of UMLsProfile applied to a class *Account*. In a UML diagram, the tagged values of a stereotype are denoted by UML notes attached to the element that is adorned with the stereotype. Consider for example attribute x annotated with stereotype *confidential*. A UML note is attached to attribute x defining its label $\{Bob:\}$. Attribute y is annotated with a label containing variable x. This means that the value of the label will be the same as for attribute x—in this case $\{Bob:\}$. Attribute z has no label. For attributes the default label is the empty label meaning that the attribute does not contain confidential data.

Next, let us consider the annotation of operation parameters. The formal parameter of operation *set* has the static label $\{Bob: Lise\}$. According to Jif,

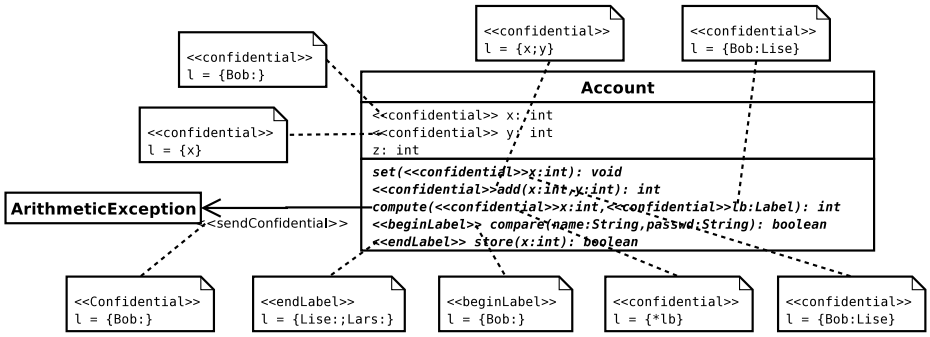

Figure 4. Example of Applying the Profile to Attributes and Operations.

this means that the *set* operation can only be invoked with arguments having this label. It would be quite tedious if one had to define a specific operation for every static label. The solution to that problem is *label polymorphism*. For example, in Fig. 4 the parameters x and y of operation *add* have no labels. This means that *add* can be called with any labels for its arguments—x and y will get the same labels as the arguments.

The first parameter of method *compute* shows how one can access the label contained *in* a variable of type *label*. The syntax is similar to the dereference operator * in the programming language C. Thus, the label $\{*lb\}$ denotes the label contained in *lb* rather than the label of *lb* (which is denoted by $\{lb\}$). When *compute* is invoked with the label value $\{Bob:\}$ for parameter *lb*, parameter x will also have the label $\{Bob:\}$. The second parameter *lb* of method *compute* is of type *label*. *lb* is a dynamic label and it would be legal to annotate it with a static label (e.g. $\{Bob : Lise\}$) since *lb* is a normal operation parameter.

Operation *compare* is an example for the use of a *begin-label*. A begin-label attached to an operation prevents the operation to be be called from a context with higher security. The default value for this kind of label is the program-counter label of the caller. In the example, the begin-label is $\{Bob:\}$. It forbids any assignment to attribute z in the body of *compare* since the empty label of z has lower security then the begin-label $\{Bob:\}$.

In the operation *add* we have attached the label $\{x; y\}$ to the return type. It was not necessary to explicitly state the return label in this case since the default label of the return value is the join of the parameter labels and the end-label, which in our case is $\{x; y\}$.

If an operation throws an exception, by default the exception has the same label as the end-label. However, it can also be given a different label as operation *compute* shows. Here, the label $\{Bob:\}$ is attached to the exception.

Declassification. In Sect. 2.2 we considered the class *PasswordFile* with the operation *check* that takes a login name (*user:String*) and a password (*pass-*

word:String) and returns a boolean. We argued that the class *PasswordFile* and the operation *check* needed the authority *root* to be able to declassify the boolean return value. Fig. 5 shows the class annotated with the authority constraint. Given this specification the code skeleton in Appendix A is generated automatically, where the type $Array < String, \{root:\} >$ of *passwordList* is translated into an array of type *String* whose elements are labeled with {*root:*}.

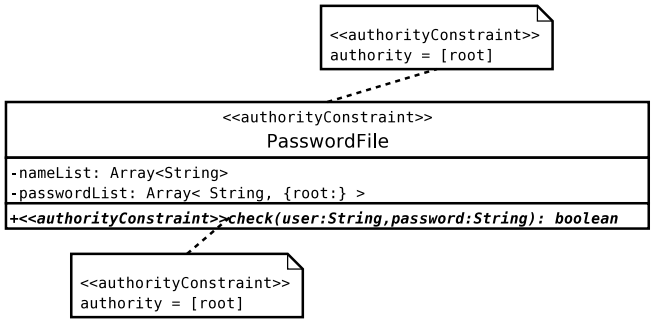

Figure 5. Example for a Class containing an *authorityConstraint*.

4.3 Notation

In a UML model the tagged values of a stereotype are denoted by UML notes attached to the element that is adorned with the stereotype. The example in Fig. 4 clearly shows that the default notation for stereotypes and tagged values clutters up the UML model and makes it hard to read. Fortunately, UML allows a profile to define its own notation that can be used instead of the standard notation of the model element which the stereotype is applied to [25, Sect. 18.3.7][6].

Our notation for the extended metaclasses follows the one in UMLS [12] and is depicted in Tab. 2. For the notation of values of *labels* and *principals* we use the syntax defined in Fig. 1.

Fig. 6 shows the example from Fig. 4 using the new notation. The advantages of our notation compared to the default notation are obvious.

Since we follow the UML standard, it is possible to use our profile with any UML-compliant tool. If a tool allows for adapting the concrete syntax, it is possible to use our more convenient notation. To guarantee interchangeability, it is however important that the XMI representation of the diagrams is independent of the chosen notation.

[6] Note, that we follow the "UML 2.0 Final Adopted Specification" which has not been finalized yet.

Metaclass	Notation
TypedElement	*element* ***label***
Class	Additional compartment for *authority* (see Fig. 5)
Operation	*visibility name* ***begin-label*** (*parameter-list*) ***end-label*** :*return-type-expression* ***return-label*** ***constraints***

Table 2. Notation for Metaclasses.

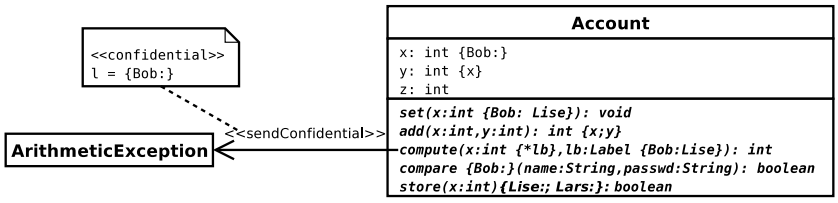

Figure 6. Application of UMLsProfile Notation to the Example shown in Fig. 4.

Parameterized Classes. The language Jif offers parameterized classes with respect to labels and principals. The UML 2.0 (final adopted specification) supports parameterized classes via the package *Template* as well. We give a little example here since parameterized classes play an important role in making reusable data structures with respect to labels and principals.

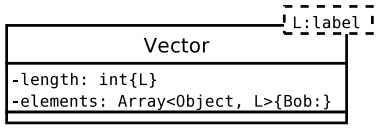

Figure 7. Vector parameterized on Label *L*.

Figure 7 shows a class *Vector* parameterized on a label *L* (in the dashed box). This label can be used to annotate attributes and operations of the class and makes it possible to instantiate *Vector* with different labels. In the example, attribute *length* is annotated with the parameter label *L*. Attribute *element* is of an array type which can have two labels: one for the array elements of type *Object* (here {*L*}) and one for the array reference (here {*Bob:*}).

4.4 Discussion

The decision on what data is confidential is usually not (and should not be) a programmer's job. Rather the customer/domain expert has to know which data

is confidential. Our profile permits this decision to be moved from code to the analysis/design level. We believe that this is crucial for building systems working with confidential data.

In some cases declassification is needed for being able to intentionally leak information (as the example from Fig. 2 and Fig. 5 showed). Note, that declassification is extremely powerful but therefore also very dangerous. Again, the question when to use declassification is an issue to be addressed on the analysis/design level. Also giving authority to classes or methods should be used with care. The places with authorities are spots where extra care has to be taken.

Finally, a short note on current UML tools. The idea behind producing a profile is that it can be applied by any UML-compliant tool. However, we encountered that most tools are not fully UML-compliant. For example, they do not allow to attach notes to arbitrary model elements (which is permitted according to the UML specification).

5 Related Work

Considering security in UML is a relatively new idea. Blobel, Pharow, and Roger-France [5] used use cases to consider security in a very informal way in a medical setting. We find it difficult to say anything about use cases since their semantics is not very well understood [10]. Furthermore, there has been work on developing a framework for model-based risk assessment of security-critical system using UML [13].

In our previous work [12] which was mainly focused on a case study we have already considered the treatment of confidential data in UML. However, we extended UML in a non-standard way. Furthermore, some features were left out, for example the treatment of *caller* and *actsFor* constraints. In this paper we give a more complete treatment of the decentralized label model in a UML-compliant manner.

The connection between language-based security and security on the specification level has been previously established by Mantel and Sabelfeld [20]. Their approach is more theoretical than ours. We hope that by choosing a more practical approach we will be able to reach more designers.

Closely related to our research is Jürjens' work on modeling confidentiality in UML [15,14,16]. Jürjens also uses the built-in extension of UML to define a profile called UMLsec. For checking constraints associated with the stereotypes of his profile, Jürjens defines a precise semantics for a restricted and simplified fragment of the UML building on a state chart semantics based on abstract state machines [6].

This approach has some limitations. The developer has to convince himself that the system is correct by examining the—possibly quite complex—UML diagrams. Furthermore, it is uncertain that the code created from these diagrams is correct. However, both problems can be eased by providing tool support.

A more serious problem are covert channels which arise from the concrete implementation of a program. For example, control flow or information about

termination of a program may reveal confidential information. So, even if confidentiality properties are proven on the UML level, which might be quite difficult in itself, there might be covert channels in the code. Our approach combining UML and Jif addresses this problem of indirect information flow [8]. The main difference between the two approaches is that Jürjens verifies security properties on specification level while we are working on the code level using the Jif type checker. So, both techniques should be complementary to each other. Furthermore, the decentralized label model permits more fine-grained control of confidential data than Jürjens' approach.

There has been some work that considers role-based access control in a UML setting [9,19]. Even though we have focused on information flow, there are some interesting parallels to this research. UMLsProfile/Jif permits declassification of data which can perhaps be considered as a form of access control.

6 Conclusion and Future Work

In this paper we have presented a profile UMLsProfile incorporating the decentralized label model into the UML. The profile allows fine-grained control of confidential data.

The decisions on which data must be kept confidential should be made at an early stage. This work permits confidentiality to be considered in the design phase of the development process. Using our profile in combination with Jif therefore contributes to building secure software systems. Jif code skeletons can be automatically generated from a class diagram making use of UMLsProfile.

In this paper we focused on class diagrams, probably the most important diagram type with a clear semantics that allows for code generation in a straightforward way. The next step in this line of work is to extend the profile UMLsProfile the other diagram types considered in UMLS. How Jif code can be generated from other diagram types needs further investigation.

There is one area we have not addressed in this paper, but which is important for our work: secure environments. Here, the deployment diagram in UML might be very useful when specifying secure environments for Jif programs. Furthermore, it would be interesting to look at state charts as well because they can be used to generate additional code which considers confidentiality. In particular, Jürjens' work [15] might be useful to take into account here.

Finally, Jif supports dynamic labels, but only in a restricted way to make sure that static checking of confidentiality is still possible. The restriction is that variables of type *label* may only be used to construct labels if they are *immutable*. We believe that this restriction could be dropped if in addition to type checking a theorem prover is used. The KeY tool [1] seems to be suited for that task for several reasons. KeY is a tool that supports the specification and verification of object-oriented software. It supports the specification languages UML/OCL and the implementation language Java. Thus, KeY seems to fit very well to UMLsProfile and Jif. Last but not least, KeY has already been successfully used to analyze secure information flow [7].

Acknowledgment. We thank R. Bubel, A. Roth, A. Sabelfeld, and the anonymous referees for important feedback on drafts of the paper.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY Tool. *Software and System Modeling*, pages 1–42, 2004. To appear.
2. D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report MTR 2547 v2, The MITRE Corporation, Nov 1973.
3. E. Bertino, S. Jajodia, and P. Samarati. Access Controls in Object-Oriented Database Systems: Some Approaches and Issues. In N. Adam and B. Bhargava, editors, *Advanced Database Concepts and Research Issues*, LNCS 759, pages 17–44. Springer, 1993.
4. K. J. Biba. Integrity Consideration for Secure Computer System. Technical Report ESDTR-76-372,MTR-3153, The MITRE Corporation, Bedford,MA, April 1977.
5. B. Blobel, P. Pharow, and F. Roger-France. Security Analysis and Design Based on a General Conceptual Security Model and UML. In P. M. A. Sloot, M. Bubak, A. G. Hoekstra, and B. Hertzberger, editors, *High-Performance Computing and Networking, 7th International Conference, HPCN Europe 1999, Amsterdam*, volume 1593 of *LNCS*, pages 918–930. Springer, April 12-14 1999.
6. E. Börger, A. Cavarra, and E. Riccobene. Modeling the Meaning of Transitions from and to Concurrent States in UML State Machines. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 1086–1091. ACM Press, 2003.
7. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In R. Gorrieri, editor, *Workshop on Issues in the Theory of Security (WITS)*. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.
8. D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, July 1977.
9. P. Epstein and R. Sandhu. Towards A UML Based Approach to Role Engineering. In *RBAC '99, Proceedings of the Fourth ACM Workshop on Role-Based Access Control*, pages 135–143, October 28-29 1999.
10. G. Génova, J. Llorens, and V. Quintana. Digging into Use Case Relationships. In J. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002*, volume 2460 of *LNCS*, pages 115–127. Springer, September/October 2002.
11. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
12. R. Heldal and F. Hultin. UMLS Bridging Model-based and Language-based Security. In E. Snekkenes and D. Gollmann, editors, *Computer Security - ESORICS 2003*, volume 2808 of *LNCS*, pages 235–252. Springer, 2003.
13. S. H. Houmb, F. Braber, M. S. Lund, and K. Stolen. Towards a UML Profile for Model-Based Risk Assessment. In *UML 2002 Satellite Workshop on Critical Systems Development with UML*, pages 79–91, September 2002.
14. J. Jürjens. Secure Java Development with UMLsec. In B. D. Decker, F. Piessens, J. Smits, and E. V. Herrenweghen, editors, *Advances in Network and Distributed Systems Security*, pages 107–124, Leuven, November 26-27 2001. International Federation for Information Processing (IFIP) TC-11 WG 11.4. Proceedings of the First Annual Working Conference on Network Security (I-NetSec '01).

15. J. Jürjens. Towards Development of Secure Systems using UMLsec. In H. Hußmann, editor, *Fundamental Approaches to Software Engineering (FASE, 4th International Conference, Part of ETAPS)*, volume 2029, pages 187–200, 2001.
16. J. Jürjens. UMLsec: Extending UML for Secure Systems Development. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002*, volume 2460 of *LNCS*, pages 412–425, Dresden, Sept. 30 – Oct. 4 2002. sv. 5th International Conference.
17. D. Kozen. Language-Based Security. In *Mathematical Foundations of Computer Science*, pages 284–298, 1999.
18. B. W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.
19. T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In J.-M. Jezequel, H. Hussmann, and S. Cook, editors, *The unified modeling language: model engineering, concepts, and tools; 5th international*, volume 2460, pages 426–441. Springer, 2002.
20. H. Mantel and A. Sabelfeld. A Generic Approach to the Security of Multi-Threaded Programs. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 126–142, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society Press.
21. A. Myers. Mostly-Static Decentralized Information Flow Control. Technical Report MIT/LCS/TR-783, MIT, 1999.
22. A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.
23. A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.
24. OMG. UML 2.0 OCL Specification. OMG Document, October 2003.
25. OMG. Unified Modeling Language: Superstructure, version 2.0, Final Adopted Specification. OMG Document, August 2003.
26. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
27. F. B. Schneider, G. Morrisett, and R. Harper. A Language-Based Approach to Security. In R. Wilhelm, editor, *Informatics—10 Years Back, 10 Years Ahead. Conference on the Occasion of Dagstuhl's 10th Anniversary*, volume 2000 of *LNCS*, pages 86–101, Saarbrücken, Germany, August 2000. Springer.

A Jif Code Skeleton for PasswordFile

The following Jif code skeleton can be automatically generated from the UML class diagram depicted in Fig. 5.

```
class PasswordFile {
  private String[] names;
  private String{root:}[] passwords;

  public boolean check(String user, String password) {}
}
```