Bridging Model-Based and Language-Based Security

Rogardt Heldal and Fredrik Hultin

Chalmers University of Technology, SE-412 96 Göteborg, Sweden
heldal@cs.chalmers.se, fredrik@hultin.info

Abstract. We present a way to support the development of software applications that takes into account confidentiality issues, and how the developed code can be automatically verified. We use the Unified Modelling Language (UML) together with annotations to permit confidentiality to be considered during the whole development process from requirements to code. We have provided support for software development using UML diagrams so that the code produced can be be validated by a language-based checker, in our case Jif (Java information flow). We demonstrate that the combination of model-based and language-based security is compelling.

1 Introduction

Our philosophy is that it should be convenient to consider security during the system development process, and that security should be automatically verifiable at code level. Addressing both of these aspects of system development is important to make a secure software system. We will show the benefits of combining a modelling language with a language-based security checker.

The development of software systems using UML [RJB99,OMG] has become the *de facto* standard for modelling object-oriented software systems in industry. There are several reasons for this: it is relatively easy to understand and learn, it permits several views of software systems, and it gives a good overview of the software's architecture. We aim to make it possible to consider security during a development process used in industry today, so UML is the obvious starting point.

One of the main problems with UML is that there has been a focus on functionality and less on constraints such as security. We want to use UML together with security annotations in such a way that developing secure programs becomes a seamless part of a project. This might seem like a difficult task since security requires rigorous treatment. Here, language-based checkers play an important role. In this approach, 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 including a formal proof or a type annotation. There have been several overview papers in this area [Koz99,SMH01,SM03]. Our combination of UML used together with annotation is intended to be used as a specification language to support building secure software systems, and a language-based checker should validate that the code really satisfies the security constraints. Therefore, our extended UML supports development of secure programs, and permits mistakes in the specification to be caught by the language-based checker. This is similar to modelling types in UML where the developer only needs to specify the types, and a type checker validates the types.

When we started to look for language-based security checkers, the Java information flow (Jif) system [Mye99a,Mye99b] was a natural choice since it handles a large subset of the object-oriented language Java [GJS96]. Object-orientation is important because UML is well suited to developing object-oriented systems. Jif is based on the Java language with some extra language constructs to control the release of data. The Jif system contains a type checker which guarantees that confidential data cannot leak. But, to make the system useful in practice, it permits data to be leaked in a controlled manner. This is acceptable provided that the system does not leak so much data that meaningful information can be derived. Technically, Jif deals with this problem in a simple way by giving part of the program authority to leak information[1].

In the standard security models, like the Bell-LaPadual model [BL73] and the Biba model [K.J77], the security policy is separated from the code. In this respect Jif differs in that the policy is incorporated into the code in the form of *labels*. Data are annotated with labels that specify the ownership and read permissions. The Jif type system checks whether the policies declared in the labels are satisfied. Jif is built on the decentralized label model [ML97,Mye99a]. In section 2 we will consider the labels of this model in more depth.

Java is not adequate for making programs which require tight control of confidentiality. Similarly, UML is not good for developing such programs. Therefore, we have created an extended version of UML, UMLS (Unified Modelling Language for Security). Our choice to support the development of Jif code had a large impact on how we extended UML. We did not extend UML in the standard way using UML's extension mechanisms (*stereotypes* and *tags* [OMG]) when modelling confidentiality. This was because we wanted more freedom in the choice of annotation in the current work. Furthermore, at present we do not automatically produce Jif code since we do not support any tools.

Several benefits follow from our work. Some of the diagrams — domain models, use-case diagrams, and activity diagrams — are so simple that most software system customers can be involved in the process of discussing confidentially issues. Customers are often the domain experts and they know best what information should be confidential. Therefore, involving the customer enhances the likelihood that confidentiality issues are handled correctly from the start. Furthermore, we have considered interaction diagrams and class diagrams where more detailed confidentiality issues can be considered by software designers. This permits confidentiality to be considered in greater depth during the development process.

[1] The Jif system has a way of dealing with information leaks, but no solution for deciding how much information can be leaked without causing problems. To solve this, information theory or complexity could be considered [VS00].

Domain models, use-case diagrams, activity diagrams, and interaction diagrams can be used to support the creation of the class diagrams[2]. Code skeletons can be created from class diagrams. There is still a lot of work to be done by the programmer, but there are confidentiality constraints on the attributes, operations, and classes which will guide/restrict the way the programmer will construct the code. This is a much simpler problem than writing the code without any support. The Jif compiler validates the code. If all confidentiality constraints are satisfied then the process is finished, otherwise the design/code has to be modified.

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 [DD77] and covert channels [Lam73]. Furthermore, the semantics of UML is still an open problem which makes it hard to prove things about UML. A further problem with validating the UML diagrams is that the transformation into code also has to be proven correct. We do not suggest our technique as an alternative for proving security on UML diagrams. It is often beneficial to prove security properties as early as possible. So, a combination of our technique and proving properties of the UML diagrams would be preferable.

In this paper we will first consider security labels similar to the ones used in Jif. Thereafter we will look at how we extended UML to consider confidentiality using labels. Then we will show a case study of how the extended UML diagrams can be used in a small process for developing a program which requires confidentiality. Finally, we will look at related work, conclusion, and future work.

2 Label

Modelling confidentiality in UMLS is done by using labels. Labels are used to specify the ownership and the read permissions of the data. We have chosen to use the same labels as used in Jif [Mye99b,ML97,ML98,ML00]. Types will be augmented with labels in UMLS.

Before we can discuss labels we first have to look at *principals* which are the building blocks of labels. A principal can be a user, a group, or a role. Principals can be arranged in hierarchies where a principal can act for another principal ("A can act for B" means that B can do anything that A can do). Principals are not purely static entities; they may also be used as values. First-class values of the new primitive type *principal* represents principals. For more information on run-time principals see [Mye99b].

To guarantee confidentiality the data needs to be annotated with labels. A label consists of policies, where a policy has the syntax: *owner:reader-list.*
The *owner* is a principal which owns the confidential data. This owner permits the principals in the *reader-list* to read the data. The *reader-list* is a list of comma separated principals that are able to read the data. Since a label can contain several policies, $\{policy_1; \ldots; policy_n\}$ the data can be owned by several

[2] We construct the code skeleton from the class diagram, which is one of the best understood diagrams in UML.

principals. A principal can only read the data if all the owners permit this — the reader is included in all the reader lists of a label. An owner is implicitly a reader and the label {} is the least restricted label. Here is an example of a label where *Bob* is the owner and *Lise* and *Lars* are readers: {*Bob:Lise,Lars*}.

Labels may exits as run-time entities as well, represented by the new type *label*. For more information on run-time labels and their use see [Mye99b].

3 UMLS

In this section we will augment UML with the labels introduced in the previous section. The UML diagrams augmented with labels are the class, the interaction, and the activity diagram. We will also give the syntax and informal semantics for the extension made to UML. We have chosen to give the syntax and semantics in similar fashion as in the *OMG Unified Modelling Language Specification 1.4*[3][OMG]. In this section we will also show the syntax of UMLS and to make our extensions clear they will be set in bold type. Let us start by looking at how we can use labels and principals to annotate class diagrams.

3.1 Class Diagrams

In a software development process, class diagrams are among the last diagrams to be considered before code is created. They usually contain information about the class name, the attributes and the operations. This makes it straightforward to construct a code skeleton directly from class diagrams.

In this section we will look at how to annotate classes, their attributes and operations in UMLS. We will also describe the parameterised class and some issues concerning authorities.

Class. The class is the central symbol in the class diagram. A class is modelled with attributes and operations. In UMLS as in UML, attributes and operations have specified compartments. We have also defined a new compartment for giving the authority of the class[4], which is needed to be able to declassify confidential data, see figure 1. The concept of authority in UMLS will be discussed further when we look at authority constraints later in this subsection.

Here we can see that besides giving the attribute *name* a type it can also be given a label. In other words, the *type-expression* is augmented with a *label*. If the *label* is omitted on an attribute, that means that there are no confidentiality constraints associated with it. The syntax of an attribute is:

visibility name: type-expression ***label*** = *initial-value*

[3] For the purpose of the presentation we have simplified the OMG syntax where it has no impact on the confidentiality extension.

[4] Due to Jif the authority list cannot be inherited, meaning that if a class C has a superclass C_s, any authority in C_s must also be in the authority clause of C. It is not possible to obtain authority through inheritance.

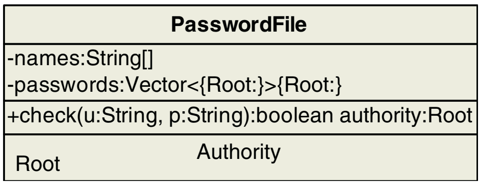
PasswordFile
-names:String[]
-passwords:Vector<{Root:}>{Root:}
+check(u:String, p:String):boolean authority:Root
Root
Authority

Fig. 1. Password file

The expression: *x: int*$\{Bob : Lise, Lars\}$ is an example of how to write an attribute with the type *int* augmented with a label where *Bob* is the owner and *Bob*, *Lise* and *Lars* can read the data. Now, when x is defined it can be used to restrict other variables, for example y : *int*$\{x\}$; meaning that the variable y should be as restricted as variable x.

The syntax of an operation is given by:

visibility name ***begin-label*** (*parameter-list*) ***end-label***
: *return-type-expression* ***return-label constraints***

where the *parameter-list* is a comma separated list of formal parameters, each given the syntax:

name : *type-expression* ***label***

Let us look at an example. Here is a public operation m with two arguments, x and y of type *int*, and a return value of type *String*. The two arguments are labelled with two different labels and in this case the return value is labelled with the joined label of the two arguments: +*m*(*x:int*{*Lise:*}, *y:int*{*Lars:*}) : *String*{*Lise:; Lars:*}

Labels may be omitted from an operation, signifying the use of implicit label polymorphism, e.g. the arguments of *check* in figure 1. When a formal argument's label is omitted, the operation is generic with respect to the label of the actual argument. We will come back to this when we consider interaction diagrams.

It is possible to specify the security context of operations with the *begin-label* and the *end-label*. The *begin-label* prevents a method from causing side effects that have lower security than the *begin-label*. The *end-label* specifies what information can be learned from the fact that the method terminates normally. For details the reader is referred to [Mye99a].

The default label for a return value is the *end-label*, joined with the labels of all the arguments. For example, for *check* in figure 1 the return label is $\{u; p\}$, so the return value could be written just as a *boolean*.

There are three types of constraints in UMLS; *authority*, *caller* and *actFor* [Mye99a]. In this paper we will only consider the authority constraint:

authority: principal-list This clause lists principals for which the operation is authorised to act. To be able to specify the authority of an operation the class needs to have at least this authority. For an example of how the operation

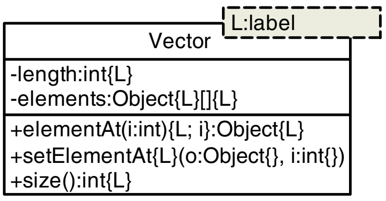
L:label
Vector
-length:int{L}
-elements:Object{L}[]{L}
+elementAt(i:int){L; i}:Object{L}
+setElementAt{L}(o:Object{}, i:int{})
+size():int{L}

Fig. 2. Vector class, showing attributes and operations

looks in a UMLS diagram see figure 1 where we have a class called *PasswordFile* with authority *root*. The authority is needed by operation *check* to be able to declassify information about whether the password is valid or not.

Parameterised Classes. Parameterised classes play an important role in UMLS for making reusable data structures with respect to labels and principals.

Let us look at an example. In figure 2 there is a class *Vector* parameterised on a label L (in the dotted box). This label is used to annotate attributes and operations of the class and makes it possible for *Vector* to be instantiated with different labels.

The attributes are annotated with the class's parameter label, L. From the figure 2 we can also see that *elements* has two labels. This is because an array needs special treatment. The first label is the label of the elements of the array. The second label is for the reference of the array.

The operation, *elementAt*, can be called with an index i as its argument. The *end-label* $\{L;i\}$ specifies what information can be learned by observing whether *elementAt* terminates normally. In this case the value returned will also have the same restrictions as the *end-label*.

In the operation *setElementAt* we need to prevent the method from causing side effect with a lower security level than $\{L\}$ by setting the *begin-label* to $\{L\}$. This is needed to be able to change any value in the array (which would fall into the category "causing side effects").

The specification of the *Vector* class put constraints on the Jif code written. The Jif code of *Vector* is given in Appendix A.

3.2 Relationships between Classes

There are many different types of relationships between classes. In this paper we need only to consider associations. They are modelled by drawing lines between the classes, see figure 7. Associations can contain *multiplicities* and *role names* [OMG] which we will see an example of in the case-study in section 4.

3.3 Interaction Diagrams

There are two types of diagrams for showing interactions between objects, the sequence diagram and the collaboration diagram. These two diagrams are similar

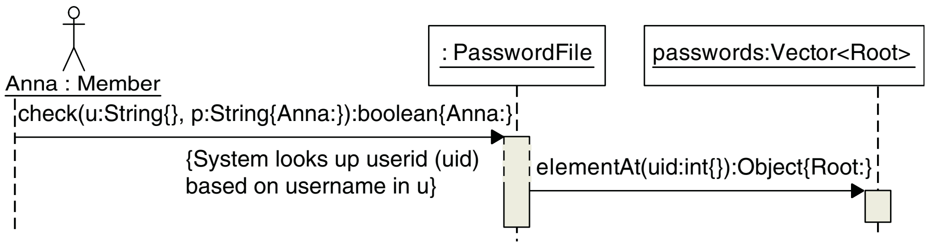
Anna : Member
: PasswordFile
passwords:Vector<Root>
check(u:String{}, p:String{Anna:}):boolean{Anna:}
{System looks up userid (uid) based on username in u}
elementAt(uid:int{}):Object{Root:}

Fig. 3. Password sequence

and for our perspective the distinction between them is not important. Interaction diagrams show dynamic properties of how objects work together to solve a larger problem, in contrast to class diagrams which shows static properties about classes, but these two diagrams are strongly related. For each message sent to an object in the interaction diagram there needs to be a matching operation in the class diagram[5].

Interaction diagrams are good for considering flow of data among objects. Data values flow to objects through arguments and back from objects via the result value. These values can be annotated with confidentiality constraints. Here is the syntax for the *sequence-expression*:

$$\begin{aligned}\textit{return-value} := \;& \textit{message-name}\ (\textit{argument-list}) : \\ & \textit{return-type}\ \boldsymbol{\textit{return-label}}\end{aligned}$$

where the syntax of *argument-list* is a comma separated list of arguments and labels: *argument* : *type-expression* ***label***.

Let us consider the sequence diagram in figure 3. First, *Anna* wants to check her password using *PasswordFile* from figure 1. The user name, *u*, is not confidential, but the password, *p*, is owned by principal *Anna*. Each user name is related to a number, *uid*, which is used to find the password in the vector from figure 2. By choosing the template parameter of the vector to be of principal *Root*, the data contained within the class will be owned by *Root*. Since this is the case the data returned from the vector must be at least as restrictive as *Root*. The password from the vector will be compared with the password from *Anna* producing a boolean value belonging to them both. Since *Anna* cannot read this value it has to be declassified. Here we have an interesting design question: how much authority should be given to *PasswordFile*? We decided to give *PasswordFile* the authority *Root* as this permits the method *check* to remove *Root* as owner of the boolean value returned to *Anna*. As we can see the interaction diagram helps to identify places where authority declarations must be considered.

In another scenario Bob might want to check his password using the principal *Bob*. This is no problem since the operation is defined as: $check(u : String,$

[5] There is one restriction on the use of interaction diagrams due to the fact that Jif cannot handle threads, so it makes no sense to talk about asynchronous communication. This is a limitation we hope will be removed in the future.

p : *String*) in *PasswordFile* which permits any label on u and p. The only change in the sequence diagram in figure 3 is that we change all principals *Anna* to *Bob*. Now, let us change the operation check belonging to *PasswordFile* to $check(u : String, p : String\{Anna :\})$. The sequence diagram in figure 3 will look the same, but now Bob cannot use the *check* operation any more. This distinction is shown in the class diagram, but not in the sequence diagram, because all principals are known in the sequence diagram.

3.4 Use Case Diagrams

Use case diagrams are used to describe the behaviour of the system in the form of use cases and the actors of the system — actors are the things which interact with the system through use cases.

The description of use cases is often done informally by description in running text. Confidentiality constraints might be considered as a part of use cases, but it is more natural to consider them as separate documents, which can be related to use cases. It is worth noting that interaction diagrams are often used to realise use cases. They are more formal than use cases and are therefore a better place to handle confidentiality constraints in a more formal way.

One benefit of using use case diagrams is that they identify the actors of the system. These actors can be used to define principal hierarchies.

3.5 Activity Diagrams

The last diagram type we will consider in this paper is the activity diagram. Activity diagrams can be used to model the flow of activities which happen in a system, a use case or a method. It is possible to show what kind of data are moved among activities within these diagrams. Furthermore, activity diagrams can contain *swimlanes* which can be used to separate the activities done by separate people, groups or organisations. This makes the activity diagrams perfect for showing how confidential data are moved among separate people, groups or organisations on an abstract level.

4 UMLS a Case Study

In this section we will show how UMLS can be used as part of a process, such as RUP[JBR99]. We will limit the discussion to the parts which are of interest when considering confidentiality. How to use UMLS in a process will vary depending on the project, in the same way as standard UML. Here we are going to look at stages where we found UMLS useful in a development process when considering confidentiality based on our case study.

4.1 The System

The example is that of a small medical application where patients can ask for information about diseases based on symptoms they provide. To obtain this information the patients also have to pay with a bank card. Since patients need to pay for the information, personal information that identifies the patients is also sent to the system. Since personal information needs to be sent to the system together with the symptoms the system could leak information about a particular patient's illness. We want to prevent this.

4.2 The Use Cases Diagram

Use-cases were developed to explore the behaviours of the system. Here we only consider the use case where a patient requires information about a disease. This use case contains interesting confidentiality issues. Due to the limited space we can only describe the *Casual Version* [Coc01] of the use case:

Use case: Obtain information about disease
The patient sends information about the symptoms and the payment to the medical-system. The medical-system validates the payment and charges the patient the specified amount. Based on the symptoms the medical-system looks up a matching disease, prepares a response and sends it to the patient.
Confidentiality constraints: Any information sent to the system regarding the patient's symptoms are strictly confidential to the patient. Only the patient himself and the medical-system should be permitted to read the payment information. The medical-system should not leak more information than absolutely necessary to inform the patient about his illness[VS00].

As we can see from this description, interesting confidentiality issues can be considered at use-case level in an informal way. This description can easily be discussed with the customer.

4.3 The Activity Diagram

We use the activity diagram to better show the flow of confidential data in the medical system.

In figure 4 we have two swimlanes which separate the patient, here represented by *Lise*, and the medical system. It is interesting to consider what confidential data flows between the patient and the medical system. For example we can see that *Symptom* and *Payment* flow from the patient to the medical system. Look at *Payment*, this object contains *amount* and *cardNumber* which are owned by *Lise* and are readable by *Doctor*. In the case of *Symptom* we want the data to be owned by *Lise* but not readable by the medical system. This is because the patient do not trust the medical system and therefore want to prevent the system from sending the *Symptom* to an output channel such as a monitor or a printer. The medical system can still use the *Symptom* to find the correct treatment.

The interesting part is the activity *Lookup disease* which uses information from the patient, *Symptom*, and the doctor to find a disease. The result, *Disease*, contains data owned by both patient and doctor and therefore not readable by

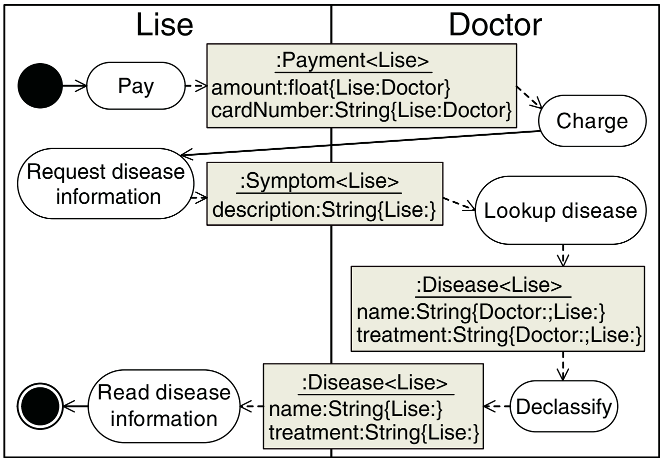
Lise
Doctor
Pay
:Payment<Lise>
amount:float{Lise:Doctor}
cardNumber:String{Lise:Doctor}
Charge
Request disease information
:Symptom<Lise>
description:String{Lise:}
Lookup disease
:Disease<Lise>
name:String{Doctor:;Lise:}
treatment:String{Doctor:;Lise:}
Read disease information
:Disease<Lise>
name:String{Lise:}
treatment:String{Lise:}
Declassify

Fig. 4. Medical system activity diagram

anyone. To make the data readable by the patient the data has to be declassified, which is done in the activity *Declassify*.

This diagram contains more information about the activities and flow of data than the use case diagram, but it is still quite informal. From this diagram we have obtained a better understanding of how the confidential data flows. Furthermore, we have started to consider principals, labels, and declassification. It is also natural to review this diagram together with the customer of the system.

4.4 The Domain

A domain model contains only concepts from the domain under consideration, and not software classes. A restricted form of class diagram is used for modelling domains, containing class names, attributes, and associations among classes. Since the domain model is a central part of the problem description this is an appropriate place to consider confidentiality.

Based on information in our use case and the activity diagram, we create a domain model[6], see figure 5. Notice that all the concepts in our domain model correspond to real world concepts: *Payment*, *Symptom*, *Patient*, *Doctor*, and *Disease*. A *Patient* is related to a *Payment*, *Symptom* and *Doctor* via associations. Furthermore, the multiplicity 1 on the association between *Patient* and *Payment* says that the *Patient* is associated with one *Payment* while the *Doctor* is associated with several *Diseases*, since the multiplicity is *. The names attached to one side of the associations are role names, here used to give the

[6] Some people prefer to model the system directly in the domain model, skipping the use cases. They believe that they obtain an object-oriented system of higher quality, which is easier to extend, reuse, and maintain.

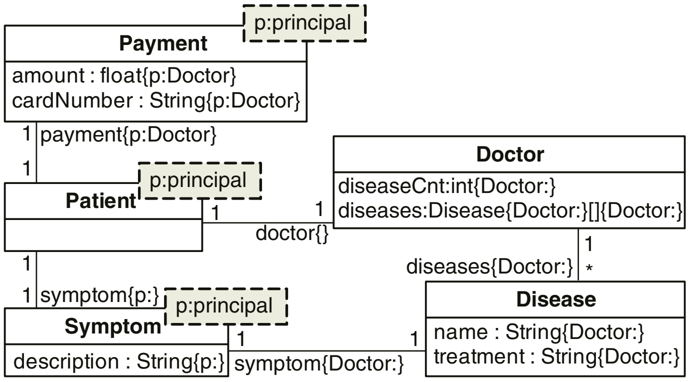

Fig. 5. Medical system domain mode

names of the attributes, for example *payment* is an attribute in *Patient* referring to *Payment*.

We will only consider a few of the confidentiality constraints considered in figure 5. The label {} on the role name *doctor*{} shows that the reference is not confidential. But, the attributes inside *Doctor* have confidentiality constraints on them. We have chosen principal-templates, to make *Patient*, *Symptom*, and *Payment* reusable, for example $Symptom\langle Lise\rangle$ would have an attribute *description* with the label $\{Lise{:}\}$.

In our case study the domain model provides deeper information about confidentiality issues than the use-case and activity diagram. But, it is still possible for a customer to consider the domain model. By using use-case diagrams, activity diagrams, and a domain model one can build up an understanding of the confidentiality issues with the system to be built.

4.5 The Interaction Diagram

Now we move from analysis to design. From the previous diagram we have obtained an informal understanding of how objects communicate confidential data. Here we will make this more precise with the help of a collaboration diagram.

In figure 6 we can see how *DataDoctor*, *Disease*, and *Symptom* collaborate to give information back to the patient, *Lise*, about her disease. From the numbers in front of the calls, we can see that the order of calls is: *getDisease*, *charge*, *match*, *equals*.

Let us consider *getDisease*. For each patient we want an instance of *DataDoctor* to be able to handle the call from the particular patient. This means that one instance of the *DataDoctor* only can handle one patient with a particular principal. As we can see from the diagram in figure 6, the call *getDisease* has the arguments $s : Symptom\langle\{Lise :; Doctor :\}\rangle$ and $p : Payment\langle Lise\rangle$. For the *DataDoctor* to handle this call it needs to know about the principal, *Lise*, which

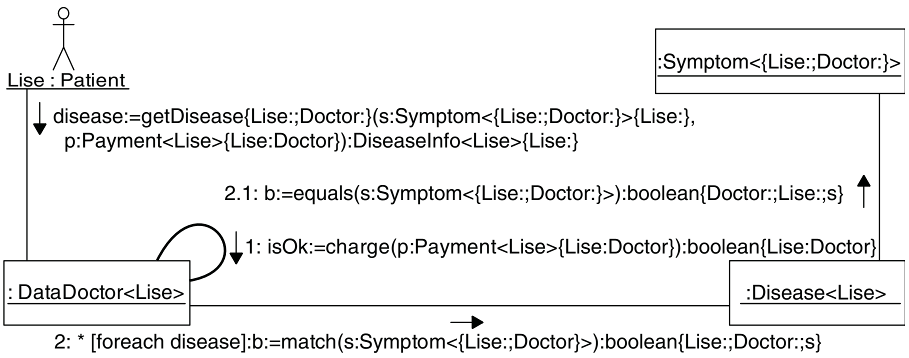
Lise : Patient
:Symptom<{Lise:;Doctor:}>
disease:=getDisease{Lise:;Doctor:}(s:Symptom<{Lise:;Doctor:}>{Lise:},
p:Payment<Lise>{Lise:Doctor}):DiseaseInfo<Lise>{Lise:}
2.1: b:=equals(s:Symptom<{Lise:;Doctor:}>):boolean{Doctor:;Lise:;s}
1: isOk:=charge(p:Payment<Lise>{Lise:Doctor}):boolean{Lise:Doctor}
: DataDoctor<Lise>
:Disease<Lise>
2: * [foreach disease]:b:=match(s:Symptom<{Lise:;Doctor}>):boolean{Lise:;Doctor:;s}

Fig. 6. Get disease collaboration diagra

is done through the template mechanism and therefore the $DataDoctor\langle Lise\rangle$ in the collaboration diagram. There are similar reasons for the $Disease\langle Lise\rangle$.

In our system we wanted to compare one *Symptom* owned by the principal *Doctor* and another owned by the patient, *Lise*. To simplify the comparison, due to Jif, we chose to make the *Symptom* owned by both principals — making it more confidential so it still satisfies the confidentiality constraint for the system. For this reason we changed *Symptom* into a class of template label instead of template principal, see figure 6. This is also the reason why the instance of the *Symptom* in *getDisease* has the label $\{Lise :; Doctor :\}$, but there is no need to make the reference more restrictive than {*Lise:*}

The instance disease of *DiseaseInfo* returned to *Lise* contains information about the treatment recommended for her illness. This data comes from information owned by both the patient, *Lise*, and the *Doctor*, the reason being that we need information from both the medical system and the patient to find the disease. This is done in the operation *match* which also uses *equals*. Therefore the *DataDoctor* needs to declassify the information so that the content can be read by *Lise*. In our case we found the need for declassification already in the activity diagram, see figure 4. We do not know if this is the case in general since the activity diagrams are often treated informal when used in the beginning of a process. In contrast to the the interaction diagram which by nature is more formal.

As we can see, collaboration diagrams are useful to show what confidential data flows from and to objects in the form of parameters and return types. These diagrams are often too detailed for a customer, but good for a designer moving one step closer to code.

4.6 The Class Diagram

We constructed the class diagram, figure 7, from the domain and the collaboration diagrams. As we can see from the collaboration diagram in figure 6 we need one more class, *DiseaseInfo*, and the template parameter of *Symptom* changed

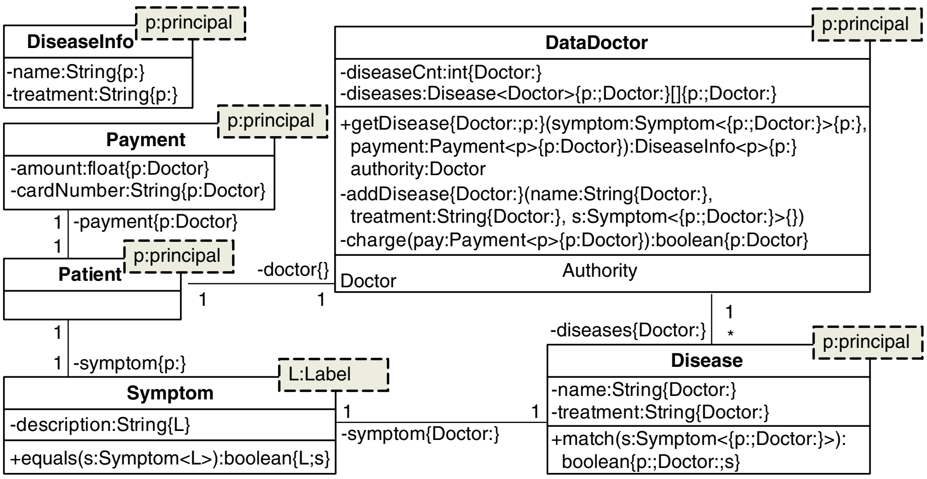
p:principal
DiseaseInfo
-name:String{p:}
-treatment:String{p:}
p:principal
Payment
-amount:float{p:Doctor}
-cardNumber:String{p:Doctor}
1 -payment{p:Doctor}
1
p:principal
Patient
-doctor{}
1
1
1
1 -symptom{p:}
L:Label
Symptom
-description:String{L}
+equals(s:Symptom<L>):boolean{L;s}
1
-symptom{Doctor:}
1
p:principal
DataDoctor
-diseaseCnt:int{Doctor:}
-diseases:Disease<Doctor>{p:;Doctor:}[]{p:;Doctor:}
+getDisease{Doctor:;p:}(symptom:Symptom<{p:;Doctor:}>{p:}, payment:Payment<p>{p:Doctor}):DiseaseInfo<p>{p:} authority:Doctor
-addDisease{Doctor:}(name:String{Doctor:}, treatment:String{Doctor:}, s:Symptom<{p:;Doctor:}>{})
-charge(pay:Payment<p>{p:Doctor}):boolean{p:Doctor}
Doctor
Authority
1
*
-diseases{Doctor:}
p:principal
Disease
-name:String{Doctor:}
-treatment:String{Doctor:}
+match(s:Symptom<{p:;Doctor:}>): boolean{p:;Doctor:;s}

Fig. 7. Class diagram

to a label. Furthermore, operations in *DataDoctor* and *Disease* require that we make them into template classes.

All the calls from the collaboration diagram are added as operations to the representing class in the class diagram plus one extra operation, *addDisease*. The reason we add the operation *addDisease* is that it shows one example of how to consider begin-labels during the creation of the class diagram. This operation has side-effects on the attribute *diseases* so it is natural to give it the begin-label {*Doctor:*}. The begin-label on *getDisease* was added for more technical reasons which came up during the implementation of the system.

From the collaboration diagram we know that the class *DataDoctor* needs to declassify information owned by *Doctor* and therefore needs the authority of the *Doctor*.

4.7 From UML to Jif

The diagram we translate into code is the class diagram. The information in the class diagram contains all the information needed to create the class skeleton containing the attributes and the method definitions (not the body). We have implemented the case study in Jif, see Appendix B for a code skeleton of *DataDoctor*. The translation from UMLS's class diagram to code skeleton is not hard, so we should be able to do this automatically.

After the code has been constructed the Jif compiler validates the confidentiality constraints. One of the major strengths of using UMLS for specification and Jif for validation is that the designer can consider security during the design phase, then gets an extra software validation step of the code which guarantees that indirect information flows [DD77] are not introduced.

4.8 Discussion

Each diagram is used to consider a different aspect of the confidentiality of the system. These diagrams are used to enhance the understanding of the system during the design phase much in the same way as the standard UML is used in the industry today.

The case study has been presented sequentially in this paper, however during the development we iterated several times through the different diagrams. The construction of the system was done by writing the diagrams on a white-board. This created a lot of discussion and several interesting confidentiality issues came up during this process. Coding confidential systems is a hard and error prone task. We feel that by using visual diagrams this task is simplified, and it is easy to invent a process to create code directly from the diagrams, making it easy for the developers to create large systems.

5 Related Work

To consider security in UML is a relatively new idea. Blobel, Pharow and Roger-France [BPRF99] used use cases to consider security in a very informal way in a medical setting. We find it very difficult to say anything about use cases since they are very informal and not very well understood semantically [GLQ02]. Furthermore, there has been work on developing a framework for model-based risk assessment of security-critical system using UML [HBLS02].

The connection between language-based security and security on the level of specifications has also been previously established by Mantel and Sabelfeld [MS01]. They have chosen a more theoretical approach than we have done. We hope that by choosing a more practical approach we will be able to reach more designers.

The research which is mostly related to ours is Jan Jürjens' work on modelling confidentiality in UML [Jür01b,Jür01a,Jür02]. Jürjens uses state-chart diagrams to handle confidentiality problems of a system. Being the first to consider confidentiality with UML it is only natural that his approach has several limitations. Firstly, the developer has to convince himself that the system is correct by examining the UML diagrams, which might be quite complex. Secondly, it is uncertain that the code created from these diagram is correct since that depends on how the code is created. Thirdly, the code is needed in order to find the covert channels. So, even if confidentiality properties are proved on the UML diagrams, which might be quite difficult in itself, there is no guarantee that the code correctly implements confidentiality constraints. All these problems are addressed by our approach.

One further problem is that Jürjens' work relies on a precise semantic definition of the state-chart diagram. Jürjens overcomes this problem by using a limited part of UML to which he gives his own semantics. Jürjens has moved towards using UML's extending mechanism for modelling confidentiality [Jür02], *stereotypes* and *tags* [OMG]. We have chosen not to do this, because we have not found any good way of expressing our extensions using stereotypes and tags that is as readable as our annotations.

There has been some work that considers role-based access control in a UML setting[ES99,LBD02]. Even though we have focused on information-flow, there are some interesting parallels to this research. UMLS/Jif permits declassification of data and this can perhaps be viewed as a form of access control.

6 Conclusion and Future Work

In this paper we have used a case study to demonstrate that our extensions to UML simplify the process of producing programs with confidentiality constraints/requirements. Furthermore, we have motivated the importance of using a language-based checker to validate the code. We believe that the combination of modelling confidentiality with a modelling language and validating the code with a language-based checker is crucial for building large applications that require a high degree of confidentiality.

The UML diagrams we have considered in this paper are, in our experience, often used in the development of object-oriented software which is the main reason behind our choice of diagram types. It would be interesting to look at state-chart diagrams as well, because state-chart diagrams can be used to generate additional code which considers confidentiality. Work done by Jürjens might be useful to consider here [Jür01b].

The main purpose of this paper is to show the powerful combination of a modelling language and a language-based checker. To take this research a step further requires more work on Jif, UMLS, and a tool to integrate them. Another interesting direction would be to see if there are other language-based checkers which also can be combined with UMLS or UML.

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. This is also something we intend to study further.

Acknowledgement

Thanks to Andrew Myers, Stephen Chong, and Lantian Zheng at Cornell for enhancing our knowledge about the Jif environment, and for advising us in how to implement correct Jif code for our case-study. Furthermore, we want to thank Daniel Hedin for several important discussions about our research. We also want to thank colleagues at computing science department at Chalmers for comments on the paper, and the three anonymous referees.

References

BL73. D. Bell and L. LaPadula. Secure Computer Systems:Mathematical Foundations and Model. Technical Report MTR 2547 v2, The MITRE Corporation, Nov 1973.

BPRF99. 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 *Lecture Notes in Computer Science*, pages 918–930. Springer, April 12-14 1999.

Coc01. Alistar Cockburn. *Writing Effective Use Cases.* Addison Wesley, 2001.

DD77. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 77.

ES99. 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.

GJS96. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

GLQ02. 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.

HBLS02. S. H. Houmb, F. Braber, M. Soldal Lund, and K. Stolen. Towards a UML Profile for Model-Based Risk Assessment. In *Critical Systems Development with UML-Proceedings of of the UML'2 workshop*, pages 79–91, September 2002.

JBR99. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Number ISBN 0-201-57169-2 in Object Technology. Addison-Wesley, 1999.

Jür01a. J. Jürjens. Secure Java Development with UMLsec. In B. De Decker, F. Piessens, J. Smits, and E. Van Herrenweghen, editors, *Advances in Network and Distributed Systems Security*, pages 107–124, Leuven (Belgium), November 26-27 2001. International Federation for Information Processing (IFIP) TC-11 WG 11.4, klu. Proceedings of the First Annual Working Conference on Network Security (I-NetSec '01).

Jür01b. 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.

Jür02. J. Jürjens. UMLsec: Extending UML for Secure Systems Development. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002 – The Unified Modeling Language*, volume 2460 of *lncs*, pages 412–425, Dresden, Sept. 30 – Oct. 4 2002. sv. 5th International Conference.

K.J77. K.J.Biba. Integrity consideration for secure computer system. Technical Report ESDTR-76-372,MTR-3153, The MITRE Corporation, Bedford,MA, April 1977.

Koz99. Dexter Kozen. Language-Based Security. In *Mathematical Foundations of Computer Science*, pages 284–298, 1999.

Lam73. Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.

LBD02. Torsten Lodderstedt, David Basin, and Jürgen Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In Jean-Marc Jezequel, Heinrich Hussmann, and Stephen Cook, editors, *The unified modeling language: model engineering, concepts, and tools; 5th international*, volume 2460, pages 426–441. Springer, 2002.

ML97. Andrew C. Myers and Barbara Liskov. A Decentralized Model for Information Flow Control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.

ML98. Myers and Liskov. Complete, Safe Information Flow with Decentralized Labels. In *RSP: 19th IEEE Computer Society Symposium on Research in Security and Privacy*, 1998.

ML00. Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

MS01. 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.

Mye99a. A. Myers. Mostly-Static Decentralized Information Flow Control. Technical Report MIT/LCS/TR-783, MIT, 1999.

Mye99b. Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

OMG. OMG. *Unified Modeling Language Specification.*

RJB99. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Number ISBN 0-201-30998-X in Object Technology. Addison-Wesley, 1999.

SM03. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

SMH01. Fred B. Schneider, Greg Morrisett, and Robert Harper. A Language-Based Approach to Security. *Lecture Notes in Computer Science*, 2000:86–101, August 2001.

VS00. Dennis M. Volpano and Geoffrey Smith. Verifying Secrets and Relative Secrecy. In *Symposium on Principles of Programming Languages*, pages 268–276, 2000.

A Jif Code of Vector

```
public class Vector[label L] {
    private int{L} length;
    private Object{L}[]{L} elements;

    public Vector{L}(){ resize(10); }

    public Object{L} elementAt(int i):{L;i}
        throws (ArrayIndexOutOfBoundsException){
            return elements[i];
    }

    public setElementAt{L}(Object{} o, int{} i) {
        if (i >= length)
            resize();       // make the array larger
        elements[i] = o;
    }

    public int{L} size(){ return length; }
    private void resize{L}(){...}
}
```

B Code Skeleton of DataDoctor

```
class DataDoctor[principal patient] authority(Doctor) {

    private Disease[patient]{Doctor:;patient:}[]{Doctor:;patient:} diseases;
    private int{Doctor:} diseaseCnt;

    public DiseaseInfo[patient]{patient:} getDisease{Doctor:;patient:}
     (Symptom[{patient:;Doctor:}] {patient:} s, Payment[patient]
     {patient:Doctor} payment) where authority(Doctor){...}

     public void addDisease{Doctor:} (String{Doctor:} name, String
      {Doctor:} treatment,Symptom[{Doctor:;patient:}] {} s) {...}

     private boolean{patient:Doctor}charge(Payment[patient]
      {patient:Doctor} pay){...}
}
```