



Adaptable Access Control for Electronic Medical Records

Kung Chen^a, Yuan-Chun Chang^a, and Da-Wei Wang^b

^aDepartment of Computer Science, National Chengchi University

^bInstitute of Information Science, Academia Sinica & National Yang-Ming University

^a{chenk,g9305}@cs.nccu.edu.tw; ^bwdw@iis.sinica.edu.tw

Abstract

This paper presents an aspect-oriented approach to providing adaptable access control framework for Electronic Medical Records (EMR) on Web-based platform. In our scheme, access control logic is decoupled from the core of application and collected into separate aspect modules which are automatically synthesized from access control rules in XML format and properly designed aspect templates. The generated aspect modules will then be compiled and integrated into the underlying application using standard aspect tools. At runtime, these aspect codes will be executed to enforce the required access control without any runtime interpretation overhead. Future changes of access control rules can also be effectively realized through these mechanisms without actual coding. This will not only improve the system's modularity but also make the task of enforcing comprehensive access control more adaptable. We have built a Web-based EMR prototype implementation using AspectJ to demonstrate our approach.

Keywords: access control, electronic medical record, aspect-oriented programming

1. Introduction

There is little doubt that healthcare information systems (HIS) will move towards a fully integrated electronic medical record (EMR). However, as we move closer to a paperless environment and Internet-based applications, we must realize that the risks to privacy and security incurred by using electronic systems are also increased. This is a very complicated issue. Besides social aspects, it also calls for much technological advancement to achieve a proper balance between public interests and personal privacy. In terms of information system development, an effective and flexible access control mechanism is clearly an essential part of a secure and privacy enhanced EMR system [4].

However, it is not easy to develop a comprehensive yet flexible mechanism for access control in HIS with EMR. There are at least two major difficulties. First, like other security requirements, access control is a system-wide concern that permeates through all the major modules of a system. Hence it is very often to see the code for implementing access control scattered over the whole system and tangled with other functional code.

This is not only error-prone but also makes it difficult to verify its correctness and perform the needed maintenance.

Second, access control rules in healthcare domain are inherently fine-grained and dynamic. It is common for information system developers to partition users into different categories, e.g. by roles in an organization, and define access privileges in terms of the application functions that a particular category of users has a right to access, e.g., an administrative clerk is limited to administrative functions and excluded from transaction functions. For HIS, however, an additional level of access control must be defined at the *data field* level. Users may be allowed to access a specific function, but some data elements may be excluded from view. Furthermore, access may be limited to specific patient records or specific elements within a patient record. For example, while a physician can view some fields of a patient's EMR, only the patient's *attending* physician can see the whole record and modify it. On the other hand, we will have to bypass the constraint in an emergency. In addition, changes in legislation or changes in the interpretation of legislation can lead to major revision of the access control rules. All these lead to the needs of frequent changes for access control rules in healthcare domain.

An ideal approach to overcome these difficulties, referred to as *adaptable access control* [9][10], is to provide a framework where the access control logic is separated from the core of application and specified declaratively in a configuration file without actual coding, while still being able to satisfy fine-grained access control requirements and incurring a low runtime overhead. This will not only improve the system's modularity but also make the task of enforcing comprehensive access control more adaptable. Here we show that the emerging technology of aspect-oriented programming (AOP) [6] is a feasible technology to providing adaptable access control. Specifically, we propose an aspect-oriented framework for adaptable access control and present a Web-based EMR prototype implementation using AspectJ [7] to demonstrate our approach.

The remainder of the paper is organized as follows. Section 2 introduces AOP and AspectJ briefly. Section 3 outlines our approach to adaptable access control and overviews our system architecture. Section 4 and 5 explain our design of access control rules and aspect code templates, followed by a description of the main



steps for synthesizing the access control aspects in Section 6. Our prototype implementation of a Web-based EMR is presented in Section 7. Finally, Section 8 concludes and outline our future work.

2. Background: AOP and AspectJ

In this section, for the sake of completeness, we highlight the basics of AOP and review the relevant features of AspectJ.

Aspect-oriented programming (AOP) aims at modularizing concerns such as profiling and security that crosscut the components of a software system [9]. In AOP, a program consists of many functional modules and some *aspects* that encapsulate the crosscutting concerns. An aspect module provides two kinds of specifications: *Pointcut*, comprising a set of well-defined points in the execution of a program, designate when and where to crosscut other modules; and *advice*, which is a piece of code, that will be executed when a pointcut is reached. The complete program behavior is derived by some novel ways of composing functional modules and aspects according to the specifications given within the aspects. This is called *weaving* in AOP. Weaving results in the behavior of those functional modules impacted by aspects being modified accordingly.

AspectJ is a seamless aspect-oriented extension to the Java programming language [8]. It provides a rich pointcut language through a powerful join point model. Typical join points in AspectJ are method call/execution and field access. Advice in AspectJ is an anonymous method bound to a pointcut and tagged with one of the three keywords: *before*, *after*, or *around*. The before advice and the after advice are executed before and after the intercepted method, respectively. The case for the around advice is more subtle. Inside the around advice, we can choose to resume the intercepted method by calling the special built-in method *proceed()*, or simply bypass its execution. The following aspect in AspectJ illustrates the power of around advice. It states that, when the update method of class `Patient` is about to execute, control will be transferred to the around advice. If the particular constraint is not satisfied, the intercepted method call will be aborted and the exception handler will take over; otherwise, it will be resumed by calling *proceed()*.

```
public aspect AccessControlPrecheck {
    pointcut pc(Data d):
        execution(public void
            Patient.update(Data d)) && args(d);
    void around(Data d) : pc(d){//advice
        if (constraint(d) )
            proceed(d); // granted
        else forwardToExceptionHandler
            ("AccessDenied");
    } // end around advice }
```

Note that “args(d)” is also a kind of pointcut in AspectJ that can capture the argument(s) passed to the intercepted method.

Furthermore, AspectJ also allows aspect inheritance, abstract aspect, and abstract pointcut. We can write an aspect with abstract pointcuts or abstract methods. A sub-aspect then extends the abstract aspect and defines the concrete pointcuts and methods.

3. Our Approach and System Architecture

We propose to apply aspect-oriented technology to design and implement a highly adaptable mechanism for enforcing fine-grained access control in Web-based EMR system. We worked toward this goal from two opposite ends and managed to meet in the middle. At one end, the objective is to accommodate more requirements. We use a flexible modeling scheme based on the tree-structure of EMR and user security attributes that can satisfy a wide range of access control requirements of various granularity levels, including those involving data contents. A high-level form of access control rules is derived from it. At the other end, since access control is a system-wide crosscutting concern, we must impose considerable architectural disciplines on Web applications to layout a good foundation for enforcing the required access control modularly. In particular, we follow the well-accepted Model-View-Controller (MVC) [3] architectural pattern and adopt the popular Apache Struts framework [1] to structure our Web applications.

Next, we apply AOP to devise a flexible implementation scheme that bridges these two ends. We developed our implementation scheme in two stages. In the first stage, we did an in-depth analysis of the structures of aspect code that we developed manually for implementing the forms of access control rule we employed. Such aspect code is classified into a few forms according to their internal structure. Basically, each access control aspect is divided into two parts: a generic part realized by an abstract aspect and a rule specific part realized by a concrete aspect. Indeed, these abstract aspects provide a solid basis towards building an adaptable mechanism.

In the second stage, we focus on how to automatically synthesize aspect code from access control rules. Given the abstract aspects derived in the previous stage, we only need to generate the parts that are rule-specific. Thus we prepared some aspect templates based on the derived aspect code structure to assist code generation. On the source side, in addition to the access control rules, we provide an application specification file that links the logical entities (data objects and operations) referenced in the rules to the real entities defined in the underlying application. Following the current practices of Web application development, we define both of the two input files in XML format and treat them as configuration files, one of each type for every application. Together with the pre-defined aspect templates, the two XML configuration files are processed by a rule translator into concrete access control aspects.



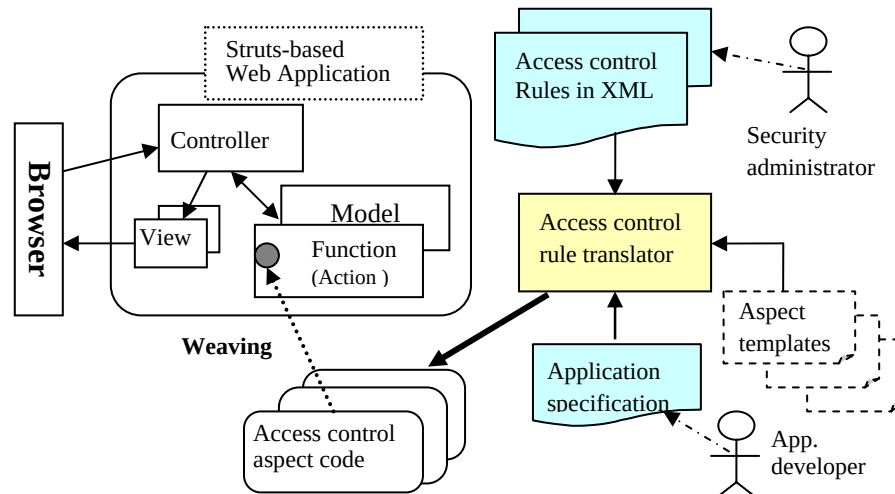


Figure 1. System architecture and mechanisms for adaptable access control

The generated aspect modules will then be compiled and woven into designated functional modules of the underlying Web application using standard aspect tools. At runtime, the aspect code will be executed like common functional code to enforce the required access control. Figure 1 illustrates the system architecture and mechanisms of our approach to providing adaptable access control. Security administrators are responsible for encoding the access control requirements of EMR using our rule format and the system developers need to supply the application specification file for linking the rules with underlying system to facilitate rule translation and aspect code generation.

4. Access Control Modeling and Rules

Access control, also known as *authorization*, is a process by which users, after being identified, are granted certain privileges to information, system functions, or resources. The step to identify a user is usually called *authentication*. Username and Password check is the most basic form of authentication, while digital certificates and biometrics are more advanced ways of authentication.

Role-based access control (RBAC) [8] is the most often cited guiding principle underlying all the proposed solutions to application-level access control. RBAC uses the role abstraction as its primary means of specifying access control requirements. However, as we have argued earlier, roles alone are not sufficient for our purposes; proper access control for EMR usually requires finer grained constraints that are derived from other user attributes and data contents. To handle such fine-grained requirements, we have taken a generic and an attribute-based scheme that is more expressive than RBAC for specifying access control requirements.

We model the interaction between a user and a Web-based EMR application as a sequence of access tuples of three elements: $\langle \text{user}, \text{action}, \text{data} \rangle$, indicating a user's request to conduct an action, such as view or

update, on a specific data component of EMR. The access control rules of an EMR system determine which access tuples are allowed and which must be denied. They are derived from the system's access control requirements.

Since EMR is our focus here, the data component of an access tuple plays the central role in specifying access control rules. It must not only facilitate the specification of access constraints of different actions on various parts of EMR but also lend to an easy integration with the underlying HIS. Here we follow the proposal of Taiwan Electronic Medical Record Template (TMT) [5], which aims to provide a suite of standard forms that will become the common basis for developing EMR systems in Taiwan. In TMT, all the data of EMR are categorized and divided into many inter-related documents called *forms*. Each TMT form has a specific name and its data are further grouped into a hierarchy of *modules*. Following TMT, our access control rules are designed to be *form-based* and treat each form as a tree organized into a hierarchy of modules. Specifically, our access control rules take the following abstract format:

```
<FormName, AuthType,
  ModPath1:
    Actions : Constraint;
  ...
  ...
  ModPathn:
    Actions : Constraint;
  ...
>
```

As authentication is required prior to authorization, we also make authentication type (AuthType) part of the rule; the type can be id/password (PWD), health digital certificate (HCA), or any other supported methods of user identification. The *FormName* refers to a specific form of TMT, and a *ModPath* is a tree path specifier referring to a particular module of the form that must be protected against any unauthorized actions. A special





module path called root, represented by “/”, refers to the whole form. The *constraint* is a Boolean expression which must be evaluated to true to grant the attempted execution of the associated actions. Inside a form, the accessibility of any module is defined in a cascading manner according to its position in the tree structured form:

- If the accessibility of the module is explicitly defined in the rule, use the defined accessibility;
- Otherwise use the accessibility of the nearest ancestor that is explicitly set.
- If no ancestor has accessibility explicitly defined, make the module always accessible.

To facilitate the expression of an access constraint without incurring the unnecessary details of the underlying system, we take an object-based approach and supply five generic objects: *User*, *Form*, *Data*, *Cxt*, and *App*, with various attributes that the constraint expression can refer to. Conceptually, the *Form* object and the *Data* object serve as the input and output of an action to execute, respectively. Typical attributes for the *User* object include user's name and roles in an organization. The attributes of the *Form* object include the arguments passed to the protected action, while the attributes of the *Data* object refer to the fields of any modules of a TMT form. The context object (*Cxt*) provides information stored on the server, such as the current time and the list of doctors on duty during off-hours. Like the context object, the application object (*App*) is also a global object which keeps various parameters related to access control yet specific to an application. Some typical attributes of the *App* object include duration of office hours and the IP address of machines dedicated to certain purposes. Clearly, the specific set of attributes for each object depends on individual application's needs. For example, the following rule dictates that, for TMT-Form-N, while all medical staff can read all parts of it, except module /A/B, only doctors can create and update it. Furthermore, for module A of the form, any updates must be done by the patient's doctor-in-charge or by doctors on duty during off-hours from certain dedicated machines. And the data in module /A/B can be read only during normal office hours.

```
<TMT-Form-N AuthType=PassWD
  path=/:
    CREATE, UPDATE:
      "Doctor" in User.roles
    READ: "Staff" in User.roles
  path=/A:
    UPDATE:
      User.name == Data.doctorInCharge or
      (User.machineIP in App.dedicatedMachines
      and User.doc_ID in Cxt.docOnDuty);
  path=/A/B:
    READ:
      Cxt.currentTime in App.officeHours
>
```

The above example illustrates the features of our access control rules. In particular, we can associate multiple kinds of attributes with the generic objects, which make it easy to model a multitude of requirements, from simple RBAC to sophisticated instance and module level constraints with little effort.

5. Aspect Templates for Access Control

We provide both authentication and authorization aspects. Due to space limitation, we shall focus on authorization aspects which are more complicated than authentication aspects.

Every module action that requires authorization will be associated with an access constraint, which in turn will be converted to real Java code and integrated into a proper access control aspect for enforcing the required control. As stated earlier, to facilitate the generation of access control aspects, an aspect is divided into two parts: generic part realized by abstract aspects and rule specific part realized by concrete aspects. The generic part captures the common code patterns one would develop manually to enforce an access control rule and forms the basis of an aspect template. After some analysis of the code pattern, we identified three most typical generic aspects, namely, *Precheck*, *PostfilterSingle*, *PostfilterCollection*. The availability of the data entities referenced in the constraint expression of a rule, such as user roles, action arguments and data contents, distinguishes these generic aspects. The pre-checking aspect handles the case when the constraint expression involves only the *User* and *Form* objects, whose attributes are all available before executing the protected function. In contrast, the post-filtering aspects are used for the cases when the constraint expression also involves attributes of the *Data* object, which are available only after executing the protected action. Furthermore, as the code structure for handling a single record retrieved by key-based queries is different from that of handling a collection of data obtained by ad-hoc queries, we provide two kinds of post-filtering aspects.

For spaces sake, Listing 1 displays only the code skeleton of the *PostfilterSingle* aspect and its associated template. Each descendant of such an aspect will enforce the access constraints associated with an action on a particular form. The specific association will be established by supplying the concrete pointcut definition in the template. The entry to execute this aspect is the *around* advice. This aspect first checks if certain constraint, *preCondition(..)*, is met before proceeding to execute the designated action. Afterwards, it applies the constraint, *postCondition(..)*, to decide whether the user is authorized to access this particular instance of patient data. Both of these two conditions are derived from the access constraints specified for the *root module* of the underlying TMT form. Finally, before returning it to the user, the aspect needs to filter out any modules of this form that are not accessible to the user. This is accomplished by invoking the *moduleFilter(..)*





method, which will traverse the tree-structured form to perform the filtering task.

Listing 1: The *PostfilterSingle* aspect and its template

```
public abstract aspect PostFilterSingle
    extends Authorization {
    abstract pointcut pc;
    abstract boolean
        preCondition(HttpServletRequest req);
    abstract protected boolean
        postCondition(HttpServletRequest req);
    ActionForward forwardToErrorPage(..){...}
    void moduleFilter(... req){ //template}
    ActionForward around(...)
        :pc(mapping,form,request,response) {
        if( preCondition(request) ){
            ActionForward forward =
                proceed(mapping,form,request,response);
            //instance (form) level filtering
            if( postCondition(request) ){
                // module-level filter
                moduleFilter(request);
                return forward;
            } //end post
        } //end pre
        return //access denied
            forwardToErrorPage(request,mapping);
    }
}
//template to be instantiated
public aspect <aspectName>
    extends Postfilter {
    pointcut pc(..): <pointcut> && args(..);
    <_Library> // library functions
    Boolean preCondition(... req)
    { <Constraint from the root module> }
    boolean postCondition(... req)
    { <Constraint from the root module> }
    void moduleFilter(... req) // visitor pattern
    { <module-level constraints> }
}
```

Since our access control rules are specified in terms of the TMT form, we have a rather stable structure of data to traverse for conducting the module filtering work. Therefore, to facilitate the code generation of the module filter part, we have chosen to implement the module filters using the famous visitor pattern [3], which provides the ability to add new operations to existing object structures without modifying those structures. This advantage of visitor pattern is exactly what we expect our module filter to achieve: traversing various tree-structures and performing different operations, such as masking a module, for access control purpose.

6. Synthesizing Access Control Aspects

This section sketches our scheme for synthesizing access control aspects. Figure 2 illustrates the major steps of our aspect synthesizing process. Besides aspect templates, there are two configuration files that serve as the input to our aspect synthesizer: the access control rules and application specification. Following the current practice of designing configuration files for Web-based

applications, both input files are formatted using an XML schema, respectively. After loading these two files and perform the required sanity check, the synthesizer will enter a loop to process every enforce-point specified in the access control rules.

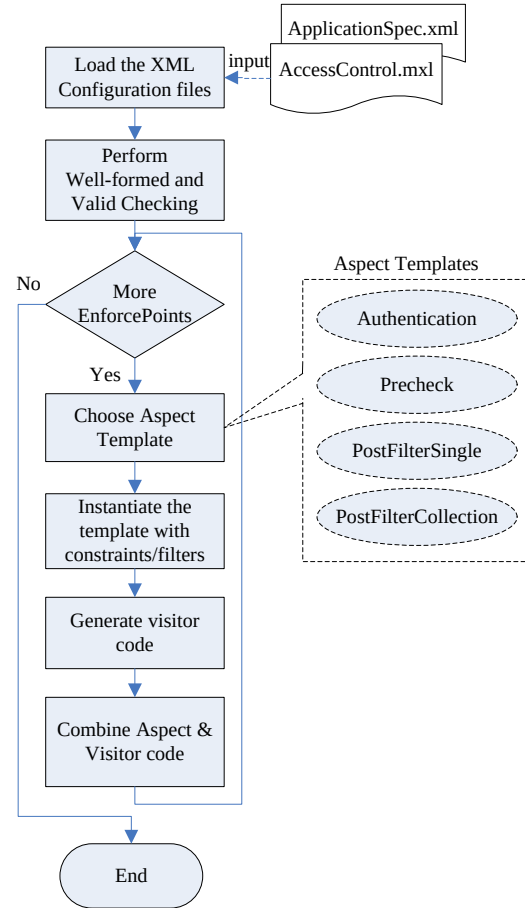


Figure 2. The major steps of the aspect synthesizer

An enforce-point corresponds to an action that needs to be protected for the *root module* of a particular TNT-form. It is the main item specified in the access control XML configuration file. For example, the action of creating a *DischargeNote* form is authorized to only users with a certain role, hence it will be represented as an enforce-point. The constraint associated with that action will become the pre-condition and post-condition of the matching enforce-point. Together with other related module-level actions and constraints, an aspect that inherits from a proper aspect template will be synthesized to realize the access control required by a particular enforce-point.

Figure 3 highlights the major parts of the XML schema for specifying the access control rules. In encoding the form-based access control rules using our schema, enforce-points of the same authentication type requirement will be grouped together under an enforce-domain, as different authentication types imply different security levels. Besides, to facilitate sharing the module-level constraints across module actions, we have prepared a separate group of elements called *formFilters*,





which comprise a list of modules that in turn encodes the associated action and constraint pairs. Appendix A lists the XML version of the access control rules we presented in Section 4.

As mentioned in the Section 3, the aspect synthesizing task is greatly facilitated by an application specification file that supplies the definitions of the mappings required for linking the entities and association relations referenced in the access control rules to those real entities of underlying EMR system. Figure 4 outlines the main structure of the XML schema of the application specification file. In particular, we group the required mappings into five sections: *EnforcePointMapping*, *AttributeGroupMapping*, *FormMapping*, *AuthTypeMapping*, and *FunctionMapping*.

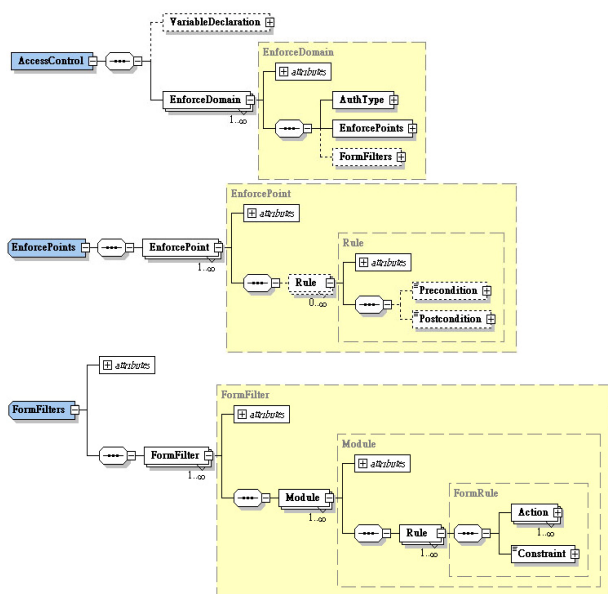


Figure 3. The structure of the XML schema for access control rules

The *EnforcePointMapping* is the basic mapping. It maps the generic objects to their concrete counterparts in an application and links every enforce-point to the real code module that performs the specified action. The *AttributeGroupMapping* maps the fields of various objects referenced in the access control rules to the actual classes that define them. The *FunctionMapping* specifies where the actual implementations for the functions, such as set membership and equality operators, used in access control rules are defined. The *AuthTypeMapping* simply links the supported authentication modules to those cited in the access control rules.

Finally, the *FormMapping* is essential to our access control scheme for hierarchical modules. Each TMT form will have a *FormMapping*. It specifies the enforce-points that correspond to the form-level actions and maps all the module paths of the form to actual application classes we used to store the corresponding EMR data, including those modules without any access

constraints. Together with the information specified in the *EnforcePointMapping*, the aspect synthesizer can easily link any protected form action to the real Java method in the EMR application.

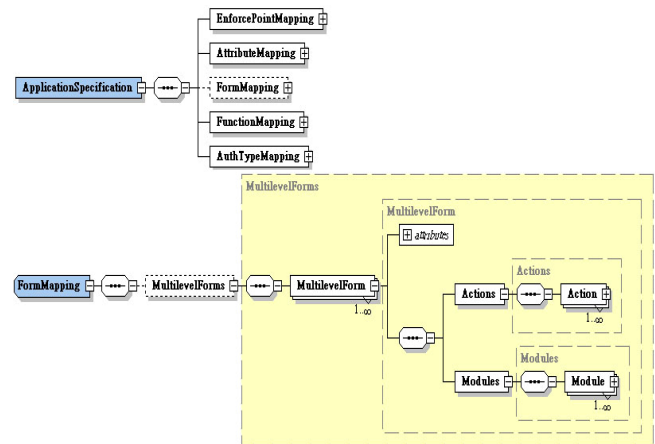


Figure 4. The partial structure of the XML schema for application specification

With such an application specification file, security administrators are free of many details of the underlying EMR application when specifying the access control rules. Changes to the EMR applications that are not related to access control can be accomplished without any undesirable effects on our aspect synthesis task. Moreover, the synthesis task is also greatly facilitated by this specification file.

7. Prototype Implementation

To demonstrate our ideas, we have built a Web-based prototype implementation to demonstrate our approach. As stated earlier, our prototype is built on top of the popular Java-based Struts framework [1], and we used AspectJ as our aspect implementation language. Our aspect synthesizer is written in Java and we have made intensive use of JAXB library to handle the parsing and sanity checking of XML configuration files.

Figure 5 shows a screenshot of our prototype for the discharge note of TMT form. Here we mask out the chief complaint field (*****) and make fields that are read-only to the current user displayed in gray. These effects are mainly achieved by the two visitors included in our aspect code. They are filter visitor and update visitor that will traverse the tree representation of a discharge note and conduct the needed access control operations at selected modules according to the paths specified in its access control rules.

It is worth mentioning that all these field protecting and masking effects are performed by the aspect code and visitors with the help of a generic JavaScript routine on the client side. We have not modified any of the functional modules of discharge note, not even the view part for displaying output results.





>>> 病歷列表 >> 出院病歷單 >> 編輯出院病歷單

■ 出院病歷單					
* 醫院名稱	台北醫學大學附設醫院		* 醫院代號	000000001	
個人基本資料					
* 姓名	王明月	* 病歷號碼	000000001	年齡	30
* 身分證字號	A123456789				
出入醫院資料					
轉入醫院地址	台中縣大里市東榮路 483號	轉入醫院流水編號	000000005		
入院日期	2005/08/12	入院科別	家庭醫學	入院床號	20
轉科日期	2005/08/05	轉科科別	耳鼻喉科	轉科床號	50
出院日期	2005/08/10	出院科別	耳鼻喉科	出院床號	52
■ 疾病診斷模組					
入院時					
序號	疾病名稱	編碼系統	編碼	備註	
1	耳炎症	0001	0001		
出院時					
序號	疾病名稱	編碼系統	編碼	備註	
1	內聽道病變	0002	0002		
■ 主訴模組					
描述	*****				
■ 出院時管況模組					
出院情況	治癒出院				

Figure 5. Screenshot of the prototype implementation for the discharge note of TMT

also plan to develop a rule analyzer for detecting such inconsistencies while synthesizing the aspect code.

8. Conclusions and Future Work

In this paper, we have presented an aspect-oriented approach to providing adaptable access control for Electronic Medical Records on Web-based systems. We followed the Taiwan Electronic Medical Record Template to design our access control scheme so that our scheme will integrate easily to future HIS systems based on this standard. Our access control modeling scheme can satisfy a wide range of requirements with different granularity. By employing aspect-oriented technology, we have obtained a highly modular implementation of fine-grained access control and the aspect code for enforcing access control is automatically synthesized. Future changes of access control rules can also be effectively realized through these mechanisms without actual coding. This will not only improve the system's modularity but also make the task of enforcing comprehensive access control more adaptable. We argue that our scheme has achieved an adequate balance between expressiveness of access control and runtime efficiency. We have also built a Web-based EMR prototype implementation using AspectJ to demonstrate our approach.

We plan to further explore this line of study along a few directions. First, besides the authentication type specified for a TMT form, we shall extend our access control rules to enable stricter authentication for individual modules. Second, we plan to design and implement a rule development tool that helps security administrator convert the abstract access control rules into the XML format. Currently, this is done manually, which is not only error-prone but also difficult to maintain. Third, once the numbers of rules increase to a degree, it is very likely that we will mistakenly create some inconsistencies between related rules that may creep into the aspect code without notice. Therefore, we

9. References

- [1] The Apache Struts Web Application Framework: <http://struts.apache.org/>
- [2] K. Chen and D.W. Wang. "Toward Configurable Access Control for Healthcare Information Systems", *MIST 2004*.
- [3] Gamma, Helm, Johnson and Vlissides, *Design Patterns*, Addison-Wesley, 1995.
- [4] HIPAA. <http://www.cms.hhs.gov/hipaa>.
- [5] W.S. Jian et al., The Development of Taiwan Electronic Medical Record Template, Draft. Templates download : <http://emr.doh.gov.tw/html/kaisya-annai.html>
- [6] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin (1997), "Aspect-Oriented Programming," in *ECOOP '97*, LNCS 1241, pp. 220-242.
- [7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold (2001), "Getting Started with AspectJ", *Comm. of ACM*, vol. 44, no. 10, pp 59-65.
- [8] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman (1996), "Role-based access control models," *IEEE Computer*, 29(2):38-47, 1996.
- [9] T. Verhanneman, L. Jaco, B. De Win, F. Piessens, and W. Joosen (2003), "Adaptable Access Control Policies for Medical Information Systems," *Proc. of Distributed Applications and Interoperable Systems*, 2003, LNCS 2893, pp. 133-140.
- [10] B. De Win, F. Piessens, W. Joosen and T. Verhanneman (2002), "On the importance of the separation-of-concerns principle in secure software engineering," *ACSA Workshop on the Application of Engineering Principles to System Security Design*, pp. 1-10, Boston, Massachusetts.





Appendix: An example of AccessControl.xml file

```

<AccessControl xmlns=http://www.cs.nccu.edu.tw/plsm/2005/AccessControl
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
  <EnforceDomain name="EMR">
    <AuthType value="PWD"/>
    <EnforcePoints>
      <EnforcePoint name="WriteTMTFormN">
        <Rule>
          <Precondition>
            _Lib.contains($User.getAttr("Role"), "Doctor")
          </Precondition>
        </Rule>
      </EnforcePoint>
      <EnforcePoint name="CreateTMTFormN">
        <Rule>
          <Precondition>
            _Lib.contains($User.getAttr("Role"), "Doctor")
          </Precondition>
        </Rule>
      </EnforcePoint>
      <EnforcePoint name="ReadTMTFormN">
        <Rule>
          <Postcondition>
            _Lib.contains($User.getAttr("Role"), "Staff")
          </Postcondition>
        </Rule>
      </EnforcePoint>
    </EnforcePoints>
    <FormFilters>
      <FormFilter form-name="TMTFormN">
        <Module name="A">
          <Rule>
            <Action name="write"/>
            <Constraint>
              _Lib.eq($User.getAttr("Name"), $Data.getAttr("doctorInCharge")) or
              ( _Lib.contains( $App.getAttr("dedicatedRooms"), $User.getAttr("machineIP") and
                _Lib.contains( $Cxt.getAttr("docOnDuty"), $User.getAttr("doc_ID")) )
            </Constraint>
          </Rule>
        </Module>
        <Module name="B">
          <Rule>
            <Action name="read"/>
            <Constraint>
              _Lib.contains($App.getAttr("officeHours"), $Cxt.getAttr("currentTime"))
            </Constraint>
          </Rule>
        </Module>
      </FormFilter>
    </FormFilters>
  </EnforceDomain>
</AccessControl>

```

