Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 266 (2010) 77–92
www.elsevier.com/locate/entcs

Design and Verification of a Trustable Medical System

Xijiao Xiong[1] Jing Liu[2,4]

Shanghai Keylab of Trustworthy Computing
East China Normal University
Shanghai, 200062, P.R.China

Zuohua Ding [3]

Center of Math Computing and Software Engineering
Zhejiang Sci-Tech University
Hangzhou, 310018, P.R.China

Abstract

Developing an advanced medical informatics system is a grand challenge in the 21st century. In this paper, we construct and analyze a trustable medical system by Refinement Calculus of Object Systems (rCOS) in a model-driven development process. Our method greatly improves the dependability and efficiency of the complicate system. This implies that the formal techniques developed in rCOS can be integrated into a model-driven development process. For the verification, a tool, called UPPAAL, is used to ensure the safety and correctness of the medical system. Our result suggests a way to corporate design and verification in system development process.

Keywords: formal method, rCOS, UPPAAL, telemedicine

1 Introduction

Recently, telemedicine is becoming more and more important, which combines the computer technology, internet technology and communication technology. It makes medical service more efficiency and convenient, and thus reduces patients' financial costs. It also provides a platform for medical experts and doctors to study and communicate. Moreover, the use of a trustable medical systems in telemedicine practice can significantly improve the efficiency and quality of health care. At the Grand

[1] Email:yuyaoxj@hotmail.com
[2] Email:jliu@sei.ecnu.edu.cn
[3] Email:zuohuading@hotmail.com
[4] Corresponding author

doi:10.1016/j.entcs.2010.08.050

Challenges Summit of U.S. National Academy of Engineering (NAE), advanced health informatics is proposed as one of the grand challenges for Engineering in the 21st Century. A major goal of advanced health informatics is to develop trustable systems that can offer relevant decision support to clinicians, patients and archive medical research[11].

A reliable and effective electronic trustable medical system (TMS) of clinical use requires both a structured and standardized Electronic Health Record (EHR) and a reliable scientific evidence based medicine (EBM). EHR systems are used to produce standardized information for the trustable medical system. The Evidence-Based Medicine Guidelines (EBMG) database [14], including about 1000 clinical guidelines, are developed since 1988. A MDA-based approach was used to automatically produce guidelines for TMS [13]. However, information sharing over regional, national, or global networks makes the situation more complicated because of the diverse of computer systems and different kinds of data recording rules [11]. Most decision support systems such as DXplain [10] do not support automatic data transfer between the EHR and TMS. Isable [6] allows users to submit the data from pre-assigned fields of EMR to Isable, but it only gives the doctors a list of possible diseases and can not give relevant reminders, suggestions and guidelines. There is no mechanism of analyzing the patient's record in different formats, and thus there are no corresponding reminders or guidelines that can be provided automatically.

To solve the above issue, our approach is firstly constructing the models in the Refinement Calculus of Object Systems (rCOS) [7] and then verifying in the real time the safety properties by tool UPPAAL [9]. TMS, along with EHR system, can be integrated into telemedicine systems and provides decision support service for decisions makers such as experts, doctors of different hospitals in different countries.

We choose rCOS to develop TMS because it supports both static structural and dynamic behavioral refinement of object-oriented system and can effectively reduce the complexity of system by separating concerns. rCOS is founded on a well-formed semantic model with a refinement calculus [7,15], including a precise notation for object-oriented and component based design, thus the formal techniques, including tools such as UPPAAL [9] can be applied in the development process to ensure the consistence between different aspects or views and correctness of the system.

This paper is organized as follows. In the next section, we introduce the class model and interaction model in rCOS. In section 3, we present our approach to construct TMS by rCOS. In section 4, we verify the system by UPPAAL. Finally we conclude in Section 5.

2 Class Model and Interaction Model

rCOS is developed to support model-driven development methods. It provides with multi-view modeling environment and combines object-oriented and component-based design and analysis techniques[17]. It effectively reduces the complexity of system and ensures the consistency and correctness of the system. In rCOS, a component is an essential concept, which is an implementation of a contract. We

use the class diagram to describe static structure of a component, and sequence diagram to describe the interaction protocol of the contract, respectively.

2.1 Class Model

To support object-oriented design of components, types of fields of component interfaces can be *classes* and thus the values of these fields are objects. In rCOS, a class can be specified in the following format[17]:

```
class      C [extends D] {
attr       T x = d , . . . , T x = d
method     m(T in; V return) {
           pre:   p ∨ . . . ∨ p
           post:    ∧ (R; . . . ;R) ∨ . . . ∨ (R; . . . ;R)
                    ∧ . . . . . .
                    ∧ (R; . . . ;R) ∨ . . . ∨ (R; . . . ;R)
             }/* method m */
method     m(T in; V return) { . . . . . }
             . . . . . .
invariant  Inv }
```

2.2 Interaction Model

In rCOS, a black box behavior model of the interface of a component is defined as a contract and currently it only considers components in the application of concurrent and distributed systems[17], and a contract is a tuple $C = (I,Q,S,P)$, where

- I is an interface, denoted by *C.IF*,
- Q is a design $\vdash R \wedge wait'$ which initializes the values of fields in *C.IF*, denoted by *C.init*,
- S specifies each method *m(in; out)* in *C.IF*, denoted by *C.spec*,
- P is a set of traces of the events over methods in *C.IF*, denoted by *C.protocol*.

The *protocol* of the *contract* describes the interaction between the actors and the system. The interaction can be illustrated by a UML *sequence diagram*, and the dynamic flow of control and synchronization can be represented by a UML *state diagram*.

3 Design of Trustable Medical System

Trustable Medical System (TMS) is a medical system that can be integrated into normal clinical system or telemedicine system. It combines medical knowledge with individual patient data and provides patient-specific guidelines and reminders for physicians and other health care professionals. In our paper, we concentrate on the system model of TMS, and leave other detailed technologies such as data conversion or data access for the further work.

3.1 System Overview

TMS can be used in clinics and hospitals of multiple countries. It transfers patient data to EHR system to produce electronic forms, and then receives structured patient data from EHR system and provides reminders and guideline links for the end-users. The end-users can also directly select an existed EHR for decision support in EHR mode. In addition to real-time use, the decision support rules can also be run in patient populations and provides in-time decision support service. Furthermore, TMS is a platform-independent service, which can be integrated into any EHRs containing structured patient data. The overview of the TMS is shown in Fig 1. TMS consists of the following subsystems:

- A *Work station* for end-users to input patient data and receive reminders.
- A *EHRDatabase* to store electronic health records.
- A *EHRGenerator* to generate corresponding EHR from the patient data inputted from the *Work station* and to send the EHR to *ScriptInterpretor*.
- A *ScriptInterpretor* to interpret the EHR from *EHRGenerator* and create the standard variables and objects used in the scripts.
- A *ScriptDatabase* to store scripts.
- A *EvidenceDatabase* to store evidences for decision support.
- A *GuidelineDatabase* to store guidelines for decision support.
- A *DecisionSupporter* to receive scripts from *ScriptInterpretor* and returns the reminders to the *Work station*.

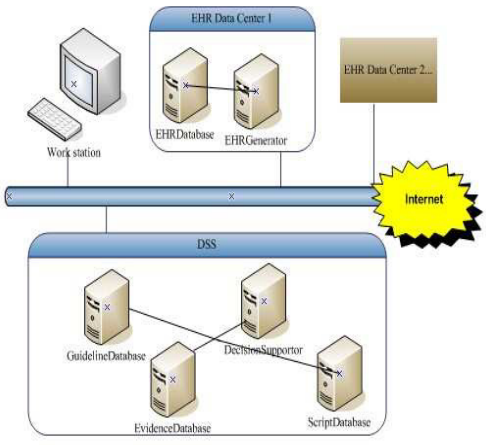

Fig. 1. Framework of TMS

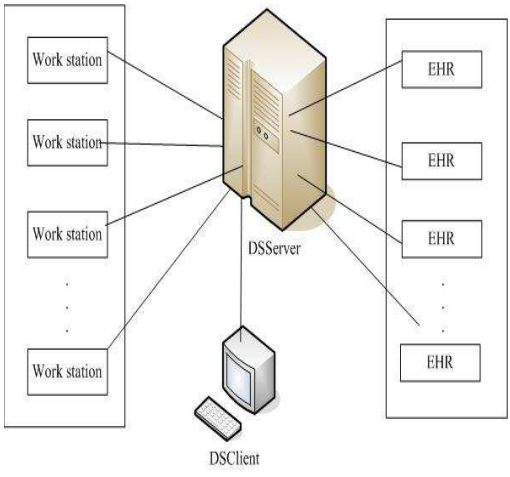

Fig. 2. Overview of the System

In fact, the whole system can integrate a number of work stations for groups of end-users and a network of EHR Data Centers for EHR systems of different criterion (See Fig 2). In our paper, we only consider the system with an EHR system.

3.2 Requirements of TMS

Requirement model can be described by a sequence diagram and a conceptual class diagram. These two diagrams describe the dynamic properties and static structural properties of the system respectively. The design process consists of a set of use cases, but we only give some typical use cases due to space limitation.

3.2.1 Use case UC 1: Decision Support

Scenario of UC 1 It describes how the system interacts with the actors by a scenario as follows:

1. When the doctor enters the decision support panel, a new decision support is initiated.
2. The doctor can select the language and the system gives corresponding interface, otherwise the system will display the default interface.
3. The doctor selects the model case and enters EHR information such as General Data, EHR Event, Diagnoses, Medication and Laboratory Result in the page, otherwise the system gives the default mode case.
4. The system sends the patient data to the EHR Data Center to generate an EHR item. When it is in EHR mode, doctor only needs to select an existed EHR item. Then after the generated or selected EHR item is analyzed by system, reminders, evidences and guideline links are provided to the doctor.
5. If the system is not in the EHR mode, the doctor can choose to save the EHR item. The process of decision support is completed.

Model of UC 1 The use case is modeled by the contract of the provided interface of a component: *SupportDecision*, with *DSPanelIF* as its provided interface.

```
component SupportDecision{
    provide interface DSPanelIF {
      public enableEHR();
      public disableEHR();
      public startDS();
      public enterLanguage(int lan);
      public enterEHR(XMLDoc xd , int mcId);
      public selectEHR(int ehrId);
      public fetchReminder(EHRItem ehr; Reminder rem);
      public saveEHR(EHRItem ehr);
    }
  }
```

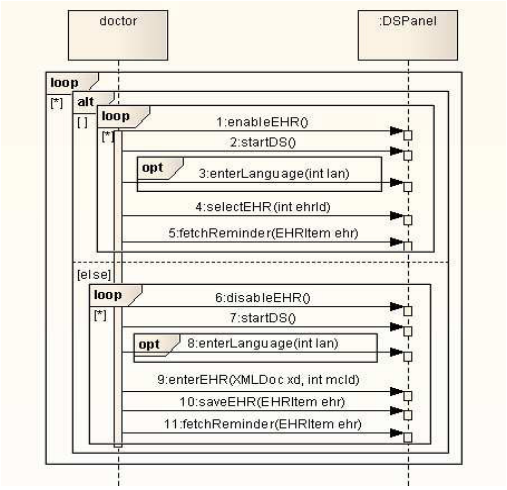

Fig. 3. Sequence Diagram

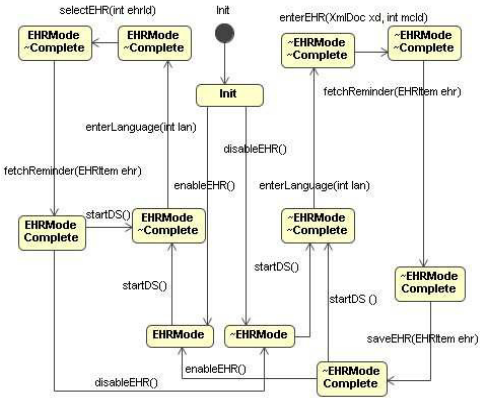

Fig. 4. State Diagram

The sequence diagram in Fig 3 represents the protocol of *DSPanelIF*, and the state diagram in Fig 4 models the dynamic flow of control. Besides,the functionality and invariant of the methods of the interface are specified by pre and post-conditions as follows.

class *DSPanel* **implements** *DSPanelIF::*

invariant $ehrDB \neq null \wedge ehrDB.catalog \neq null$
$\wedge\ dsStore \neq null \wedge dsStore.scripts \neq null$
$\wedge\ dsStore.evids \neq null \wedge dsStore.gllinks \neq null$
$\wedge\ (EHRMode = true \vee EHRMode = false)$

method *enableEHR()*
pre: *true*
post: *EHRMode' = true*

method *disableEHR()*
pre: *true*
post: *EHRMode' = false*

method *startDS()*
pre: *true*
post: *rem' = Reminder.new(empty/evids, empty/gllinks);*
lan = 0; mcId = 0
*/*a reminder is created and its evidence list and guideline link list initialized, and default language and mode case are set*/*

method *enterLanguage(***int** *lan)*
pre: *true*
post: *language' = lan*

method *enterEHR(XMLDoc xd,* **int** *mcId)*
pre: *true*
post:*ehrItem' =EHRItem.New(xd/xmlDoc, mcId/mc, clock.date()/date)*
*/*a new EHR item is created*/*

method *selectEHR(***int** *ehrId)*
pre: *ehrDB.catalog.find(ehrId) ≠ null*
*/*EHR item with id ehrId exits in the EHR*/*
post: *ehrItem' = ehrDB.catalog.find(ehrId)*

method *fetchReminder(EHRItem ehr; Reminder rem)*
pre: *dsStore.scripts.findBymc(ehr.mc) ≠ null*
*/*the script to deal with this kind of mode case exists*/*
post:*rem' = dsStore.scripts.findBymc(ehr.mc).run()*
*/*reminder for the EHR returns for the decision support*/*

method *saveEHR(EHRItem ehr)*
pre: *true*
post:*ehrDB.catalog.add(ehr)*
*/*EHR item is add to EHR catalog*/*

The data types of fields of the interface and their relations are represented by the class diagram in Fig 5.

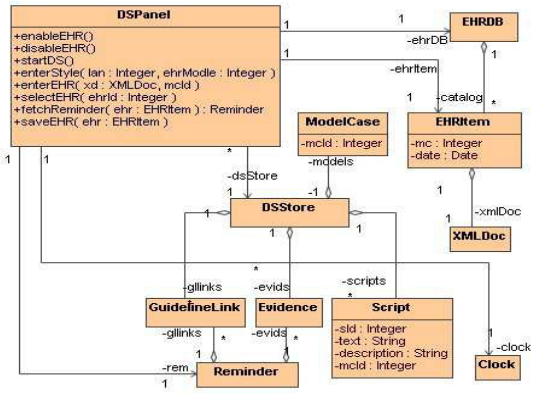

Fig. 5. Conceptual Class Diagram

3.2.2 Use case UC 2: Script Management

This use case carries out updating to the script, where updating includes changing script text and adding a new script. As this use case is simple and allows any sequence of invocations of these operations, we omit its sequence diagram and class diagram and just give the functionality of the methods of the use case.

component *ScriptManagement* {
  provide interface *ScriptPanelIF* {
    public *changeScriptText(***int** *sId,* **String** *newText);*
    public *addscript(***int** *sId,* **String** *sText,* **String** *desc,* **int** *mcId);* }
  class *ScriptPanel* **implements** *ScriptDeskIF* {
    protected *DSStore dsStore;*
    public *changeScriptText(***int** *sId,* **String** *newText)* {
      pre: *dsStore.scripts.find(sId)* $\neq$ *null*
      post: $\forall$ *s* $\in$ *scripts* $\cdot$ *(***if** *s.sId = sId* **then** *s.text' = newText)* }
    public *addscript(***int** *sId,* **String** *sText,* **String** *desc,* **int** *mcId)* {
      pre: *true*
      post: *dsStore.scripts.add(Script.New(sId, sText, desc, mcId))* }
  }
}

3.2.3 Model integration and global constraints

Two use cases are both modeled by a contract of the provided interface of a component, thus they seem two separate closed components. But after analyzing the relation between these two use cases, we find that they share the same object: *DSStore dsStore*. They can be composed into one component only if object *dsStore* in two components are the same:

component *DSComp* {
  invariant $\exists$ *DSStore ds* $\cdot$ ds = SupportDecision.dsStore = ScriptManagement.dsStore }

Then these two components can be integrated into a bigger one:

$$DSComp =_{df} SupportDecision \parallel ScriptManagement$$

This kind of constraints should be established when integrating system and constructing components. [17] proposes an apporach to guarantee these constraints by using a global set of classes to represent all the use cases, and once a new use case is captured, new class, together with its attributes and associations will be added to this set, thus the consistency of the class definitions can be guaranteed.

3.3 Logical Design Model of TMS

In this section, we will build the logical design model of TMS by refinement rules of rCOS, especially by expert pattern[17]. We focus on the design of some methods in **UC 1** and **UC 2** and use the convention *Class C::m(){c}* to denote the method *m()* of class *C* and the specification statement *c*.

Method *startDS()* The precondition of *startDS()* is trivial, thus we only need to consider its postcondition. The specification of the postcondition means to create a new reminder. Following the expert pattern, the responsibility to create the reminder can be delegated to class *Reminder*:

Class *DSPanel:: startDS()* {
rem:= Reminder.New(); lan:= 0; mcId:= 0 }
Class *Reminder::Reminder()* {
evids := Set(Evid).New();
gllinks := Set(gllinks).New() }

The specification uses the constructor method of class *Reminder* which takes no input parameters. It initializes the attributes of the class: *evids* and *gllinks*. We can see that the type of *evids* is a set of Evidence and the type of gllinks is a set of *GuidelineLink*. For a set of classes S(T), we use the Java notation *s := S(T).New()* and its rCOS semantics *true* $\vdash$ *s'*$= \emptyset$ for the creation of a set of objects.

Method *selectEHR()* In its specification, there is a non-trivial precondition: *ehrDB.catalog.find(ehrId)* $\neq$ *null*. For it, we use the refinement schema (**PP**):

(**PP**) *m() pre:p; post:R* $\sqsubseteq$ *m()* {**if** *p* **then** *R* **else throw expception** *(p)*}

where *throw exception(p)* is the specification when the precondition fails to hold.

Applying the expert pattern to the precondition of *selectEHR()*, we can delegate the responsibility for finding the EHR Item to the class *EHRDB*. Thus we need to define the following methods:

Class *DSPanel::* *findEHR(int ehrId; EHRItem return)*
{*return := ehrDB.findEHR(ehrId)* }
Class *EHRDB::* *Set(EHRItem) catalog;*
findEHR(**int** *ehrId; EHRItem return)* {
{*return := catalog.find(ehrId)* }
Class *Set(EHRItem):: find*(**int** *ehrId; EHRItem return)*
{*return := find(ehrId)* }

Here, we assume that the method *find(ehrId)* of *Set(EHRItem)* implements the specification that returns the model case with Id *ehrId*.

For the postcondition of *selectEHR()*, we let *setEHR(int ehrId)* denote the

method that realizes the postcondition of *selectEHR()* when the preconditions holds. We still apply the expert pattern to refine the specification of postcondition using sequential compositions.

```
Class DSPanel::    setEHR(int ehrId; EHRItem ehrItem) {
                    ehrItem := ehrDB.findEHR(ehrId) }
Class EHRDB::      Set(EHRItem) catalog;
                    findEHR (int ehrId; EHRItem return)
                   {return := catalog.find(ehrId) }
Class Set(EHRItem):: find(int ehrId; EHRItem return)
                   {return := find(ehrId) }
```

Finally, we apply the refinement **(PP)** and obtain the following design:

```
Class DSPanel::    selectEHR(int ehrId) {
                   if findEHR(ehrId) ≠ null then setEHR(ehrId)
                   else throw exception (findEHR(ehrId) ≠ null) }
```

Method *updateScriptText()* Based on the refinement rule **(PP)** and expert pattern, for the precondition, we define following methods:

```
Class DSPanel::    findScript (int sId; Script return)
                   {return := dsStore.findScript (sId) }
Class DSStore::    Set(Script) scripts;
                    findScript(int sId; Script return)
                    {return := scripts.find(sId) }
Class Set(Script):: find (int sId; Script return)
                    {return := find(sId) }
```

We see that the postcondition involves quantifications over elements of sets:

$$\forall\ s \in scripts \cdot \ (\textbf{if}\ s.sId = sId\ (\textbf{then}\ s.tex' = newText)$$

For the specification of the form $\forall\ T\ o \in s \cdot statement(o)$, where s is a set of type set(T), we use universal quantification pattern(**UQP**):

```
(UQP):  ∀ T o ∈ s ·  statement(o) ⊑ Iterator i := s.iterator();
                                    while i.hasNext() { T o := i.next(); statement(o) }
```

That is to say, we should define the semantics of the "Java" statements on the right hand side of the above refinement in rCOS as $\forall\ T\ o \in s \cdot\ statement(o)$, where statement(o) is an rCOS statement. Now applying the expert pattern and (**UQP**), we define the update methods in these two classes.

```
Class ScriptDesk::   changeScriptText(int sId, String newText)
                     {dsStore.updateScriptText(sId, newText)}
Class DSStore::      updateScriptText(int sId, String newText) {
                     Iterator i := scripts.iterator();
                     while i.hasNext()
                     { Script s := i.next();
                     if s.sId=sId then s.text'=newText } }
```

Finally, we obtain the following design:

```
Class ScriptDesk::   updateScriptText(int sId, String newText) {
                     if findScript(sId) ≠ null
                     then changeScriptText(sId, newText)
```

else throw exception *(findScript(sId) ≠ null)* }

The design of other operations can be worked out with expert pattern in a similar way. The design sequence diagram of the logical model is shown in Fig 6 and class diagram is shown in Fig 7.

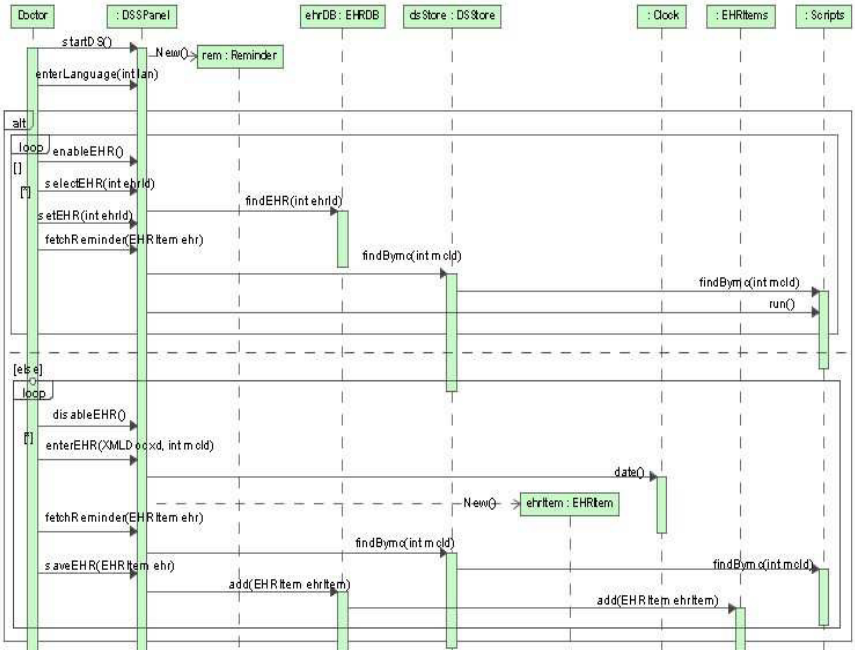

Fig. 6. Design Sequence Diagram

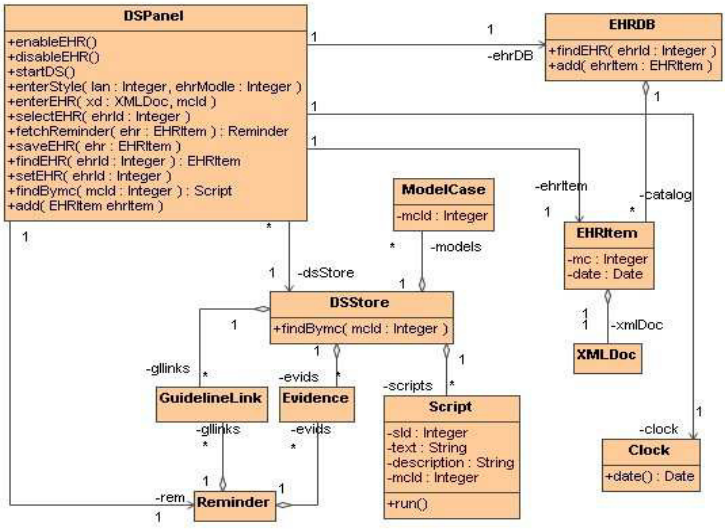

Fig. 7. Design Class Diagram

3.4 Component Architecture Model of TMS

In this section, we design the component-based architecture by decomposing some use case components into a set of components plugged together and composing a number of simple components into a larger component.

We can identify object *dsStore* and *ehrDB* as a unique and permanent object, which provides much functionalities to the other objects. Moreover, there is a global constraint for *dsStore* on the two use cases that they both share a single *dsStore* object. Thus we can make this object into a component called *DSDecision*, together with its aggregated object *scripts*, *evids*, *gllinks* and *models*. We use *DSDecisionIF* to denote the interface of *DSDecision*, which consists of those methods of use case **UC 1** in the design sequence diagram (Fig 6) through which the component *DSPanel* invokes on the objects and now are bundled in *DSDecision*. The temporary objects *rem*, *ehrItem* and *language* form an open component, denoted as *DecisionSupporter*, which has the provided interface *DSPanelIF* and required interfaces *DSDecisionIF* and *EHRGeneratorIF*, which will be referred later. Similarly, object *ehrDB*, together with its aggregated object *catalog* and *catalog*'s aggregated object can be used to construct a component called *EHRGenerator* which provides interface *EHRGeneratorIF* for *DecisionSupporter*. Besides, the clock can be considered as a component with well-known implementation. Then we have

$$SupportDecision =_{df} DecisionSupporter \parallel DSDecision \parallel EHRGenerator \parallel Clock$$

Similarly, the component *ScriptManagement* can be decomposed into *ScriptHandler* and *DSScript*. *ScriptHandler* has provided interface *ScriptPanelIF* and required interface *DSScriptIF* provided by *DSScript*.

The object *dsStore* in component *DSDecision* and *DSScript* are the same and we compose the two component into a bigger component *DSSever* with interfaces *DSDecisionIF* and *DSScriptIF*:

$$DSSever =_{df} DSDecision \parallel DSScript$$

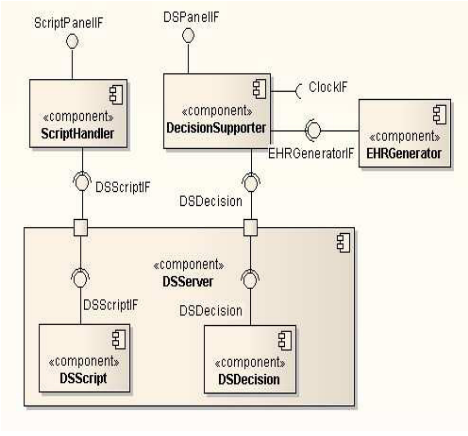

Fig. 8. Logical Component-Based model

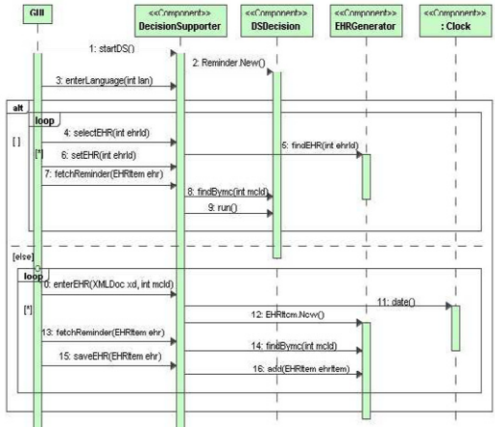

Fig. 9. Component Sequence Diagram

The UML component diagram in Fig 8 is used to represent the component-based

architecture of TMS. The design sequence diagram of **UC 1** can be transformed into component sequence diagram in Fig 9.

4 Verification of Real-Time Property in TMS by UPPAAL

In order to relieve doctors from suffering information overload when treating specific patients, the TMS should provide "just in time, just for me" advices at the point of care. Besides, as EHR Item usually contains multiply DICOM (Digital Imaging and Communications in Medicine) which needs to be transmitted from PACS (Picture Archive and Communication System) in telemedicine center to partner clinics, there will be delay for the system to make the decision for specific case.

To ensure that the end-users obtain decision support service in time, the real-time and safety properties of system should be guaranteed. Our approach is to construct a timed automata model for the process of **UC 1** and then verify the model with UPPAAL [9], a tool for modeling, simulation and verification of real-time system.

4.1 Specification of DICOM Transfer Problem

Based on component sequence diagram in Fig 9, which omits the time information, we add a tag value and a constraint to respectively describe the time initialization and the time constraint the events should conform to in the diagram. A *Controller* and *DICOMDB* are also introduced to the diagram to control and transfer the DICOM, respectively. The whole process of the transfer of DICOMs with time constraint can be described as follows:

1. The *EHRGenerator* informs the *Controller* to get corresponding DICOMs after *DecisionSupporter* asks for an EHR item.
2. The *Controller* informs *DICOMDB* to transfer the first DICOM to *EHRGenerator* within 200 ms. Here we assume the *DICOMDB* deals with one DICOM at a time and it won't accept another command for DICOM transfer.
3. When all the DICOMs are transferred from *DICOMDB*, the EHR item is returned to *DecisionSupporter*.

According to the informal description above, we obtain the sequence diagram with time constraints in Fig 10.

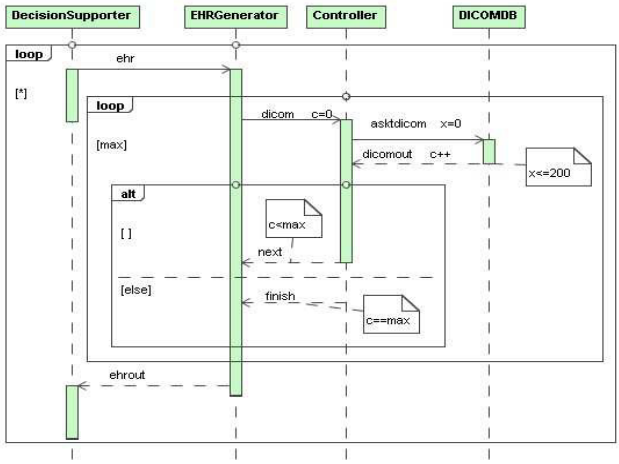

Fig. 10. Sequence Diagram with Time Constraints

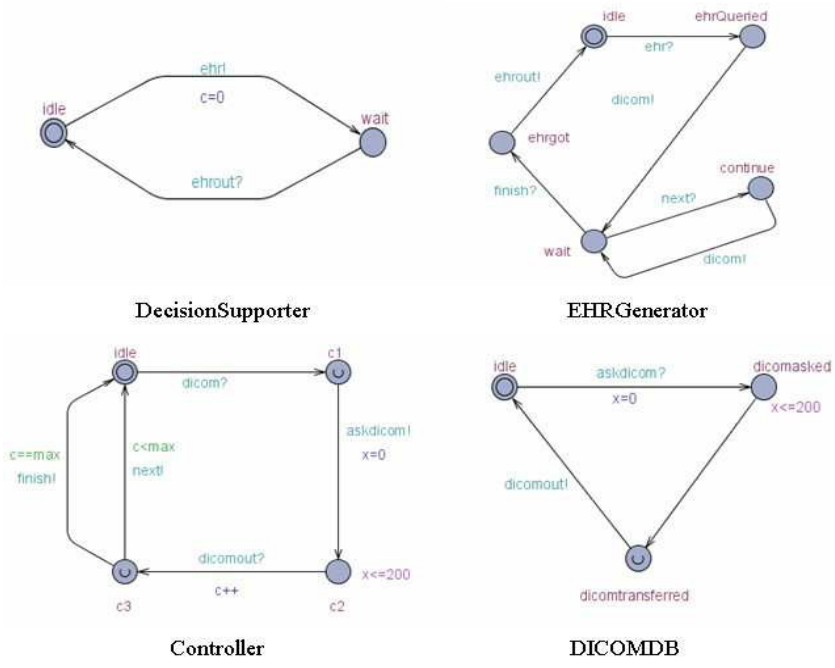

Fig. 11. Automata Models

4.2 Timed Automata Model of DICOM Transfer

According to the description and sequence diagram with time constraints for DICOM transfer problem, we firstly declare all the variables and channels used in this subsystem. There are two global variables *max* and *c*, denoting the maximum number of DICOM for an EHR item and the count of DICOM that has been transferred successfully. The controllers of the whole subsystem interact with each other through channels, including *ehr*, *next*, *dicom*, *dicomout*,

finish, *askdicom*, *ehrout*, and we define these channels as urgent channels which disallow delay in the state transition. Thus our model system can be defined as follows:

int *c;*
const int *max = 5;*
system *DecisionSupporter, EHRGenerator, Controller, DICOMDB;*
urgent chan *ehr, next, dicom, dicomout, finish, askdicom, ehrout;*
clock *x;*

Then we obtain corresponding time automata models in Fig 11.

4.3 Analysis and Verification of DICOM Transfer

After modeling, the interactions between objects can be observed in the simulator of UPPAAL. We give the random message sequences of communication and control between these objects through channels in Fig 12. Based on the message sequences, we can make a preliminary judgement whether the model conforms to system.

To make further verification of the properties of our model, we may enter query language in the verifier of UPPAAL. For example, *A[] not deadlock* is to verify the system is deadlock free, and *A[] DICOMDB.dicomtransferred imply x <= 200* is to verify that the time to transfer each DICOM to the *EHRGenerator* should not exceed 200 ms, and *DecisionSupporter.wait → EHRGenerator.ehrgot* is to verify that once the *DecisionSupporter* asks for a EHR item, the system is sure to send the EHR item back. Other properties can be verified with similar expressions.

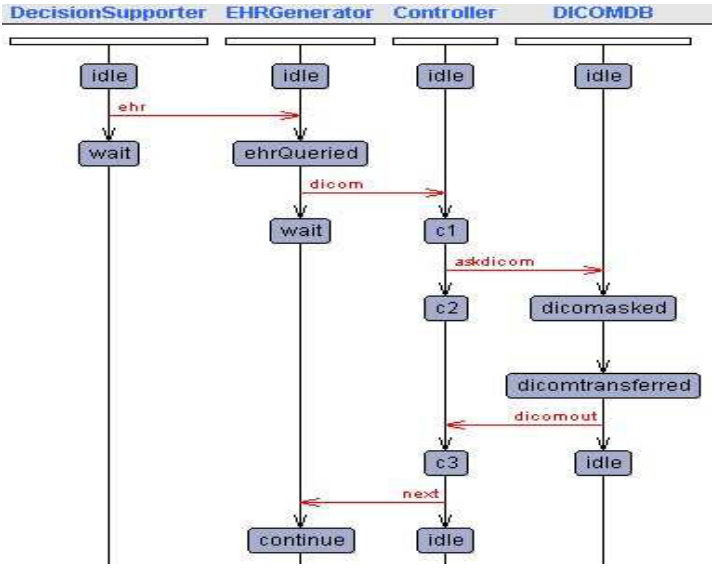

Fig. 12. a Random Message Sequence Chart

With the help of UPPAAL, the safety and correctness of the system can be ensured by the verification of the properties such as safety and liveness.

5 Conclusions and Discussion

Telemedicine is a new medical diagnosis and medical education mode to build the connection between different medical institutions. But due to the differences in computer systems and data recording rules, seamless sharing of data and developing a trustable health care system become a grand challenge in the 21st century. In this paper, we construct TMS that can be integrated to telemedicine systems based on the technology of rCOS. On one hand, the system uses EHR system to store the patient health record in a standard way and also can be linked to any other EHR systems, which realizes the data sharing between different hospitals in different countries. On the other hand, TMS analyzes the EHR information based on its reliable scientific EBM and responds with reminders and guidelines, thus enables the end-users to make correct and in-time decisions.

In practical software engineering, complexity of software system is handled by separation of concerns and incremental development [8,16], which forms *a component-based model-driven development*(CB-MDD) process. In CB-MDD process, as the system model is split into several parts and thus different views of the system are described in a multiview language[2,12], it is important to ensure consistency among these views and correctness of the system properties. [16] proposed an approach to support separation of concerns and consistent modeling of requirements. [1,4] used design patterns, object-oriented and component-based design to support separation of design and validation of different concerns. To assure the correctness of system properties, formal methods is needed for precise verification of the models produced in such a process. One of the most popular developed semantic theories of verification is *model checking*[3]. It uses automata, state transition systems, and temporal logics to verify dynamic control behaviors of system by the support of model checking tools[5,9]. However, most of the framework in verification ignored the feasibility of formal verification on models generated in CB-MDD process, thus it become not easy for verification method using model checking tools to integrate into this process.

To integrate verification method into CB-MDD process, Our approach is constructing system models by rCOS[7], a rather rich and mature formalism that models both static and dynamic features for component based systems. rCOS separates the protocol of the provided interface methods from that of the required interface methods for components and a UML sequence diagram is used to describe the interaction protocol. By the stepwise functionality refinement in rCOS, the sequence diagram can be refined to a component sequence diagram, which becomes the basis model for verification by UPPAAL. Thus the CB-MDD approach and verification method are combined together by the use of rCOS. Our work also suggests a way to corporate design and verification in system develop process.

Acknowledgments

We are grateful to the anonymous reviewers for their detailed comments and suggestions that improved the paper. This work is partially supported by National High

Tech Research 863 Program of China under Grant No.2006AA01Z165; the National Natural Science Foundation of China under Grant No. 90718014, No.90818013, No.60673114; 973 Program of China under Grant No.2009CB320702 and STCSM No.08510700300.

References

[1] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, Prentice-Hall International, 2001.

[2] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison- Wesley, 1998.

[3] Edmund M. Clarke, et al., Model Checking, The MIT Press, 1999.

[4] E. Gamma, et al., Design Patterns, Addison-Wesley, 1995.

[5] G. Holzmann, The SPIN Model Checker: Primer and Reference Mannual, Addision-Wesley Professional, 2003.

[6] Isabel Healthcare: isabel products, http://www.isabelhealthcare.com/.

[7] J. He, X. Li, Z. Liu, rCOS: A refinement calculus for object systems, *Theoretical Computer Science*, vol.365, no.1-2, pp.109-142, 2006.

[8] K. Chandy, J. Misra, Parallel Program Design: a Foundation, Addison-Wesley, 1988.

[9] K. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, STTT 1. 1-2 pp.134-152, 1997.

[10] LCS: Using decision support to help explain clinical manifestations of disease, http://lcs.mgh.harvard.edu/projects/dxplain.html.

[11] NATIONAL ACADEMY OF ENGINEERING: GRAND CHALLENGE FOR ENGINEERING: http://www.engineeringchallenges.org/.

[12] Object Management Group, Unified Modeling Language: Superstructure, http://www.omg.org/cgi-bin/doc?formal/05-07-04.

[13] Victoria Torres, Pau Giner, and Vicente Pelechano, Development of a Ubiquitous Decision Support System for Clinical Guidelines using MDA, In *Proceedings of the CAiSE'07 Forum*, Trondheim, Norway, June 2007.

[14] Varonen H, Jousimaa J, Helin-Salmivaara A, Kunnamo I, Electronic primary care guidelines with links to Cochrane reviews-EBM Guidelines, Fam Pract. 22, pp.465-9, 2005.

[15] X. Chen, J. He, Z. Liu, N. Zhan, A model of component-based programming, *Lecture Notes in Computer Science*, vol.4767, 2007.

[16] X. Chen, Z. Liu, V. Mencl, Separation of concerns and consistent integration in require- ments modelling, *Lecture Notes in Computer Science*, vol. 4362, 2007.

[17] Z. Chen, Z. Liu, A. Ravn, V. Stolz, N. Zhan, Refinement and Verification in Component-Based Model Driven Design, *Science of Computer Programming*, vol.74, no.4, pp.168-126, 2009.