MEDINFO 2010
C. Safran et al. (Eds.)
IOS Press, 2010
© 2010 IMIA and SAHIA. All rights reserved.
doi:10.3233/978-1-60750-588-4-299

Goal-based design pattern for delegation of work in health care teams

Adela Grando[a], Mor Peleg[b], David Glasspool[a]

[a] School of Informatics, Edinburgh University, Edinburgh, United Kingdom
[b] Department of Management Information Systems, University of Haifa, Israel, 31905

Abstract

We show how a domain and language independent design pattern, defined as networks of tasks and goals, can be used to formally specify the transfer of responsibility and accountability when tasks are delegated in healthcare teams. The pattern is general enough to be applied unchanged across a broad range of different healthcare situations.

Keywords:

Clinical guideline, Goal, Design pattern, Medical error, Exceptions

Introduction

Clinical guidelines can contribute to the definition of better, safer, and more efficient evidence-based clinical care. Computer-interpretable guidelines (CIG) [1] can potentially increase the effectiveness of clinical guidelines by delivering patient-specific decision-support at the point of care. In general CIGs are defined in a particular language and lessons learned while developing them are difficult to share with groups working with other languages or different medical conditions. A possible answer is to specify generic solutions or *design patterns* [2] to recurrent common problems recognized in health informatics using a formal vendor-independent framework that allows sharing, reuse and study of patterns. The idea of creating a catalog of generic patterns that could be accessed and instantiated into particular problems using different languages has been previously suggested [3-5].

Healthcare processes, such as those modeled in clinical guidelines, are often carried out by teams. Incomplete or ambiguous specification of responsibilities and accountabilities in collaborative team work and the possible lack of accountability of medical staff working in shifts are important problems in healthcare [6, 7]. According to [8] "When delegating work to others, registered practitioners have a legal responsibility to have determined the knowledge and skill level required to perform the delegated task. The registered practitioner is accountable for delegating the task and the support worker is accountable for accepting the delegated tasks, as well as being responsible for his/her actions in carrying it out. This is true if the support worker has the skills, knowledge and judgement to perform the delegation".

In team work, delegation and assignment of tasks/goals is done based on the competences of the members of the team. During delegation, the responsibility for enacting a service and handling exceptions is passed from the requester (client) to a performer (provider); when the provider cannot cope with the exceptions he has to inform the client to transfer the responsibility. The accountability for the service outcome and exceptions arising during the service enactment is retained by the client [9].

We aim to tackle incomplete and ambiguous specification of responsibility and accountability in health care teams by formally specifying the transfer of responsibility and accountability in normal and abnormal situations during delegation of tasks/goals.

Methods

We formalize cooperative work in teams by extending a vendor-independent framework that we previously developed for specifying clinical design patterns [5]. We use the extension to define a generic pattern for delegation of tasks/goals that specifies levels of responsibility and accountability in normal and abnormal situations.

Framework for specifying design patterns for normal and exceptional behavior

In our framework [5] design patterns are specified as networks of tasks and goals (collectively termed "keystones") connected by scheduling constraints based on Petri Nets: all the incoming keystones need to be completed to enact the out coming keystone (AND join), the execution of only one of the incoming keystones is required to enact the out coming keystone (XOR join), all the out coming keystones are enacted after the antecedent keystone is completed (AND split) and only one of the out coming keystones is enacted after the antecedent keystone is completed (XOR split). As in the PRO*forma* model [1], tasks can be *decisions, enquiries, actions*, or *plans* (careflows comprising activities and goals). Goals represent temporal patterns of state variables which should be *achieved* or *maintained*. When a goal is active, a decision-support system proposes from a repository one or more candidate plans for satisfying the goal. Once the plan chosen for achieving a goal has been completed the goal is still active and its *successCondition* is checked to see if it has been achieved.

The framework allows abstraction of recurrent domain-specific scenarios as patterns, as well as abnormal scenarios originating from domain-specific or generic medical errors. Deviations from the expected process are abstracted using hierarchical definitions in a catalog of state-based exceptions, such that an exception is triggered when the corresponding state occurs, activating a goal-based pattern which abstracts commonly used strategies for repairing or recovering from the detected error. These strategies include invoking exception-handling flows and suspending or discarding affected keystones. The suspended or discarded keystones can revert to their previous state only after the exception-handling flow is completed. Exceptions are classified as hazards or obstacles. A *hazard* corresponds to a state that can *potentially* produce harm to the patient and an *obstacle* corresponds to a state where nominal execution of the guideline is not possible, either because the task cannot be completed or if its completion is no longer beneficiary to the patient.

Extending the design-pattern framework by specifying roles and actors

We extend the framework of [5] by proposing four new types:

type Role= <name, competences, restrictions, constraints>

Name uniquely identifies the *role; competences and restrictions* are sets of keystones that the *actors* performing the *role* can and cannot perform, respectively; *Constraints* are predicates that an *actor* must satisfy to play a *role*. For example to play the *role* of general practitioner (GP) the role player must be a registered practitioner. Role competence of health professionals is regulated by statues and professional bodies.

type Actor= <name, roles, competences, restrictions, attributes >

The *name* uniquely identifies the *actor; Roles* are set of *role* names that the *actor* is playing; *Competences* and *restrictions* specify those different from the ones inherited from the roles played by the actor. The sets of competences and restrictions should be based on the *actor*'s attributes. For instance in general nurses are not allowed to provide service X but nurse Ana can do it because she has taken a recognised course. Finally the *Attributes* are set of predicates that can be used to check if the *actor* satisfies the *role*'s constraints (e.g., has_degree_GP) or to select the actor for service delegation (e.g., based on the attributes experience, other_medical_specialities).

The competence, accountability, and delegation of services for some health registered professionals are regulated by statutes and regulatory bodies. In the UK regulatory bodies include the Nursing and Midwifery Council for nurses, midwives and health visitors, the Health Professions Council for physiotherapists, dieticians, speech and language therapists, and so on. Roles not regulated by statutes are accountable for their actions in three ways: civil law (duty of care), criminal law, and employment law. Therefore there are good sources of information that can be used to specify, in the way proposed above, the competences of the roles played by health care professionals. Once the *roles* and *actor* specifications have been completed the following functions can be used to determine (1) conflicts between two sets of competences and restrictions, (2) an actor's competence to perform a service (keystone), and (3) the set of actors who can provide a service for a client based on their competences and the client's constraints.

1. Boolean **function *areConflicting***(keystoneSet Compentences, keystoneSet Restrictions)=

```
{ If intersection (Competences, Restrictions)!=null
        then return true else return false;  }
```

2. Boolean **function *isCompetent*** (Actor actor, Keystone service)=

```
{ roleCompetences, roleRestrictions==emptySet;
  roles=actor.GetRoles() ;
    While roles!=null
    { roles.GetFirst()=role;
      roleCompetences= union(role.GetCompetences(),
                             roleCompetences);
      roleRestrictions= union(role.GetRestrictions(),
                             roleRestrictions);
      roles.remove(role);
    }
  allCompetences= union(roleCompetences,
                        ator.getCompetences());
  allRestrictions= union(roleRestrictions,
                        actor.getRestrictions());
  If not areConflicting(allCompetences, allRestrictions) &&
     allCompetences.contains(service) &&
     not allRestrictions.contains(service)
            then return true
            else return false;
  }
```

An actor is competent to perform a service if and only if: there is no conflict between the restrictions and competences defined for actor and role, the actor is competent to perform the service (actor's and roles' competences satisfy the requirements for the service), and the service is not included in the actor's and role's sets of restrictions.

3. ActorSet **function *ObtainCompetentProviders***(Keystone service, Proposition constrains, ActorSet staff)=

```
{ providers=emptySet;
  While staff!=emptySet
  { staff.Retrieve()=staffmember;
    If isCompetent(staffmember, assignment) &&
       canSatisfy(staffmember, assignment, constraints)
          then
             providers.add(staffmember) ;
             staff.remove(staffmember);
    }
  return providers;
}
```

The function ***canSatisfy*** takes as arguments an actor, a service, and a constraint and it returns true if the actor can perform the service satisfying the constraints. Examples of constraints include time restrictions, place where the service should be provided, etc.

Each delegation starts with a service request:

type request = <*client, provider, service, service type, satisfyCompletion, constraints*>

Client identifies the agent that requires the service; *provider* corresponds to the agent that agrees to provide the service; *service* is the task that is assigned/delegated to the provider by the client; *service type* indicates the type of service requested and can take the values *assg, deleg, sdeleg* indicating assignment, delegation without supervision and delegation with supervision; *satisfyCompletion* is a function given by the client of an assignment to the provider to check if the service satisfies the client's criteria of service completion; *constraints* can be defined by the client to restrict the way the service should be provided. For instance the time constraint that the service should be provided in less than 3 hours.

If a provider accepts a service request a contract is defined:

type contract= < *service request, startTime, finishTime*>

Service request is the identifier of the service request that originated the contract; *startTime* is the date the contract starts; *finishTime* corresponds to the date the contracts finishes. Always *finishTime>startTime.*

Figure 1 shows the relationship between the new introduced types and the already existing types from the framework [5] used for the specification of the delegation pattern.

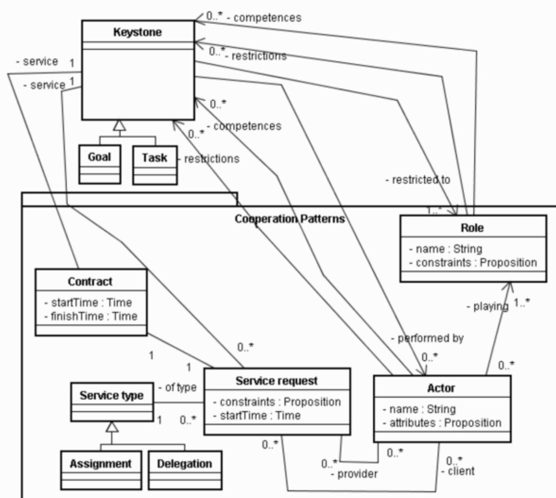

Figure 1- Class Diagram showing the connection between the terms used for the specification of the Delegation Pattern

Definition of service delegation

An actor called *client* delegates the enactment of a task or the achievement of a goal to a competent actor called *provider* such that:

Property 1. The provider is competent and responsible for providing the service.

Property 2. The client retains accountability for the service's outcome and any exceptions arising from the service enactment.

Property 3. The provider is responsible for handling any exceptions arising during the service enactment. When the provider cannot handle an exception the provider must transfer responsibility back to the client.

Property 4. The client is responsible for managing any exceptions that the provider cannot handle (whether detected by provider or client).

Design pattern for delegation of services

We define the delegation pattern based on formal approaches for delegation of tasks (services) between collaborative agents [10] from agent-oriented software engineering.

The delegation pattern is divided between the client's delegation workflow (Figure 2.1) and the provider's delegation workflow (Figure 2.2).

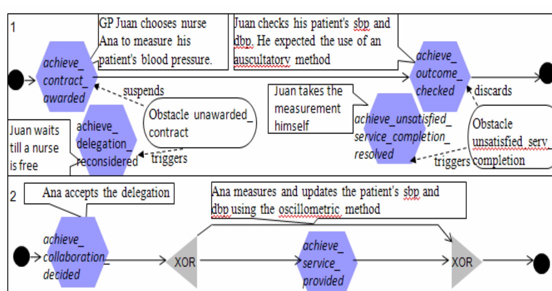

Figure 2- Delegation Pattern: 1) Client_delegation 2) Provider_delegation. In Figure 2.2 the first scheduling constraint corresponds to an XOR split, and the second constraint is an XOR join.

Table 1 contains the formal specification of the *Client_ delegation* workflow. As specified by the *precondition* of the client's workflow, a *service* can be delegated if the client has the competence to do so (according to function *isCompetent* the client can pursue the goal *achieve_ delegated*) and the service is not already assigned to another provider (there is no open *contract* and the service has not been requested according to *serviceRequest*). For instance the role general practitioner (GP) is competent to delegate the measurement of the patient's blood pressure to members of the hospital staff competent for that task, only if the same request is not being processed.

As shown in Figure 2.1 first the client tries to achieve the goal *achieve_contract_awarded*. For instance, the GP Juan can check the set of competent staff and delegate the task of measuring the patient's blood pressure to nurse Ana because she is available at the time he is requesting. In the exceptional case where no provider is willing to provide the requested service or a timeout has elapsed and no contract has been awarded, the discarding obstacle *unawarded_contract* is triggered, which discards the goal *achieve_contract_awarded* and triggers the

goal *achieve_delegation_reconsidered*. Thus Juan may decide to relax his delegation condition delaying the task to the first time when there is a nurse available.

In the best case a contract is awarded between client and provider (goal *achieve_contract_awarded*) and the client waits for service completion. In our example Juan can check the service completion by accessing the patient's record that contains the latest measures of the patient's systolic and diastolic blood pressure (sbp, dbp).

It may happen that after the provider has completed the service the client's criterion of service completion is not satisfied; in this case the suspending obstacle *unsatisfied_service_ completion* is triggered. For instance, Juan specified that he wanted to have his patient's blood pressure measured using the auscultatory method, but according to the patient's records the measurement has been done by an oscillometric method. The obstacle *unsatisfied_service_completion* suspends the goal *achieve_ outcome_checked* and triggers the goal *achieve_unsatisfied_ service_completion_resolved*. In our example Juan decides to make an appointment with the patient to take the measurement himself.

Client Juan is responsible and accountable for both exceptions *unawarded_contract* and *unsatisfied_service_completion* because they happened before and after the service enactment, respectively. If any exception had happened during the service enactment nurse Ana should be responsible for dealing with it.

The workflow *Client_delegation* is completed when the goal *achieve_outcome_checked* is achieved and, as described in Table 1, the contract between client and provider has been closed, and the client's completion criteria is satisfied.

Table 1- Client_delegation

Attribute	Client_delegation
Parameters	service, contracts, staff, preferences, isComplete, serviceRequests
Precondition	isCompetent(actor, achieve_delegated((service, contracts, staff, preferences))) & not contracts.contains(service, anytype, actor, anyProvider, start, null) & not serviceRequests.ObtainAll(). contains(this.GetActor(),anyProvider, service, anyType)
Success Condition	ObtainProviders(service,preferences,staff). contains(provider) & contracts.contains (service,deleg, this.GetActor(),providers,start, finish) & isComplete (service.GetSuccessCond())

We now turn to the provider's workflow in the delegation pattern (Figure 2.2). For the sake of brevity we do not provide the formal specification for the provider's workflow. The provider's workflow is activated when an actor receives a request for service delegation from a client and there is no contract between the client and any provider for this service. The provider decides whether he wants to collaborate, in which case he satisfies the goal *achieve_collaboration_decided*. If he does not want to collaborate the provider's workflow ends without activating the goal *achieve_service_provided*. For instance Ana receives a request from Juan to make an appointment to measure a patient's blood pressure at a time she is available. Ana is competent to perform the task so she accepts the appointment. If as in this example the provider accepts the *achieve_contract_awarded* is achieved, the provider's first goal is achieved and the provider's second goal *achieve_ service_ provided* is activated. The provider's workflow finishes when according to his completion criteria the service has been completed. The provider's criteria for service completion are not necessarily identical to the client's criteria for service completion. For example for Juan the blood pressure should be taken using an auscultatory method, while for Ana the task is achieved when any accurate measuring method is used. Because of possible differences between the client's and provider's completion criteria after the workflow *Provider_ delegation* has been completed the contract between client and provider is still open until the client checks that the service's outcome is the desired one (goal *achieve_outcome_checked*).

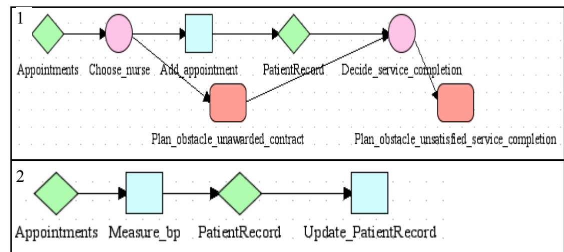

Figure 3 - Implementation of the delegation pattern in the Tallis toolset.

Results

Properties satisfied by the pattern

Property 1: In the case of the client the pattern is defined in terms of the goal *achieve_contract_awarded*. Therefore providing the service has not been assigned to anyone else, the provider is *competent* to provide the requested service and accepts the delegation (as specified by goal *achieve_ collaboration_decided*) then a contract is opened between them, which makes the provider *responsible* for providing the service.

Property 2: As specified by the delegation pattern, the goal *achieve_outcome_checked* is part of the client's workflow, therefore he checks that his completion criteria is satisfied after the provider has finished the delegated service. Only if the client's completion criteria are satisfied is the delegation contract between the client and provider closed. Because a delegation contract is signed between the client and the provider when the goal *achieve_collaboration_decided* is achieved, the client becomes *accountable* for any exception arising from the service enactment.

Property 3 can be proved only if the following property is satisfied by the catalogue of exceptions provided by the exception manager:

Property to be satisfied by the catalogue of exceptions: *Each exception from the provided catalogue is specified such that the actor responsible for meeting the goal that was triggered in order to handle the exception is the actor who enacted the keystones or goals that triggered the exception.*

For any catalogue of exceptions provided for the patterns this property must be checked.

Property 3: If the repository of exceptions satisfies the property explained above then the provider is responsible for enacting exceptions arising from the service enactment. But in case the provider cannot cope with the exception he can inform the client and transfer to the client the responsibility of dealing with the exception. A hazard can be triggered to inform the client about the exception and the recovery strategies that he has unsuccessfully tried.

Property 4: When the provider achieves the goal *achieve_exception_informed* the provider has been informed about the unresolved exception which arose during service enactment and responsibility for enacting a plan to recover from the exception has been transferred the provider. Once the provider has been informed about the unresolved exception he can activate the goal *achieve_exception_recovery_decided*.

Pattern enactment

Design patterns have proved to be very powerful generic and abstract mechanisms for software analysis, design, and comparison, provided they can be mapped to concrete executable languages. In Figure 3 we show an implementation of the delegation pattern in the Tallis[11] toolset used for enacting PRO*forma* guidelines. Each component from the delegation pattern is mapped into one or more Tallis components. Figure 3.1 corresponds to the *Client_delegation_pattern*. To pursue the goal *achieve_contract_awarded* the GP starts querying the existing appointments (query *Appointments*) and chooses an available nurse (decision *choose_nurse)* to delegate the task of measuring his patient's blood pressure. When a nurse is chosen a new appointment is created (action *add_apointment*). Both GP and nurse roles and the actors playing those roles are specified as described by the types *Role* and *Actor* that we introduced to extend the design-pattern framework. To satisfy the goal *achieve_outcome_ checked* the GP activates the action *check_patient_bp* after the appointment date. Possible exceptions are: the case when no nurse is free, which activates the plan *Plan_obstacle_unawarded_contract;* or the case when the service has not been completed according to GP's requests, which activates the plan *Plan_obstacle_unsatisfied_ service_ completion*. Figure 3.2 corresponds to the plan *Provider_ delegation_ pattern.* In this hospital the nurses cannot refuse to take appointments, therefore the goal *achieve_ collaboration_ decided* is always satisfied after the GP chooses a nurse. The provider's plan starts when the nurse pursues the goal *achieve_service_provided* by taking the blood pressure measurement the date chosen for the appointment (query *Appointments* followed by action *measure_bp*). The delegated service is completed when the patient's record is updated with the measurement (query *PatientRecord* followed by action *update_ PatientRecord*).

Discussion

The delegation and assignment patterns have been enacted by mapping them into the Tallis tool used for running PRO*forma* guidelines. In addition a simplification of the patterns, which does not include exception detection and recovery, has been implemented and enacted in a COGENT prototype. What remains to be done is to fully explore the practical benefits of the use of these patterns, by mapping them into a real clinical-based application.

Acknowledgements

This work was supported by EPSRC grant EP/F057326/1 and by a programme grant from Cancer Research UK to D. Glasspool.

References

[1] Peleg M, Tu SW, Bury J, Ciccarese P, Fox J, Greenes RA, et al. Comparing Computer-Interpretable Guideline Models: A Case-Study Approach. JAMIA 2003;10(1):52-68.

[2] Gamma E, Helm R, Johnson R, Vlissides JM. Design patterns: Elements of reusable object-oriented software. Reading, MA: Addison-Wesley Publishing Company; 1995.

[3] Tu SW, Campbell JR, Glasgow J, et al. The SAGE Guideline Model: achievements and overview. JAMIA 2007;14(5):589-98.

[4] Peleg M, Tu SW. Design Patterns for Clinical Guidelines. AIIM 47-1:1-24, 2009.

[5] Grando A, Peleg M, Glasspool D. A goal-oriented framework for specifying clinical guidelines and handling medical errors. J Biomedical Informatics. Accepted Nov 2009.

[6] Mackey H. Assistant Practitioners: issues of accountability, delegation and competence. Intl J therapy and rehabilitation 2005;12(8):331-8.

[7] Richardson G, Maynard A, Cullum N, Kinding D. Skill mix changes: substitution or service development? Health policy 1998;45(2):119-32.

[8] CSP, RCSLT, BDA, RCN. Supervision, accountability and delegation of activities to support workers, a guide for registered practitioners and support workers. 2006.

[9] Barter M, Furmidge ML. Unlicensed assistive personnel. Issues relating to delegation and supervision. JONA 1994;24(4):36-40.

[10] Sycara K, Sukthankar G. Literature Review of Teamwork Models: Robotics Institute, Carnegie Mellon University; 2006. Tech Report CMU-RI-TR-06-50.

[11] Tallis toolset, available at: http://www.cossac.org/tallis