



Patterns for collaborative work in health care teams

Maria Adela Grando^{a,*}, Mor Peleg^b, Marc Cuggia^c, David Glasspool^a

^a School of Informatics, University of Edinburgh, 10 Crichton Street, Edinburgh EH8 9AB, UK

^b Department of Management Information Systems, University of Haifa, Haifa, Rabin Bldg. 31905, Israel

^c Unité Inserm U936, IFR 140, Faculté de Médecine, University of Rennes 1, 2 Avenue du Professeur Léon Bernard, 35043 Rennes Cedex 9, France

ARTICLE INFO

Article history:

Received 24 February 2010

Received in revised form 5 July 2011

Accepted 9 August 2011

Keywords:

Design patterns

Medical guidelines

Exceptions

Team work

Tallis

PROforma

ABSTRACT

Objective: The problem of designing and managing teams of workers that can collaborate working together towards common goals is a challenging one. Incomplete or ambiguous specification of responsibilities and accountabilities, lack of continuity in teams working in shifts, inefficient organization of teams due to lack of information about workers' competences and lack of clarity to determine if the work is delegated or assigned are examples of important problems related to collaborative work in healthcare teams. Here we address these problems by specifying goal-based patterns for abstracting the delegation and assignment of services. The proposed patterns should provide generic and reusable solutions and be flexible enough to be customizable at run time to the particular context of execution. Most importantly the patterns should support a mechanism for detecting abnormal events (exceptions) and for transferring responsibility and accountability for recovering from exceptions to the appropriate actor.

Method: To provide a *generic* solution to the problematic issues arising from collaborative work in teams of health workers we start from definitions of standard terms relevant for team work: competence, responsibility, and accountability. We make explicit the properties satisfied by service assignment and delegation in terms of competences, responsibilities, and accountability in normal scenarios and abnormal situations that require the enactment of recovery strategies. Based on these definitions we specify (1) a basic terminology, (2) design patterns for service assignment and delegation (with and without supervision), and (3) an exception manager for detecting and recovering from exceptions. We use a formal framework to specify design patterns and exceptions.

Results: We have proved using Owicki–Gries Theory that the proposed patterns satisfy the properties that characterize service assignment and delegation in terms of competence, responsibility and accountability in normal and abnormal (exceptional) scenarios. We show that although *abstract*, the proposed patterns can be instantiated in an executable COGENT prototype, and can be mapped into the Tallis tool that enacts PROforma language specifications of medical guidelines.

Conclusions: The proposed patterns are *generic* and *abstract* enough to capture the normal and abnormal scenarios of assignment and delegation of tasks in collaborative work in health care teams.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

Medical errors constitute an important source of adverse events and may even result in patients' death, according to the Committee on Quality of Health Care in America [1]. To tackle this problem, clinical guidelines have been proposed to assist practitioner and patient decision making about appropriate health care for specific clinical circumstances [2]. Clinical guidelines can contribute to the definition of better, safer, and more efficient evidence-based clinical care, where errors, inconsistencies, incompleteness, or inefficiencies can be detected and can provide a basis

for computer decision support. Studies have shown that guideline implementations can best affect clinician behavior if they can deliver patient-specific advice during the clinical encounter [3,4]. Several groups have developed computer-interpretable guideline (CIG) specification languages for this purpose, see Peleg et al. work for a review [5]. In general CIGs are defined in a particular language and the lessons learned from modelling clinical guidelines are difficult to share with developers who are using other languages. One way to address this problem is to specify generic solutions or *design patterns* for common situations using a formal framework that allows sharing, reuse and analysis of proposed patterns. The idea of creating a catalogue of generic patterns or hierarchical skeletal plans that could be accessed and instantiated into particular problems using different languages is not new in health informatics [6–10].

* Corresponding author. Tel.: +44 0131 651 5657; fax: +44 131 650 6899.

E-mail address: adelagrando@yahoo.com.ar (M.A. Grando).

In this paper we focus on a particular set of common scenarios encountered when teams of workers collaborate together towards common goals. Incomplete or ambiguous specification of responsibilities and accountability in collaborative team work and the lack of continuity in teams of medical staff working in shifts are important problems in healthcare. An evaluation of assistant practitioners in occupational therapy in the UK [11] exposed ambiguity and lack of clarity regarding whether work is delegated or assigned, and difficulty in determining levels of competence of staff. Many professionally trained health care staff including nurses, doctors, and the allied health professionals spend a high proportion of their time performing tasks that do not require their expertise, according to the survey carried on by Richardson et al. [12]. Quoting Mackey [11] “These issues need to be addressed in order to optimise the relationships between staff, clarify the roles of team members and ensure service users receive the most appropriate care from the most appropriate practitioner”.

Here we propose patterns for specifying assignment and delegation of tasks and goals during collaborative work, considering normal and abnormal scenarios where exceptions can occur. To specify assignment and delegation patterns we use the framework developed by Grando et al. [6] for specifying generic goal-based CIGs provided with catalogues of state-based exceptions for detecting states of error (obstacles) or situations possibly harmful for the patient (hazards). The approach allows abstraction of patterns representing not only the common scenarios themselves, but also abnormal situations occurring as a result of domain-specific or generic errors and unanticipated events. Abnormal situations are abstracted using hierarchical catalogues of state-based exceptions, such that an exception is triggered when the corresponding state occurs, activating a goal-based pattern encapsulating commonly used strategies for repairing or recovering from this type of error.

This work is an extension of the initial research presented at MedInfo 2010 [13] and can be seen as a continuation of proposed patterns for addressing the problem of collaboration in teams of collaborative agents like [14,15], or distributed peers in service-oriented systems [16], or proposed agent-based approaches for managing team work in health care like [17].

The paper is organized as follows. In Section 2 we explain our methodology. In Section 2.1 we define some standard terminology used for cooperative work in teams: competence, responsibility, accountability, assignment and delegation (with and without supervision) of tasks. In the case of service assignment and delegation we have made explicit the responsibilities and accountabilities that apply if things go wrong. In Section 2.2 we introduce the formalism introduced by Grando et al. [6] chosen for the specification of patterns in Section 3.

In Section 3 we present our results as follows. In Section 3.1 we introduce the formal terms resulting from formalizing the concepts introduced in Section 2.1. In Sections 3.2 and 3.4 we present the patterns that we propose for the assignment and delegation (with and without supervision) of tasks and goals, whose control conditions are based on the terminology presented in Section 2.1. Both patterns are provided with a basic catalogue of exceptions for specifying accountability in abnormal situations. While we have formally proved that the patterns satisfy the properties related to competence, responsibility and accountability in normal and abnormal scenarios, as specified in Section 2.1, we only give in Sections 3.3 and 3.5 few details of the formal proofs and instead we informally explain them.

In Section 4 we discuss the applicability of the patterns, first in Section 4.1 using a concrete healthcare scenario to instantiate and enact the patterns in the prototyped framework used for the patterns' specification, and then in Section 4.2 we provide an example of pattern mapping into the PROforma clinical guideline modelling

language using the Tallis enactment tool [18]. Finally in Section 5 we present our conclusions.

2. Methods

2.1. Specification of the adopted terminology for team work

Our starting point is a set of specifications for the processes of assignment and delegation of tasks. We base our specifications on three related concepts: the *competence* of an individual to carry out a task, the *responsibility* of an individual for a task, and the *accountability* of an individual for successful completion of a task. We adopt the following definitions from the medical informatics literature:

- **Competence:** According to the Royal College of Nursing [19], competence is the ability of an individual to effectively apply his knowledge, experiences, training, and further skills in the execution of a piece of work. Here we extend the notion of competence to the ability of an individual to assign/delegate the execution of a piece of work to other competent individual, in order to allow chains of collaborations.
- **Responsibility and accountability:** According to Webster's dictionary, being responsible implies holding a specific office, duty, or trust; while accountability suggests imminence of retribution for unfilled trust or violated obligations. As expressed by Shardlow [20] accountability relates to the position held within an organization while responsibility is a personal concept and derives from being a member of society, and therefore responsible for one's actions. “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, and that the delegation of task falls within the guidelines and protocols of the workplace, and the level of supervision and feedback is appropriate” [19].

Based on these definitions, the bibliographic survey from Mackey [11] and the Royal College of Nursing [19], our experience of team work in the medical field, and formal studies from the area of agent-oriented and service-oriented systems [14–16] we introduce the following definitions:

2.1.1. Assignment

Precondition 1 The actor called *client* is competent to assign the service to a *provider* and there is no assignment/delegation service request or open contract between the client and any provider for the same service.

Property 1. *The provider is responsible for providing the service and the client can check that the provider is competent to provide the service himself, or competent to assigning/delegating the service.*

Property 2. *The provider is accountable for the service's outcome.*

Property 3. *The provider is responsible and accountable for any exceptions arising during the service enactment.*

2.1.2. Delegation

Precondition 2 The actor called *client* is competent to delegate the service to a *provider* and there is no assignment/delegation service request or open contract between the client and any provider for the same service.

Property 1. *The provider is responsible for providing the service and the client can check that the provider is competent to provide the service himself, or competent to assigning/delegating the service.*

Property 4. *The client retains accountability for the service's outcome and any exceptions arising from the service enactment.*

Property 5. *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 6. *The client is responsible for managing any exceptions that the provider cannot handle (whether detected by provider or client).*

Delegation with supervision should additionally satisfy the following property:

Property 7. *The client is responsible and accountable for supervising the provider's work.*

In a delegation with supervision, the client is responsible for putting in place a supervision system that protects the patient and maintains the highest possible standard of care. Supervision may be periodic or continuous, and may incorporate elements of direction, guidance, observation, joint work, discussion, exchange of ideas, and coordination of activities.

Unless there is some specific policy that prohibits or limits the subdelegation or subassignment of tasks, it should be possible to assign/delegate all or some of the following competences and responsibilities: service enactment, exception detection, exception handling or service enactment supervision (for delegations with supervision).

As an example of assignment a nurse can assign administration of a medication either to another nurse or to unlicensed nursing personnel. Once the nurse has confirmed that the assignment is within the knowledge base and skill level of the provider, the nurse is no longer responsible for oversight of the activity.

For example a physician may delegate (without supervision) a certain pharmacy (preferred by the patient) to dispense drugs to his patient. The pharmacist must check the prescription and any available source of information, like an electronic medical record, that could provide more details of the patient's medical condition. It may be that the pharmacy can provide an equivalent treatment (same molecule and galenic but a different brand). Under this exception the pharmacist is allowed to provide the new treatment without asking the physician. It can also happen that the pharmacist does not have the exact treatment prescribed by the physician but a similar one (for instance another molecule which has exactly the same effect). In this case the pharmacy needs to contact the physician to ask for his consent. Another possible exception corresponds to the scenario in which the pharmacist finds that the physician has prescribed a non recommended treatment, for instance an inflammatory to a patient with kidney disease. In this case the pharmacist must contact the physician to change the prescription.

As an example of delegation with supervision, a home health nurse, after assessing a patient's home environment, instructs a home health aide who is living at the patient's house and taking care of her. The nurse then supervises the aide during the following year meeting with the aide on a biweekly basis for evaluation of the patient. As another example, a nurse who has recently graduated and has started working in a hospital will need to have a more experienced nurse assigned to carry on a continuous supervision until the nurse has demonstrated a certain level of expertise. Here the recently graduated nurse is the provider and the experienced nurse is the client.

Chains of collaborative interactions are possible, for instance a family physician examines a patient and assesses that the patient has a medical condition whose diagnosis and treatment are outside his scope of knowledge, therefore he refers the patient to a specialist. The chosen specialist can delegate to a nurse the task of

periodically giving a patient injections, or assessing the patient's condition. The physician retains accountability for the service and for any abnormal situation that arises while the service is being provided. If the nurse has to deal with an exception during enactment of the service, the nurse will try to exhaust a range of straightforward possibilities for recovery from the abnormal scenario. If the nurse is unable to resolve the problem they will contact the physician to ask for advice and support. The physician may do something to recover from the exception, or may instruct the nurse. In case of delegation with supervision there is also the possibility that the physician discovers abnormal events while supervising the task enactment for which the nurse has performed no recovery actions. In this case the physician, who was accountable for the exception, will also take responsibility of recovering from the exception. In Section 4.1 we provide another example of chains of collaborative interactions.

The last example that we will give corresponds to the delegation of the management of an exception. In this example a haematologist is treating a patient for amyloidosis with thalidomide. During the treatment he monitors the patient's heart rate because one of thalidomide's side effects is lowering of heart rate. It happens that the patient's heart rate becomes lower than recommended so he decides to delegate the task of managing this exception to a cardiologist. The haematologist cannot cope with this exception himself but he is competent to delegate it to a cardiologist while retaining the accountability for the cardiologist's actions for recovering from the exception.

2.2. Selection of a formalism for specifying clinical guidelines and handling medical errors

To specify the patterns we use the formalism presented by Grando et al. [6], which is based on the task structure of the PROforma clinical workflow language [21]. We have chosen this formalism because it has features that make it suitable for specifying process-based patterns describing recurrent normal and abnormal clinical workflows. The features are:

Workflow-based: a workflow is a network of nodes called *keystones* with unique starting and ending points, connected by any of the following scheduling constraints: sequential composition, XOR-split, AND-split, XOR-join and AND-join. These scheduling constraints are provided with formal, unambiguous semantics based on Petri Nets. A keystone is an abstraction of tasks and goals. At run-time it is assigned a *state*. Initially a keystone's state is Dormant and it changes to state InProgress when it is ready for enactment. The keystone only changes from InProgress to Completed when the *successCondition* is satisfied. While a keystone is InProgress it can be Discarded or Suspended. After a task is Suspended it can resume its execution from the point where its enactment was suspended, while if it is Discarded it needs to be reinitialized for its enactment

A keystone can be assigned a *precondition*, i.e. a predicate that must be satisfied before the keystone becomes InProgress. A keystone can be cyclic, in which case the *cycleInterval* indicates its frequency of iteration. Time constraints can be attached to a keystone like: *startAt*, *duration* and *finishAt*. Those attributes indicate the starting time, duration and completion time for the keystone.

Tasks are keystones that can change the workflow state (including the flow of control in the workflow and the state of the environment) when enacted. The task can be: *decision*, *enquiry*, *action*, or *plan* (workflows comprising activities and goals). Each type of task can have its own attributes.

Goal-based: the framework allows goals to be specified that are linked at run time to recommended tasks that may be executed to achieve them. Goals are keystones that represent temporal patterns of state variables which are not satisfied but could be *achieved*

in the future, or need to be *maintained* for some period of time or indefinitely. When the goal is *InProgress*, a decision-support system proposes candidatePlans for satisfying the goal. The candidates are those plans that are retrieved during run time from a repository of plans and whose enactment can lead to the goal's achievement.

Provides an exception handler: exceptions can be linked with goals to manage them, where these goals are linked with tasks for recovering from errors or monitoring hazardous situations. To achieve a link between tasks, goals, monitored effects, and exceptions, the definition of goals and exceptions is state-based. The exception handler comprises a hierarchical catalogues of state-based exceptions and run-time monitoring systems that check the execution state of CIGs to recognize states corresponding to exceptions for which handlers exist in the catalogue. The exception's *condition* corresponds to the state that needs to be satisfied in order to trigger the exception. The *exceptionType* determines whether the exception is a hazard or an obstacle. A *hazard* corresponds to a state that can *potentially* produce harm to the patient. For instance, a hazard might be detected when the user decides to deviate from the most recommended plan for achieving goal, when the recommended choice in a decision task is overridden, or when some known side effect needs to be monitored. An *obstacle* corresponds to a state where nominal execution of the guideline is not possible. For example, the user prescribed the recommended treatment but the patient did not respond as expected, or the user started to perform a recommended task at a certain time but could not finish it because the resources required were not available. The *isA* attribute is used to refine exceptions and provide hierarchical specifications. This attribute has the standard semantics assumed for class-based specifications. The exception's *template* determines the mode of execution of candidate plans. When a *parallelExecuting* hazard is triggered it does not affect the state of any keystone. But if it is a discarding (suspending) obstacle is triggered it changes the state of a set of keystones to *Discarded* (*Suspended*). Once an exception is triggered a goal (*triggeredGoal*) is activated to start the exceptional flow that deals with the exception. The actor responsible and accountable for enacting the exceptional flow needs to be indicated as a parameter of *triggeredGoal*. The keystones that were *Suspended* can go back to the state they held at the time the exception was triggered only after the exceptional flow triggered to manage the exception is *Completed*.

The catalogue of exceptions that we have provided for the assignment and delegation patterns (see Tables 5, 12 and 14, which will be explained later in detail) satisfies the following property:

Property 8. *Each exception from the provided catalogue is specified such that the actor responsible for achieving the goal (triggeredGoal) that was triggered in order to handle the exception is the same actor who enacted the keystones or goals that triggered the exception.*

A prototype of this framework, apart from the exception manager module, has been developed using the COGENT Cognitive Modelling system [22]. The COGENT tool was used because it allows development and enactment of symbolic and hybrid models of cognitive processes without commitment to a particular architecture.

In the next section we use this framework to define patterns for assignment and delegation of services. A simplification of the introduced patterns, which does not consider the occurrence of exceptions, has been implemented using the COGENT prototype.

3. Results

The patterns that we propose as generic templates for assignment and delegation of tasks can be seen as the hierarchical skeletal plans introduced by Friedland et al. [7]: a sequence of generalized steps, which, when instantiated by specific operations in a

specific problem context, will solve a given problem. The idea of using skeletal plans to share and reuse procedural knowledge while leaving room for execution-time flexibility is not new [6–10]. There are some tools available for specifying medical workflows based on skeletal plan refinement including Asbru [23], EON [24], the MOLGEN project [7], ONCOCIN [25], T-HELPER [26] and DILEMMA [27]. Additionally, mechanisms such as heuristics [7], specialization rules [28] and argumentation systems [29] have been proposed to suggest at run time concrete plans for enacting skeletal plans.

3.1. Terminology of the specification of the patterns

In Fig. 1 we present the terminology used for the specification of the delegation and assignment patterns. The patterns are defined as an extension of the goal-based framework of [6] explained in Section 2.2, therefore they inherit all the concepts (keystone, task, goal, actor, role) and data types (boolean, string, integer, real, time stamps, etc.) from it. While the inherited concepts are shown in the UML Class Diagram [30] (Fig. 1) in the “package workflow-model” the types are not shown. While the patterns are defined in terms of basic sets of attributes (properties), the pattern instantiation in a particular context allows the introduction of more specific additional attributes.

In formalisms used for the specification of clinical guidelines, like PROforma [21], Asbru [22], and the framework presented by Grando et al. [1], it is possible to indicate the preferred actor or role that is responsible for the enactment of a task or goal. But these formalisms have not contemplated the possibility of providing more complex specifications of actor, roles or the introduction of organizational groups or policies (positive or negative authorizations) which restrict the actors/roles/groups of actors who could carry out a task. Information related to actors, roles and groups is vital for the correct implementation of guidelines by health care teams. Therefore we propose here definitions that could be adopted by existing formalisms used for the specification of guidelines in order to provide the required expressiveness. The specification of an actor's particular features should be descriptive enough such that it would be possible to determine whether the actor can play a role, belong to a group, satisfy/not satisfy a policy, be competent to provide a service.

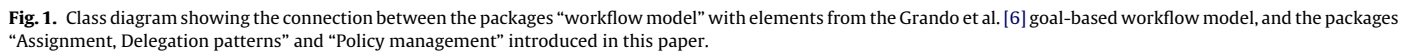
Below we explain in detail each of the terms presented in Fig. 1.

3.1.1. Actor, role and group

Type **Actor** as <name, roles, groups, features, policies> where:

- *name*: uniquely identifies the actor.
- *Roles, groups*: set of role(group) names that the actor can play (can belong to).
- *features*: set of pairs (name, value) that characterize the actor and can be used to check if the actor can play a role (e.g., (name:has_gp_degree_gp, value:true) is a feature that certifies that the actor can play the role of gp). They can be also used to check if the actor complies with a policy's condition.

For instance another type of feature is the personnel working schedule which indicates which working days and times the members of the staff have been assigned. Nowadays there are multiple computer-based applications available which could be chosen by hospitals to record the personnel's working schedule. The working schedule assigned or chosen by the staff can be used to check if the actor complies with the hospital's temporal policies and time-related working regulations. An example of work regulation is the full time working schedule (which means that the person should work for example 38 h a week, maximum 6 assignments to the night shift type, maximum 2 weekends per month); other work



```
feature.getName() == "has_gp_degree"&& feature.getValue() == true.
```

Similarly we define the type **Group** as $\langle name, policies, satisfyRequirements \rangle$.

- For instance we can define the group with name *SNL_biospy_surgeons*, corresponding to the surgeons who can perform sentinel lymph node biopsies. For this group the function *satisfyRequirements* returns true for an *actor* iff

```
actor.getRoleName()=="surgeon" &&  
actor.getFeatures().contains(feature) &&  
feature.getName()=="SNL_biopsy_skill" &&  
feature.getValue()== true.
```

3.1.2. Service request

Each time an actor called client wants to assign or delegate a keystone he starts by sending a service request, defined as `<service, type, completion, client, postedDate>`, where:

- *service*: is the keystone that is assigned/delegated to the provider by the client.
- *type*: indicates the type of service requested and can take the values *assg*, *deleg*, *sdeleg* indicating assignment, delegation without supervision and delegation with supervision.
- *completion*: this boolean function is given by the client to the provider during the service assignment to check if the service satisfies the client's criteria of service completion.
- *client*: identifies the agent that requires the service.
- *postedDate*: date when the service is requested by the client.

3.1.3. Contracts

When the client and provider decide to participate in a service assignment or delegation a contract is awarded. A contract is defined as a tuple

`<service, type, completion, client, provider, startDate, finishDate>` where:

- *service, type, completion, client*: are interpreted as in service request.
- *provider*: identifies the agent that is going to provide the service.
- *startDate*: is the date the contract starts.
- *finishDate*: corresponds to the date the contracts finishes. Always *finishDate* > *startDate*

Depending on the type of service (assignment, delegation with or without supervision) an open contract (with start day and without finish date) signed by a client and a provider determines precisely the level of responsibility and accountability of the actors on providing the service and managing the exceptions associated with the service enactment. When we refer to a contract signature we do not assume any specific semantic: it can be cryptographic signing, some atomic action that formally confirms consent to act and formally identifies the signee, etc.

3.1.4. Policies

A *policy* is a statement enabling or constraining execution of some *keystone* (task or goal) by one or more *actors*, *roles* or *groups* depending on the *conditions* of the context of execution (for instance *time constraints*). Policies in general could be *positive* or *negative authorizations*. Policies can be given *priorities*, for instance ordering them in a system of preferences. Because our patterns should be as generic as possible we do not assume any specific policy management system, but we define the patterns in terms of an abstract UML Class Diagram shown in Fig. 1. The policy framework that we present here is an abstraction and generalization of the common features of two ontology-based general-purpose policy management systems called KAOs [31] and Rei [32]. These policy management systems are generic and do not make any assumptions on the system implementation. For instance KAOs can be used for policy management of software agent applications, grid computing technologies and web services environment.

A policy is defined as a tuple `<name, condition, keystones, priority>` where:

- *Name*: uniquely identifies the policy.
- *Condition*: a condition that should be satisfied in order to allow the policy's activation. Every condition is defined with a name and a Boolean function that returns true when the condition is satisfied. Examples of types of conditions are

- *Skill Category*: determines a class of staff who has a particular level of qualification, skill or responsibility to perform a task. For instance surgeons who can perform sentinel lymph node biopsies.
- *Coverage Constraints*: express the number of personnel needed for every skill category and for every shift or time interval during the entire planning period. For example in every time shift one head nurse is mandatory per each 15 regular nurses.
- *Time Related Constraints*: all the restrictions on personal schedules. Like the personal requests, personal preferences, and constraints on balancing the workload among personnel. For an extensive bibliographic survey on types of temporal constraints used for hospital personnel scheduling, methodologies for temporal policy decisions and temporal management decision making we recommend [33].
- *Work Regulations*: it refers to the contract that personnel members have with the hospital. For example a full-time or half-time contract.
- *Keystones*: set of keystones that are affected by the policy.
- *Priority*: this is an integer and it is optional. It allows ordering the policies in a system of preferences. For instance if `priority(policy1) > priority(policy2)` then policy1 is preferred over policy2, otherwise policy1 is preferred.

Policies can be classified as positive or negative authorization to enact a set of keystones, as indicated by the specialization of class *Policy* into subclasses *NegativeAuthorization* and *PositiveAuthorization* (see Fig. 1).

The negative and positive authorizations and the required skills (role's and group's function *satisfyRestrictions()*) for some health registered professional are regulated by statutes and regulatory bodies, see for instance the one from the British Dietetic Association and Royal College of Nursing [19]. 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. For instance a hospital can limit the number of delegations/assignments in the chain of team collaborations by defining a negative policy which does not allow a health worker to subdelegate/subassign a service if the same service has been subdelegated more than once in the same team.

The policy management system should provide functions to determine:

- (1) conflicts between two sets of policies. Because an actor can play multiple roles, conflicts between roles' policies can arise. The same can happen for groups. For instance it can be the case that a role (or group) forbids an actor to perform a keystone while another role (or group) authorizes the actor to perform it. Similarly there can be conflicts between the actor's policies and the policies associated with the roles the actor is playing or the groups the actor belongs to;
- (2) an actor's competence to perform a service (keystone) based on the actor's features, the service's constraints, and the actor's, roles' and groups' policies;
- (3) the set of actors who are competent to provide a service.

Because our purpose is to specify very generic and reusable patterns for collaborative work, we do not make any assumption on the semantics of these functions. But when the patterns are instantiated these functions need to be defined. Below we provide some examples of possible C++ function specifications:

- (1) `bool areConflicting(policy authorized[], policy nonauthorized[])`

Boolean function that takes as input parameters two arrays of policies. This function can be defined based on the following principles:

- (a) If a keystone1 with priority1 is in the set of *nonauthorized* and in the set of *authorized* then there is a conflict.
- (b) If a keystone1 with priority1 is in the set of *authorized* and the same keystone with priority2 is in the set of *nonauthorized* and priority1>priority2 then there is no conflict.
- (c) If a keystone1 with priority1 is in the set of *authorized* and the same keystone with priority2 is in the set of *nonauthorized* and priority1<priority2 then there is a conflict.
- (d) If a keystone1 with priority1 is in the set of *authorized* and the same keystone with priority2 is in the set of *nonauthorized* and priority1==priority2 then there is a conflict.

- (2) bool **isCompetent** (actor actor, keystone service, function: completion)=canEnact(actor, service) || canAssign(actor, service, completion) || canDelegate (actor, service, completion) where:

- function *canEnact* takes as parameter an *actor* and a keystone and it returns true if the actor is competent to perform the *service* himself, otherwise it returns false.
- function *canAssign*(*canDelegate*) takes as parameter an *actor*, a keystone and a *completion* function and they return true if the *actor* is competent to assign(delegate) the *service* to other provider, otherwise it returns false.

We differentiate between the actor's competence to perform the service himself vs. to assign or delegate it. An actor is competent to perform a service himself (canEnact(actor, service)) if and only if: (a) using the actor's *features* the service constraints are satisfied (for instance the provider can enact a keystone at its *startTime*) and the conditions associated with the policies that control the service enactment are also satisfied, and (b) there is no conflict between the policies that control the service enactment and the policies that apply to the actor (his own policies and those inherited from the role he is playing and group that he belongs to).

An actor is competent to assign(delegate) a service whose termination criteria is given by the function *completion* (canAssign(actor, service, completion) or canDelegate(actor, service, completion)) if and only if: (a) there is a positive policy that determines he is authorized to delegate or assign the service; (b) there is no negative policy that determines that he could not assign or delegate the service; (c) the actor is responsible and/or accountable for the task that he wants to assign/delegate (he could have inherited the responsibility and accountability from adopting the role of service provider during a delegation or assignment in a chain of delegation or assignment – see the example of chain of collaborations given in Section 2.1);

We do not require that the actor has the knowledge and skills to perform the service that he is assigning/delegating, for example take the example given in Section 2.1 when the haematologist cannot cope with patient's low heart rate but he can delegate managing this adverse event to the cardiologist. In addition we do not require that the actor has the knowledge and skills to resolve himself abnormal situations or exceptions that arise from the enactment of the service that he is responsible or accountable to provide. If an actor is not competent to carry out actions to recover from an exception he can inform the client (in case the actor is a provider in a delegation) or he can assign/delegate the actions required to recover from the exception to a competent provider.

Besides defining its own conflict resolution strategies, an organization could reuse existing like the ones provided by the policy management systems KAOs and Rei. These mentioned policy management systems are provided with reasoning mechanisms to detect policy conflicts (for instance an actor has

Table 1
Plan client_assignment_pattern.

Name	client_assignment_pattern
Parameters	service: Keystone, completion: function
Precondition ^a	this.GetActor().GetContracts(service, anytype, completion, this.GetActor(), anyprovider, start, null)== null && this.GetActor().GetServRequests (service, anytype, completion, this.GetActor(), anydate)==null && canAssign(this.GetActor(), service, completion)
successCondition ^b	this.GetActor().GetContracts(service, "assg", completion, this.GetActor(), provider, start, null)== Contracts && Cardinality(Contracts)==1 && provider!= this.GetActor() && && this.GetActor().GetServRequests (service, "assg", completion, this.GetActor(), anydate)==null && this.GetActor().GetContracts(service, "assg", completion, null, anyprovider, start, null)== null

^a The actor has no contract awarded for the service that he tries to assign, he has not requested the service (via assignment or delegation) to any provider and according to the function *canAssign* he is allowed to assign the service to someone else.

^b The contract between the client and the selected provider has been closed, and there is no service request or open contract for the assigned service.

positive and negative authorization to perform a keystone) or to resolve policy conflicts (for instance based on policy priorities). In KAOs and Rei the enforcement is based on a decision engine that uses deductive reasoning to infer the rights and obligations of objects in the managed system in response to requests that specify the current state of the system.

- (3) actor [] **ObtainCompetentProviders** (Keystone service, actor staff[], function completion)

This function returns all the actors from the list (array) of actors *staff* who according to function *isCompetent* are competent to provide the *service* himself or by assigning/delegating the service to other competent provider.

3.2. Assignment pattern

For abstracting the problem of assigning services (tasks or goals) we introduce the plan *client_assignment_pattern* (Table 1, Fig. 2(1)) and the plan *provider_assignment_pattern* (Table 2, Fig. 2(2)). These plans allow specification of the transfer of responsibility and accountability for providing a service from an actor called *client* to another actor called *provider*.

We specify the plan *client_assignment_pattern* such that the client can only make an assignment if he has the competence to assign the service to a provider and there is no service request or open contract of any type for the same service between the client and any provider.

As shown in Fig. 2, this pattern consists of the goal *achieve_serv.requested* (Table 3) which succeeds when the client opens a service request, followed by the goal *achieve_contract.awarded* (Table 4) which succeeds when the client has awarded a contract to a provider, has removed the open contracts from unselected providers and has removed the service request that originated the contract.

We assume that when the client wants to assign a service, he creates a service request, and those actors who are competent to perform the requested service (themselves or by assigning/delegating the service to other competent provider) receive the service request. The providers who decide to participate in the service assignment can sign a contract for the requested service. From all the signed contracts that correspond to competent providers (as specified by function *ObtainCompetentProviders*) the client can choose one, sign it and award the contract. Row 1 in

Table 2
Plan provider_assignment_pattern.

Name	provider_assignment_pattern
Parameters	service:Keystone, completion:function
Precondition ^a	<pre> this.GetActor().GetContracts(service, anytype, completion, client, anyprovider, anystart, null)!=null && this.GetActor().GetServRequests(service, "assg", completion, client, date)!=null && isCompetent(this.GetActor(),service, completion) && this.GetActor()!=client </pre>
successCondition ^b	<pre> (decision==accept && this.GetActor().GetContracts(service, type, completion, client , this.GetActor(), start,end)!= null) (decision==accept && otherProvider!=this.GetActor()&& this.GetActor().GetContracts(service, type, completion, client , otherProvider, start, null)!= null && this.GetActor().GetContracts(service, type, completion, null , this.GetActor(), start, null)!= null) (decision==reject && this.GetActor().GetContracts(service, type, completion,null, this.GetActor(), start, null) == null) </pre>

^a The provider is competent to perform the assigned service (doing it himself or assigning/delegating it to a competent provider), there is no contract of any type for the same service between the client and any provider, and there is a service request posted by the client to request the service.

^b There are three possibilities: (1) the actor offered to provide the service and he was selected as provider, (2) the actor offered to provide the service but he was not selected as a provider, or (3) the actor rejected to provide the service.

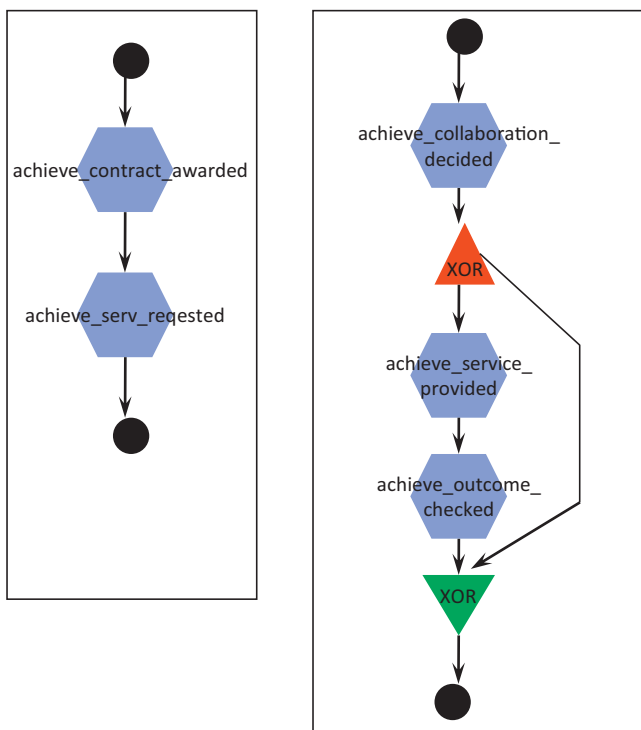


Table 3

Goal achieve_serv_requested.

goalType	Achieve
Name	achieve_serv_requested
Parameters ^a	service: Keystone, type:ServiceType, staff: actor[], completion: function
Precondition ^b	<pre> this.GetActor().GetContracts(service,anytype, completion, this.GetActor(), anyProvider,anystart, null) == null && this.GetActor().GetServRequests (service, type, completion, this.GetActor(), date)!=null </pre>
successCondition ^c	<pre> this.GetActor().GetContracts(service,anytype, completion, this.GetActor(), anyProvider,anystart, null) == null && this.GetActor().GetServRequests (service, type, completion, this.GetActor(), date)!=null </pre>

^a ServiceType is introduced as a new type that can take the values: assg, sdeleg, deleg corresponding to assignment, supervised delegation and unsupervised delegation.

^b The client has not sent a service request or opened a contract with any provider for this service and he is competent to assign the service.

^c A service request has been sent by the client., the contract between the client and the service provider remains open.

there is no ongoing contract between the client and any provider, and the provider is competent to perform the requested service (himself or by assigning/delegating the service to other competent provider). This pattern consists of three goals. When the pattern is activated, the first goal, *achieve_collaboration_decided* (Table 6), becomes active. The provider can choose to provide the service or not. If the provider decides to provide the service he opens and signs a contract. Then the second goal, *achieve_service_provided* (Table 7), becomes active if the provider is chosen by the client. Row 4 in Table 5 describes an exception that can take place during the service enactment. Completion of that goal invokes the third goal, *achieve_outcome_checked* (Table 8), which is achieved when the provider confirms that the service has been completed according to the client's criteria. If the client's completion criteria are satisfied the provider closes the contract and the plan *provider_assignment_pattern* is completed. Row 2 in Table 5 specifies an exception that occurs when the client's criterion of service completion is not satisfied. The plan *provider_assignment_pattern*

Fig. 2. (1) Client_assignment_pattern, (2) Provider_assignment_pattern. The hexagons represent goals, the triangles corresponds to split points and the inverted triangles to join points.

Table 5 describes an exception that occurs when no contract is awarded. The plan *client_assignment_pattern* is completed when the contract between client and provider is closed, the client has removed the service request which originated the closed contract and the client has removed the contracts opened by unselected providers.

We specify the plan *provider_assignment_pattern* (Fig. 2(2)) such that it is activated when the provider receives a request from a possible client to participate in a collaborative task of type assignment,

Table 4
Goal achieve_contract_awarded.

goalType	Achieve
Name	achieve_contract_awarded
Parameters	service: Keystone, type:ServiceType, staff: actor[], completion: function
Precondition ^a	this.GetActor().GetContracts(service,anytype, completion, this.GetActor(), anyProvider,anystart, null) == null && this.GetActor().GetServRequests (service, type, completion, this.GetActor(). date)!=null
successCondition ^b	this.GetActor().GetContracts(service,type,completion, this.GetActor(), provider, start, null)== Contracts && Cardinality(Contracts)==1 && provider!=this.GetActor() && ObtainCompetentProviders(service, staff, completion). Contains(provider) && this.GetActor().GetServRequests(service, type, completion, this.GetActor(), date)== null && this.GetActor().GetContracts(service,type, completion,null, anyprovider, start, null)== null

^a The client has sent a service request but he has not signed a contract with any provider for this service.

^b A contract between the client and the provider has been opened and both have signed it. In the signed contract the provider is different from the client and the provider is competent to perform the service (the actor is contained in the set of actors competent to perform the service). The service request that originated the contract has been deleted. Besides all the contracts opened by providers who were not selected are removed.

is completed when the actor rejects to provide the service, or the provider completes the service and fixes the termination date in the contract, or the actor offered to provide the service but he was not selected as provider by the client.

3.3. Properties satisfied by the assignment pattern

The analogy between concurrent programs as transformers of predicates and our patterns as concurrent planning systems is clear: just as each statement in a program modifies the state of the system, every enacted keystone modifies the state of the system. As programs run concurrently interacting with other programs, keystones (tasks and goals) in the client plan run concurrently interacting with keystones in the providers' plans of the same pattern. Just as a program is a composition of (sequential, iterative, conditional, concurrent, etc.) statements, a plan from framework [6] is a composition (via scheduling constraints) of keystones. In a program a statement *S* transforms a *precondition* *P* into a *postcondition* *Q* (denoted as the Hoare triple $\{P\} S \{Q\}$), such that the *precondition* describes the set of initial states in which the statement is started and the *postcondition* describes the set of desirable final or output states. Similarly in our patterns the completion of the keystones should transform the system's state described by the keystone's *preconditions* into a desirable state expressed by the keystone's *successConditions*. Due to these similarities we can map the patterns for assignment and delegation of tasks into equivalent programs. While this mapping is not automatic and it has to be done once per each of the proposed pattern it allows to prove desirable that which are preserved each time the patterns are reused. Once the patterns are mapped into programs formal verification techniques available in the programming field can be used. A verification system is defined as a set of verification rules, one rule for each type of statement in a programming language. The correctness of $\{P\}S\{Q\}$ is proved in the following way: using the verification rules a set of predicate logic formulas called verification conditions are generated. The proof of these verification conditions ensures the correctness of a program (in our case the patterns).

Table 5
Catalogue of examples of exceptions that can be triggered during the assignment and delegation of tasks in team work.

Exception	Type	Triggered by	Exception's effects	Candidate plans for new goal
<i>unawarded contract</i>	Discarding obstacle	Timeout of goal <i>achieve_contract_awarded</i>	goal <i>achieve_contract_awarded</i> is discarded, the actor responsible for the discarded goal (the client of an assignment or delegation) becomes responsible for the new goal <i>achieve_new_contract_reconsidered</i>	(1) The actor decides to aborts the pattern (assignment of delegation) that generated the discarded goal (2) The actor considers other potential providers (3) The actor relaxes the restrictions associated to the contract
<i>unsatisfied service completion</i>	Suspending obstacle	Client's criterion of service completion is not satisfied after the service has been provided	goal <i>achieve_outcome_checked</i> is suspended, the actor responsible for that goal (in assignments the provider, in delegations the client) becomes responsible for the new goal <i>achieve_unsatisfied_service_completion_resolved</i>	The client and provider communicate in order to understand the reasons for the unsatisfaction of the completion criteria
<i>unresolved_exception_during_delegation</i> (Table 11)	Hazard	The provider of a delegation decides to inform the client when he cannot deal with an exception	The provider is responsible for the new goal <i>achieve_exception_informed_to_client</i>	Depends on the way the provider can inform the client, for instance by filling a form, by making a telephone call, etcetera.
<i>failure_hemodialyzer</i> (Table 12)	Suspending obstacle	The hemodialyzer shows a message of error	Goal <i>achieve_service_provided</i> is suspended and the actor responsible for the goal (the provider) is in charge of the new goal <i>achieve_failure_assessed</i>	Check the cause of the hemodialyzer failure. For instance a possible cause is the presence of air bubbles in the circuit, which can increase the risk of air embolism.
<i>symptoms_air_embolism</i> (Table 13)	Hazard	The patient shows symptoms of air embolisms, for instance difficulty to breath	If the actor who is assessing the patient is not competent for solving the exception and he has been delegated the tasks he needs to inform the client activating the goal <i>achieve_exception_informed</i>	The client of the delegation is called and informed about the high risk of an air embolism.

Table 6
Goal achieve_collaboration_decided.

goalType	Achieve
Name	achieve_collaboration_decided
Parameters	client: Actor, service: Keystone, type: serviceType, completion: function
Precondition ^a	<pre> this.GetActor().GetContracts(service, anytype, completion, client, anyprovider, anystart, null)!=null&& this.GetActor().GetServRequests (service, type, completion, client, date)!= null </pre>
successCondition ^b	<pre> (decision==accept && this.GetActor().GetContracts(service, type, completion, null , this.GetActor(), startTime,null)!= null (1)) (decision==reject && this.GetActor().GetContracts(service, type, completion,null, this.GetActor(), startTime,null) == null (2)) </pre>

^a The actor has opened a service request in the role of client for a service, but no contract has been opened.

^b Two situations can happen: (1) the actor decides to collaborate, he adopts the role of a provider and opens and signs a contract for the service; (2) the actor decides not to collaborate and no contract is opened.

Table 7
Goal achieve_service_provided.

goalType	Achieve
Name	achieve_service_provided
Parameters	service: Keystone, type: serviceType, client: actor, completion: function
Precondition	this.GetActor().GetContracts(service, type, completion, client, this.GetActor(), startTime, null) != null
successCondition	service.GetState()==Completed

Table 8
Goal achieve_outcome_checked.

goalType	Achieve
Name	achieve_outcome_checked
Parameters	service:Keystone, type: serviceType, completion: function, client: actor, provider:actor
Precondition ^a	<pre> this.GetActor().GetContracts (service, type, completion, client, provider, startTime,null) !=null && service.GetState()==Completed&& [(type=="assg" && provider==this.GetActor()) (type!="assg" && client==this.GetActor())] </pre>
successCondition ^b	<pre> completion(service.GetSuccessCondition())==true && this.GetActor().GetContracts (service, type, completion,client, provider, startTime,now())!=null </pre>

^b The service is completed by the provider (in case of assignment) or by the client (in case of delegation).

^a The client is satisfied with the completion (according to *completion*) and the contract is closed.

Here we have chosen the Dijkstra's Guarded Command (DGC) language [34] for translating the assignment/delegation patterns into programs. From the DGC language we have used the following elementary statements:

- *skip*, which does nothing, it does not change the system's state.
- *sequential composition*: a statement S_1 is executed only after statement S_0 is executed, denoted as $S_0; S_1$.
- *alternative construct*: used for nondeterministic conditional selection of n statements S_i . Denoted as:

```

if
    B0 then S0
    | ...
    | Bn then Sn
fi

```

Where the B's are called guards and the S's are the guarded statements. Any guard B_i that evaluates to *true* is nondeterministically selected, and the associated guarded statement S_i is performed. In a concurrent program the alternative construct where all the guards are false is equivalent to a busy wait, i.e. it keeps evaluating until one of the B's becomes true.

Table 9
Plan client_delegation_pattern.

Name	client_delegation_pattern
Parameters	service:Keystone, completion: function
Precondition ^a	<pre> this.GetActor().GetContracts (service, anytype, completion, this.GetActor(), anyProvider, startTime, null)==null && this.GetActor().GetServRequests (service, anytype, completion, this.GetActor(),anydate)==null &&canDelegate(this. GetActor(),service,completion) </pre>
successCondition ^b	<pre> this.GetActor().GetContracts (service, type, completion, this.GetActor(), provider, start, end)==Contracts &&Cardinality(Contracts)==1 && provider!= this.GetActor() && && (type=="deleg" type=="sdeleg") && && this.GetActor().GetServRequests (service, type, completion, this.GetActor(),anydate)==null && this.GetActor().GetContracts (service, type, completion, null, anyprovider, anydate, null)==null && completion(service.GetSuccessCondition())==true </pre>

^a The actor has not posted any service request or awarded any contract awarded for the service that he tries to delegate and according to the function *isCompetent* he is allowed to delegate of the service to someone else.

^b The delegation contract (with or without supervision) between the client and the provider has been closed, there is no service request or opened contracts for the delegated service and the service has been successfully completed according to the client's completion criteria.

We have added to the DGC language a statement corresponding to the parameterized call to subprograms, which we use to map keystones from the framework [6]. If a keystone K has *precondition* P and *successCondition* Q we consider that the Hoare triple $\{P\} ProgramK\{Q\}$ resulting from mapping the keystone K into subprogram $ProgramK$ is correct, i.e. when the $ProgramK$ is enacted with *precondition* P its completion changes the system's state into the state described by the *successCondition* Q .

As an example we show below how we mapped into an alternative construct in the DGC language the workflows from the plan *provider_assignment_pattern* (Fig. 2) which could be enacted when the XOR split scheduling constraint is evaluated:

```

if decision      then
==accept
    achieve_service_provided(service, type, client,
                             completion);
    achieve_outcome_checked(service,
                             type,completion, client, provider)

    |decision
    == reject
    then skip;
fi

```

If the provider's decision is to accept the client's request then sequentially after the subprogram *achieve_service_provided*

Table 10
Plan provider_delegation_pattern.

Name	provider_delegation_pattern
Parameters	service:Keystone, type:serviceType, completion:function
Precondition ^a	<pre> this.GetActor().GetContracts(service, anytype, completion, client, anyprovider, start, null)!=null && (type=="deleg" type=="sdeleg") && this.GetActor().GetServRequests (service, type, completion, client, date)!=null && isCompetent(this.GetActor(),service,completion) && this.GetActor()!=client </pre> <pre> decision==accept && service.GetState()==Completed && this.GetActor().GetContracts(service, type, completion, client, this.GetActor(), start, null)!=null </pre> <pre> decision==accept && otherProvider!=this.GetActor() && this.GetActor().GetContracts(service, type, completion, client , otherProvider, start, null)!= null && this.GetActor().GetContracts(service, type, completion, null, this.GetActor(), start, null)!= null </pre> <pre> decision==reject && this.GetActor().GetContracts(service, type, completion,null, this.GetActor(), start, null) == null </pre> ① ② ③
successCondition ^b	

^a The provider is competent to perform the service delegated (with or without supervision), there is no contract of any type for the same service between the client and provider, and there is a service request posted by the client to request the service.

^b There are three possibilities: (1) the actor offered to provide the service and he was selected as provider, (2) the actor offered to provide the service but he was not selected as a provider, or (3) the actor rejected to provide the service.

(corresponding to the goal *achieve_service_provided*) is called, the subprogram *achieve_outcome_checked* (corresponding to the goal *achieve_outcome_checked*) is enacted. But if the provider's decision is to reject the client's request then the *skip* statement is enacted, which is equivalent to taking no action. This alternative construct is deterministic, because it can never be the case that more than one guard simultaneously evaluates true.

In the verification technique called Owicki–Gries Theory [35] the semantics associated with each statement in the DGC language is given in first order logic. For instance an alternative construct statement transforms a precondition P expressed into a postcondition Q if and only if we can prove the correctness of the Hoare triple

$$\{P\} \quad \text{if} \quad \begin{matrix} B_0 \\ \dots \\ B_n \end{matrix} \quad \rightarrow \quad \begin{matrix} S_0 \\ \dots \\ S_n \end{matrix}$$

According to the Owicki–Gries Theory the previous Hoare triple is correct if and only if we can prove that $P \rightarrow [B_0 \parallel \dots \parallel B_n] \&\& \{P \&\& B_0\} S_0 \{Q\} \&\& \dots \&\& \{P \&\& B_n\} S_n \{Q\}$. Basically the correctness of the alternative construct consists on proving that if the precondition is satisfied then some of the guards will evaluate to true, and that the execution of any of the guarded statements will lead to the satisfaction of the postcondition.

Therefore the proof that a program (in our case a pattern) is correct with respect to a specification (in our case the formal specification of the patterns as workflows of keystones and the specification of the keystones in Tables 1–14) is reduced to first order proofs.

In Fig. 3 we show our translation into a DGC program of the assignment pattern with one client and arbitrary $n \geq 1$ providers. In Fig. 3 the operator $\parallel$ between two programs indicates the concurrent enactment of the corresponding programs. Once the correctness of a DGC program is verified in the Owicki–Gries Theory it can be annotated by indicating for every statement S_j with precondition P_j and postcondition Q_j the corresponding Hoare triple $\{P_j\}S_j\{Q_j\}$.

Table 11
Goal maintain_delegation_supervised.

goalType	Maintain
Name	Maintain_delegation_supervised
Parameters	service: Keystone, provider: Actor, completion: function, cycle: Time
cycleInterval	Cycle
Precondition	this.GetActor().GetContracts(service, "sdeleg", completion, this.GetActor(), provider startTime, null) != null
invariantCondition ^a	completion(service, GetSuccessCondition())!=true && this.GetActor().GetContracts(service, "sdeleg", completion, this.GetActor(), provider startTime, anyfinishTime) == null
successCondition ^b	completion(service, GetSuccessCondition())==true

^a The goal cycles while the service's outcome has not been checked and the contract between client and provider has not been closed.

^b The actor who is performing the goal *maintain_delegation_supervised* finds out that the service has been successfully completed.

In Figs. 4 and 6 we show our translation into DCG annotated programs of respectively the plan *client_assignment_pattern* and the plan *provider_assignment_pattern.i* corresponding to an arbitrary i th provider ($1 \leq i \leq n$) from the set of n providers. The translation consists on: (a) mapping the parameterized goals from the patterns and their preconditions and postconditions into equivalent annotated function calls, and (b) mapping the scheduling constraints and part of the goal's precondition into equivalent alternative constructs, skips and sequential compositions from the DGC language. For instance the parameterized goal *achieve_collaboration_decided* with its precondition and postcondition from the pattern *Provider_assignment_pattern* is mapped into the annotated call to function *achieve_collaboration_decided* with identical parameters in the program *Provider_assignment_pattern.i*. The same as no assumption is made about the goal's enactment, no assumption is made on the function's implementation, but as the annotation indicates after the function enactment the provider should indicate its desire to participate or not in the collaboration

Table 12
Hazard unresolved_exception_during_delegation.

exceptionType	Hazard
Template	Parallel_executing
Name	exception_in_monitored_delegation
Parameters	keystone: Keystone, decideToInform: bool
Condition ^a	$\begin{aligned} & \text{exception.getState()=triggered} \ \&\& \\ & \left(\begin{array}{l} \text{exception.GetTemplate()=discarding(S)} \parallel \\ \text{exception.GetTemplate()=suspending(S)} \end{array} \right)^{(1)} \ \&\& \\ & \left(\begin{array}{l} \text{S.contains(keystone)} \ \&\& \\ (\text{keystone==service} \parallel \text{IsSubComponent(keystone, service)}) \end{array} \right)^{(2)} \ \&\& \\ & \left(\begin{array}{l} \text{keystone.GetActor()=provider} \ \&\& \\ \text{provider.GetContracts(service, type, completion, client, provider,} \\ \quad \text{startTime, null)} \neq \text{null} \\ \&\& (\text{type}=\text{"deleg"} \parallel \text{type}=\text{"sdeleg"}) \end{array} \right)^{(3)} \end{aligned}$
triggeredGoal	achieve_exception_informed_to_client(provider, client, keystone, exception)

^a An exception has been triggered, (1) with template of type discarding or suspending, and (2) the set S of keystones discarded or suspended by the exception contain the service or part of a service, (3) the service has been delegated in a delegation with monitoring, and (4) the client has decided to inform the provider.

Table 13
Suspending obstacle failure_hemodialyzer.

exceptionType	Obstacle
Template	suspending({keystone})
Name	Failure_hemodialyzer
parameters	hemodialyzer:resource, hemodialyzer.failure: trigger
Condition ^a	hemodialyzer.failure.getState()=Triggered && keystone.GetResources().contains(hemodialyzer) && keystone.getState()=InProgress
triggeredGoal	achieve_failure_assessed(keystone.GetActor(), hemodialyzer, hemodialyzer.failure)

^a The hemodialyzer used for the keystone enactment has failed while the keystone was in progress.

(*decision==accept* or *decision==reject*) and a contract should accordingly be opened or not. In the case of the XOR split scheduling constraint from the pattern *Provider_assignment_pattern* it is mapped into the alternative construct with guards *decision==accept* and *decision==reject* in the program *Provider_assignment_pattern.i* based on the disjunctive proposition that defined the precondition of goal *achieve_service_provided*.

Because we prove the program correctness for arbitrary $n \geq 1$ providers and we formally verify an arbitrary i th provider we can claim that the obtained proofs of correctness and property satisfaction for the patterns are valid independently of the number of

Table 14
Hazard symptoms_air_embolism.

exceptionType	Hazard
Template	Parallel_executing
Name	Symptoms_air_embolism
Parameters	assessment:task, air_embolism_symptoms: symptoms[]
Condition ^a	assessment.GetResult()=patient_symptoms. && intersection(patient_symptoms, GetSymptoms(air_embolism_symptoms))!= null && assessment.GetActor()=actor && not (isCompetent(actor, resolve(air_embolism_symptoms))) && actor.GetContracts(service, type, completion, client, actor, anystart, null) != null && (type="deleg" type="sdeleg")
triggeredGoal	achieve_exception_informed(client, patient_symptoms, symptoms.air_embolism)

^a During the assessment the patient is showing symptoms of air embolism. The actor in charge of the assessment is a provider in a delegation, and he is not competent to resolve problems related to air embolism.

providers used at run-time to enact the patterns. We have proved that the annotated programs shown in Figs. 4 and 5 are correct with respect to the formal patterns' specifications even when the following cases are considered:

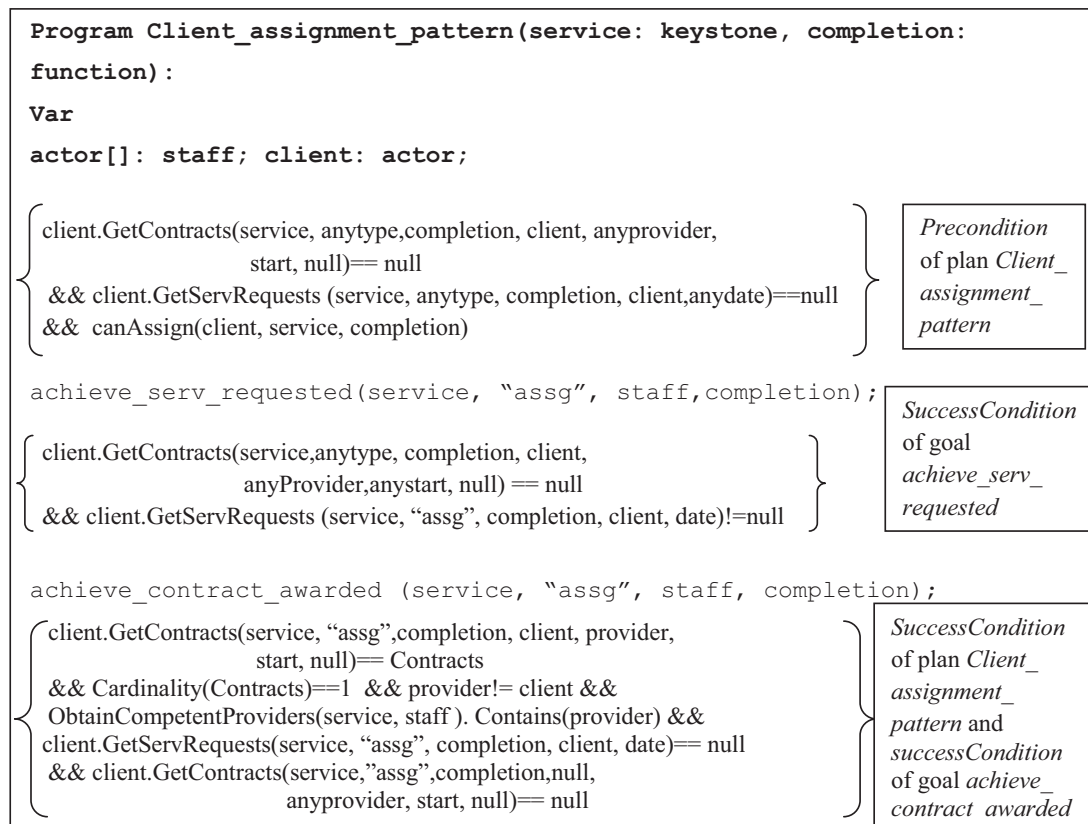
Program Assignment:

Var

```
keystone: service; function: completion;
```

```
Client_assignment_pattern(service, completion)
|| Provider_assignment_pattern_0 (service, completion) || ...
|| Provider_assignment_pattern_i (service, completion) || ...
|| Provider_assignment_pattern_n(service, completion)
```

Fig. 3. Program Assignment, resulting from mapping the Assignment pattern into the DGC language. The symbol $\parallel$ denotes the parallel enactment of the programs *client_assignment_pattern* and $n \geq 1$ programs *provider_assignment_pattern.i* with $1 \leq i \leq n$.

Fig. 4. Annotated program *client_assignment_pattern*.

- (1) No provider accepts to provide the service: in this case of the actors who offered to provide the service by opening a contract complete their *provider_assignment_pattern* but the *client_assignment_pattern* cannot be completed. In this case the exception *unawarded_contract* (top row of Table 5) is enacted.
- (2) More than one provider accepts to provide the service: in this case the client selects only one provider (as indicated by *successCondition* of goal *achieve_contract_awarded*) and the other open contracts are deleted by the client. As indicated by *successCondition* of the plan *provider_assignment_pattern* the providers who were not selected by the client complete the plan *provider_assignment_pattern*.
- (3) The provider who signs the contract wants to assign (delegate) the service: in this case the provider assumes the role of a client who is looking for another provider. The provider who wants to subassign (subdelegate) the service has to be competent to assign (delegate) the service in order to allow the satisfaction of the *precondition* of the plan *client_assignment_pattern* (*client_delegation_pattern*). When this happens, the plan *client_assignment_pattern* (*client_delegation_pattern*) can be chosen to achieve the goal *achieve_service_provided*. In this way chains of assignment/delegations can be enacted. There is no limit on the number of delegations/assignments to be made, but the hospital can have a policy to control the number of delegations/assignments in the chain of collaborations.
- (4) A provider rejects to provide the service: in this case the *successCondition* of the plan *provider_delegation_pattern* (Table 10) is satisfied after the goal *achieve_collaboration_decided* is achieved when the provider rejects to provide the service.

In order to be valid as generic patterns for assignment and delegation the patterns we have proposed must satisfy the properties set out in Section 2.1. Therefore after proving the patterns'

correctness we must verify that from the proof of the patterns' correctness we can deduce the satisfaction of each of the patterns' properties. In this section we explain how we deduce the satisfaction of the properties for the assignment pattern. In Section 3.5 we verify the properties of the delegation pattern.

Precondition 1 The actor called client is competent to assign the service to a provider and there is no assignment/delegation service request or open contract between the client and any provider for the same service.

Informally according to the *precondition* of plan *client_assignment_pattern* (Table 1) this plan gets active only if there is no contract or service request of any type (delegation or assignment) between the client and any provider for the same service request that originates the pattern, and the client is competent to assign the service. The formal specification of *Precondition 1* is equivalent to the *precondition* of the plan *client_assignment_pattern* (Table 1). Because in the chosen framework [6] a plan is only enacted if its *precondition* is satisfied, then *Precondition 1* is satisfied when the plan is activated.

Also in the *precondition* of plan *provider_assignment_pattern* (Table 2) it is checked that no contract of any type exists between client and provider.

Property 1. The provider is responsible for providing the service and the client can check that the provider is competent to provide the service himself, or competent to assigning/delegating the service.

In the case of the provider he will enact the plan *provider_assignment_pattern* (Table 2) only if he is competent to perform the requested service (himself or by assigning/delegating the service enactment) as indicated in the plan's *precondition*. The goal *achieve_collaboration_decided* (Table 6) is part of this plan, and according to the goal's *successCondition* if the provider accepts to collaborate he opens and signs a contract. In the case of the client the pattern is defined in terms of the goal *achieve_contract_awarded*

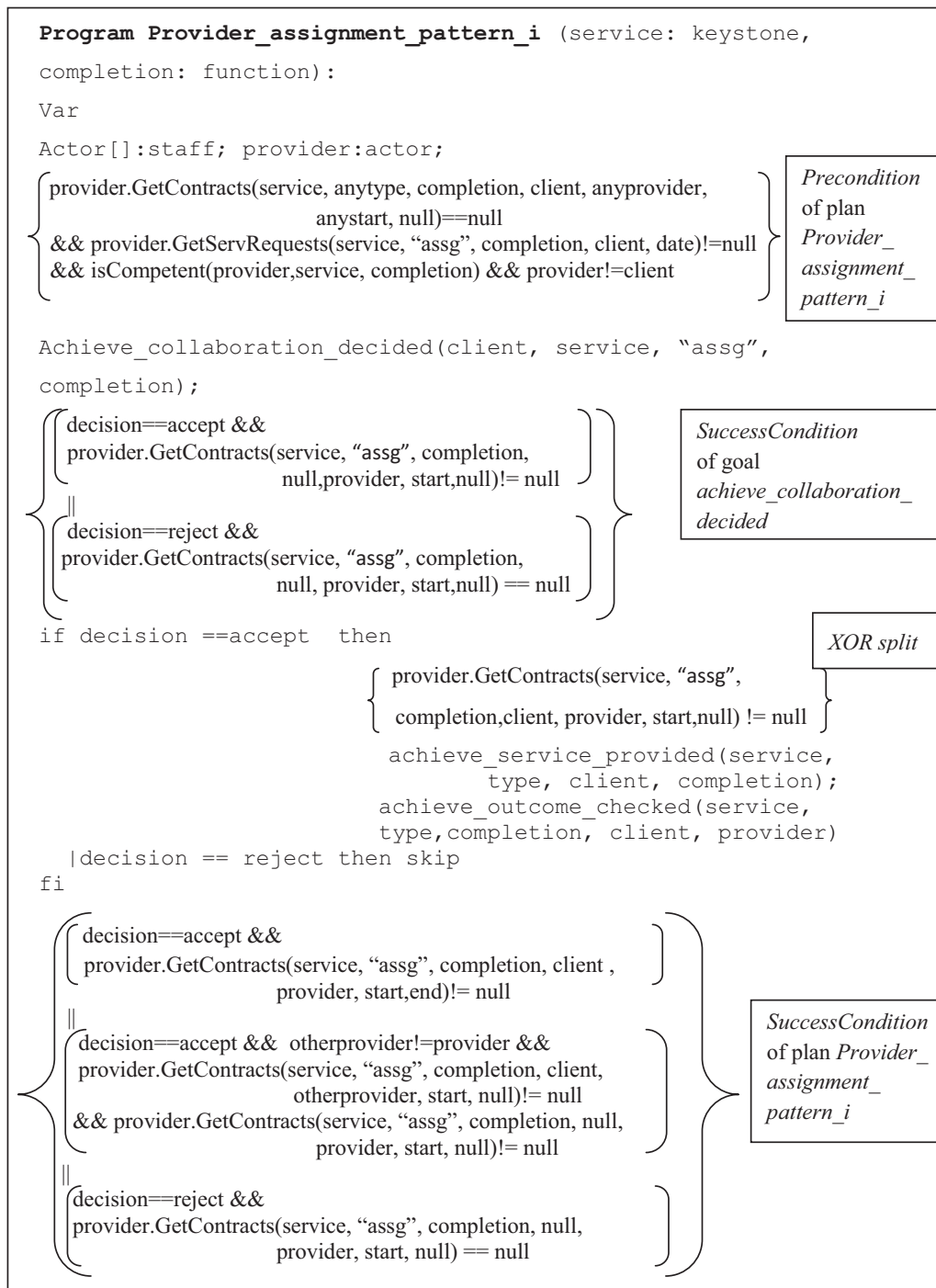


Fig. 5. Annotated program *provider_assignment_pattern.i* corresponding to the *i*.th provider, $1 \leq i \leq n$.

(Table 4). When the goal *achieve_contract_awarded* is achieved *Property 1* is satisfied: according to the goal's *successCondition* the client has checked that the provider is competent (the provider can enact the service himself or can delegate/assign the service to another competent provider) and the client and provider have signed an assignment contract that makes the provider responsible for providing the service.

Property 2. *The provider is accountable for the service's outcome.*

When a provider agrees to perform an assigned service he is provided with criteria to check when the service has been completed (parameter *completion* in goal

achieve_collaboration_decided, Table 6). After that when the client's plan the goal *achieve_contract_awarded* is achieved the *Property 2* is satisfied: according to the goal's *successCondition* the client and provider have signed an assignment contract that makes the provider accountable for the service's outcome.

Besides, the contract between client and provider can only be closed by the provider when he checks that the service has been completed according to the client's completion criteria, as specified by the *successCondition* of goal *achieve_outcome_checked* (Table 8).

Property 3. *The provider is responsible and accountable for any exception arising during the service enactment.*

When the goal *achieve_contract_awarded* (Table 4) is achieved then *Property 3* is satisfied: according to the goal's *successCondition* an assignment contract has been signed between a client and provider, which make the provider responsible and accountable for any exception arising during the service enactment. According to the pattern after the goal *achieve_contract_awarded* is achieved the provider has to pursue the goal *achieve_service_provided* (Table 7). If the repository of exceptions satisfies *Property 8* the provider is responsible for handling exceptions arising from the service enactment. As indicated in Section 2.2 the catalogue of exceptions provided by our exception manager satisfies *Property 8*.

3.4. Delegation pattern

For abstracting the problem of delegating services we introduce the plan *client_delegation_pattern* (Table 9, Fig. 6(1)) and the plan *provider_delegation_pattern* (Table 10, Fig. 6(2)). These plans allow the transfer of responsibility for performing a service to a provider while the client retains accountability for the service outcome.

The precondition for *client_delegation_pattern* is similar to that of the assignment pattern: the client can only delegate a service if he has the competence to do so and there is no ongoing request or awarded contract for the same service with any provider. The *client_delegation_pattern* is shown in Fig. 6(1). The first goal, *achieve_serv_requested* (Table 3) is achieved when the client sends a service request to all the potential providers. The second goal *achieve_contract_awarded* (Table 4) in the plan is achieved as described for the assignment pattern. In the case of exceptions the client retains the accountability but transfers the responsibility of dealing with them to the provider. Only if the provider cannot deal with them will the client become responsible for the recovery strategies.

Once the client has awarded the contract he has to wait for the service completion (provider's goal *achieve_service_provided*, Table 7) to check if the service satisfies its completion criteria (goal *achieve_outcome_checked*, Table 8). The goal *achieve_outcome_checked* is achieved when the client can check that his criterion of service completion is satisfied. Since the provider's criteria of service completion are not necessarily identical to those of the client, the contract between client and provider is still open until the client checks that the service's outcome is the desired one. The plan is completed when the opened contract is closed, the service request that originated the contract is removed, and the opened contracts corresponding to unselected providers are removed.

Two types of delegations are possible: with supervision (*sdeleg*) and without supervision (*deleg*). If the delegation is with supervision, then once the client has awarded the contract he has, besides the goal *achieve_outcome_checked*, the goal of supervising the service enactment (goal *maintain_delegation_supervised* presented in Table 11). The goal *maintain_delegation_supervised* becomes active when the contract has been awarded to the provider (as specified in the goal's precondition) and it consists on periodically supervising the provider (as indicated by attribute *cycleInterval*) while the service's outcome has not been checked and the contract between client and provider has not been closed. The candidate plan for this goal will depend on the type of service. For instance the type of supervision can depend on the trust between client and provider. Any chosen candidate plan will have an associated frequency of supervision (*cycleInterval*). If during the supervision the client detects an exception he is responsible and accountable for that exception. After the goal *achieve_outcome_checked* is completed the client does not need to supervise the provider anymore, therefore the goal *maintain_delegation_supervised* becomes completed (as specified by the goal's *successCondition*) and the plan *client_delegation_pattern* is completed. If during the supervision the

client detects an exception he is responsible for managing it. Possible exceptions during delegation are described in Table 5.

Independently of the type of delegation, when the provider detects an exception and cannot cope with it he needs to inform the provider who activated the goal *achieve_exception_informed*. Once the client is informed of an exception he becomes responsible to manage it and therefore the goal *achieve_exception_recovery_decided* is activated. A possible candidate for this goal is the plan *client_plan_react_to_exception*, shown in Fig. 8.

As in the assignment pattern, the plan *provider_delegation_pattern* becomes in progress when an actor receives the request from a potential client to collaborate in a service delegation, there is no contract between the client and any provider for the service that the client wants to delegate and the actor is competent to perform the requested service (himself or by assigning/delegating the service to other competent provider). This pattern is similar to the provider assignment pattern, except that it does not include the goal *achieve_outcome_checked* which is part of the client's pattern in the case of delegation. A possible exception related to the service enactment is described in rows 3, 4 and 5 of Table 5. If the provider cannot cope with an exception he must inform the client. The plan is completed when the provider has rejected to provide the service, or the provider has offered to provide the service but he was not selected by the client, or when the provider has been chosen by the client to provide the service and according to the provider the service has been completed.

3.5. Properties satisfied by the delegation pattern

For reasons of space we do not provide here the proof in Owicki–Gries Theory of the correctness of the Delegation pattern. Below we only explain informally why the properties in the delegation pattern can be deduced from the pattern's correctness.

Precondition 2 The actor called client is competent to delegate the service to a provider and there is no assignment/delegation open service request or contract between the client and any provider for the same service.

According to the precondition of plan *client_delegation_pattern* (Table 9) this plan becomes active only if there is no contract of any type (delegation or assignment) between the client and any provider for the same service that originated the pattern, and the client is competent to delegate the task. The formal specification of *Precondition 2* is equivalent to the precondition of the plan *client_delegation_pattern* (Table 9). Because in the chosen framework [6] a plan is only enacted if its precondition is satisfied, then *Precondition 2* is satisfied when the plan gets activated.

Also in the precondition of plan *provider_delegation_pattern* (Table 10) it is checked that no contract of any type exists between client and provider.

Property 1. The provider is responsible for providing the service and the client can check that the provider is competent to provide the service himself, or competent to assigning/delegating the service.

In the case of the provider he will enact the plan *provider_delegation_pattern* (Table 10) only if he is competent to perform the requested service (see plan's precondition). The goal *achieve_collaboration_decided* (Table 6) is part of this plan, and according to the goal's *successCondition* if the provider accepts to collaborate he opens and signs a contract. In the case of the client the pattern is defined in terms of the goal *achieve_contract_awarded* (Table 4). According to the goal's *successCondition* the client signs a contract with the provider who had opened and signed a contract if he can check that the provider is competent to perform the service. When the contract is also signed by the client the provider becomes responsible to provide the service.

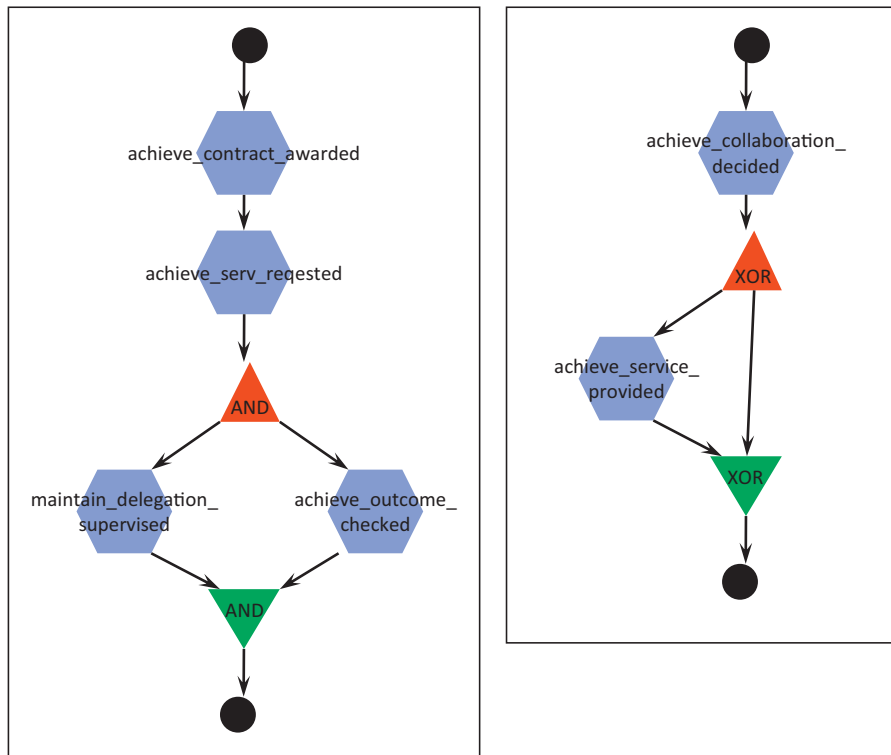


Fig. 6. (1) Client.delegation_pattern and (2) Provider.delegation_pattern.

Property 4. *The client retains accountability for the service's outcome and any exceptions arising from the service enactment.*

When a provider decides to provide the requested service the goal *achieve_collaboration_decided* (Table 6) is achieved and a delegation contract is signed (see goal's *successCondition*) between the client and the provider. This contract makes the client *accountable* for the service's outcome and any exception arising from the service enactment.

As specified by the delegation pattern, the goal *achieve_outcome_checked* (Table 8) is part of the client's workflow. According to the goal's *successCondition* this goal consists on checking that the client's 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. The client's accountability for the service outcome and for any exception arising from the service enactment is reflected in the goal *maintain_delegation_supervised*, which obliges the client to supervise the delegation while the service has not been completed.

Property 5. *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.*

Once the provider has signed a delegation contract he is responsible for the goal *achieve_service_provided* (Table 7).

As indicated in Section 2.2 the catalogue of exceptions provided by our exception manager satisfies Property 8, therefore 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, raising the hazard *unresolved_exception_during_delegation* (row 3 in Table 5). For this goal to be achieved, the provider has to inform the client about the

exception and the recovery strategies that he has unsuccessfully tried.

When the goal *achieve_contract_awarded* is achieved a delegation contract has been signed and the provider becomes responsible for handling any exceptions arising during the service enactment. After the goal *achieve_contract_awarded* is achieved the goal *achieve_service_provided* becomes active. As indicated in Section 2.2 the catalogue of exceptions provided by our exception manager satisfies Property 8, therefore 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, raising the hazard *unresolved_exception_during_delegation* (row 3 in Table 5). For this goal to be achieved, the provider has to inform the client about the exception and the recovery strategies that he has unsuccessfully tried, as indicated by the goal *achieve_exception_informed_to_client* (Fig. 8). For reasons of space we do not provide the formal specification of this hazard and goal.

Property 6. *The client is responsible for managing any exceptions that the provider cannot handle (whether detected by provider or client).*

When the provider achieves the goal *achieve_exception_informed_to_client* the client 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 to the client. In case of supervised delegation the goal *maintain_delegation_supervised* (Table 11) becomes active after a contract is awarded therefore if during the supervision the client detects an exception according to Property 8 he is *responsible* for managing it.

Delegations with supervisions should satisfy the following property:

Property 7. *The client is responsible and accountable for supervising the provider's work while the service has not been completed according to the client's criteria.*

When the client and the provider sign a contract for a delegation with supervision (as specified by *successCondition* of goal *achieve_contract_awarded*) the client becomes *responsible* and *accountable* for supervising the provider's work. In the plan *client_delegation_pattern* the goal *maintain_delegation_supervised* becomes active only when the delegation is with supervision (see goal's *precondition*) and it remains active until the client accepts the service's outcome (see goal's *successCondition* and *invariant*).

4. Usability of the patterns

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 and real applications.

The framework used for the specification of the patterns (Section 4.1) has been prototyped in COGENT [22]. A simplification of the proposed patterns, which does not include abnormal scenarios triggered by exceptions, has been implemented and instantiated in the prototype. In Section 4.1 we show the patterns' instantiation in the framework, using a concrete clinical scenario.

In Section 4.2 we provide the mapping of a part of the delegation pattern to the PROforma language. PROforma can be used for modelling medical processes and be enacted using the Tallis tool [18]. Other languages for specifying and enacting medical workflows could be used for this mapping. In particular PROforma and ASBRU are good candidate languages for mapping the patterns, because: (1) they incorporate some notion of goals or intentions, (2) they include enactment conditions like precondition, termination condition, abort condition, and (3) these languages have an execution engine. Because the patterns' specification is based on the terminology presented in Section 3.1, the patterns' mapping into a concrete modelling language also requires the specification of the mapping between the terms used in the language and the terms explained in Section 3.1.

4.1. Example patterns instantiation

Quoting Headley et al. [37] “Newly diagnosed cases of end stage renal disease (ESRD) have increased by 9% each year since 1970 [in United States]. It has been estimated that there will need to be a significant increase in the number of nephrologists to care for the ESRD population by the year 2010. Recent reports have advocated the use of advanced practice nurses (APN) to collaborate with nephrologists to meet increasing patient care demands. Clinical evidence has supported the financial and clinical advantages of APN utilization in nephrology.” Similar conclusions are presented by Easom et al. [38].

Below we introduce a scenario describing the collaboration between a general practitioner (GP), a nephrologist and an APN for the treatment of a patient with kidney failure. While the GP **assigns** a patient to the nephrologist, the nephrologist **delegates(with supervision)** part of the treatment to the APN. During the delegation an exception takes place and the nephrologist enacts a plan to recover from it. Because the GP assigned the patient to the nephrologist he is not responsible or accountable for the detected exception.

We model the scenario shown in Fig. 7 by using the patterns presented in Sections 3.2 and 3.4:

1. Mr K is 76 years old with diabetes mellitus type 2 and hypertension. He comes to see his GP for complaining of flu-like

symptoms that he has been having for the past two weeks. The GP makes a diagnosis of ARF. As the GP does not have the required skills to treat the patient, he decides to **assign** the patient to a nephrologist to urgently stabilize the patient's condition. To assign Mr. K to Dr. John, who is a nephrologist, the GP initiates the plan *client_assignment_pattern*. As part of the activated plan the goals *achieve_serv_requested* and *achieve_contract_awarded* become active. The GP calls Dr. John to ask if he can stabilize the patient's condition.

2. Dr. John agrees to handle Mr. K's case and provides him urgent care for ARF. The GP fills in a form indicating that Dr. John is being assigned the treatment of Mr. K. Therefore the plan *provider_assignment_pattern* becomes active and the goal *achieve_collaboration_decided* is completed. This means that the GP is no longer responsible or accountable for Mr. K's treatment, and therefore the GP's goal *achieve_contract_awarded* is completed and his plan *client_assignment_pattern* is also completed.
3. For Dr. John, the goal *achieve_service_completed* becomes active. He chooses to first assess the patient condition (goal *achieve_patient_assessed*) by performing a clinical exam and some laboratory tests. From the assessment he concludes that the patient has to be treated urgently using hemodialysis. To achieve the goal *achieve_service_completed* a hemodialysis treatment thus needs to be given urgently to the patient.
4. Dr. John decides to **delegate(with supervision)** the hemodialysis treatment. The plan *client_delegation_pattern* therefore becomes active and the goal *achieve_contract_awarded* becomes in progress. Dr. John chooses nurse Ana as the APN to delegate the hemodialysis treatment.
5. Ana can start hemodialysis immediately so she accepts the delegation. Then the plan *provider_delegation_pattern* becomes active and the initial goal *achieve_collaboration_decided* is completed, and the new goal *achieve_service_provided* becomes in progress. For Dr. John his goal *achieve_contract_awarded* is completed and two new goals become in progress: *maintain_delegation_supervised* and *achieve_outcome_checked* (shown at the bottom of Fig. 7).
6. Ana's responsibility is to decrease the rate of creatinemia and normalize the kalemia of Mr. K. Therefore to achieve the goal *achieve_service_provided* she programs the hemodialyzer and she starts the hemodialysis treatment.
7. As it is an emergency situation, the nephrologist is also present and he monitors the dialysis that nurse Ana is performing, as indicated by the active goal *maintain_delegation_supervised*.
8. During the dialysis session, a problem occurs. The hemodialyzer gives an error message that triggers the suspending-obstacle *failure_hemodialyzer* (row 4, Table 5) which suspends the goal *achieve_service_provided*, stopping the dialysis.
9. The obstacle *failure_hemodialyzer* triggers the goal *achieve_failure_assessed*. Because the nurse was managing the hemodialyzer she is responsible for this exception. The nurse checks the hemodialyzer and finds air bubbles in the circuit, which can cause an air embolism. In addition, the nurse observes the onset of breathing difficulties in the patient, indicating symptoms of air embolism. These new symptoms trigger the hazard *symptoms_air_embolism* (row 5, Table 5).
10. In case of danger of air embolism, because the nurse is not competent to deal with this exception she needs to inform it to the nephrologist. So the hazard *symptoms_air_embolism* triggers the goal *achieve_exception_informed*.
11. Dr. John is informed about the exception (task *receive_exception_report*), which triggers his goal *achieve_exception_recovery_decided*. To achieve the goal he chooses the candidate plan *client_plan_react_to_exception* (Fig. 8). Dr. John decides to take charge of the patient stabiliza-

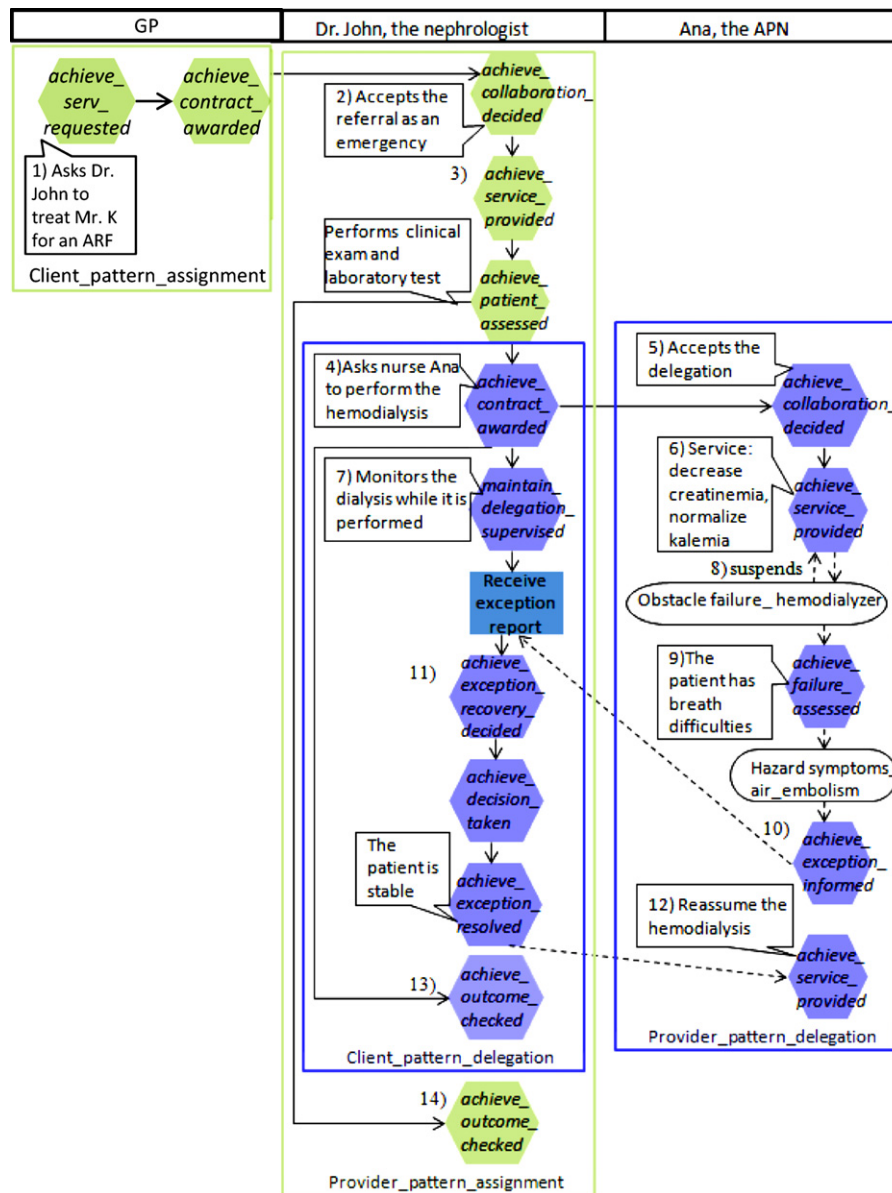


Fig. 7. Pattern-based specification of the scenario described in Section 4. Goals are represented as hexagons and tasks as rectangles.

tion (goal *achieve_decision_taken*), enacting the plan shown in Fig. 9 for achieving the goal *achieve_exception_resolved*.

12. After some hours the patient is stable, therefore the suspended goal *achieve_service_provided* becomes active again. The dialysis can be resumed by the nurse Ana in a normal way and is completed. Therefore the plan *provider_delegation_pattern* is completed.
13. Dr. John checks that the hemodialysis has been successfully completed (goal *achieve_outcome_checked* from plan *client_delegation_pattern* has been achieved), which completes the goal *maintain_delegation_supervised* and completes the plan *client_delegation_pattern*.
14. Dr. John has completed the assignment given by the GP, because he has stabilized Mr K's condition by an urgent hemodialysis (goal *achieve_outcome_checked* from plan *provider_assignment_pattern* gets completed). Therefore the plan *provider_assignment_pattern* gets completed.

4.2. Example pattern mapping

In Fig. 10 we provide a screenshot of a PROforma plan specified using the Tallis tool. Tallis is a suite of software tools built around PROforma to support authoring, publishing and enacting of clinical knowledge applications.

The plan specified is *provider_delegation_pattern*. Because the Tallis tool does not provide any library of terms that can be mapped to the terminology from Section 3.1, the full UML class diagram can be associated with the Tallis pattern implementation. If Tallis had provided a class diagram of the sort required by the pattern, a mapping should have to be provided to match the existing Tallis terms with those terms used in the conditions required for the pattern's enactment.

The Tallis plan *provider_delegation_pattern* is specified with *termination condition* equivalent to the pattern's *successCondition*. The enquiry *enquiry* and the task *check_precondition* are introduced

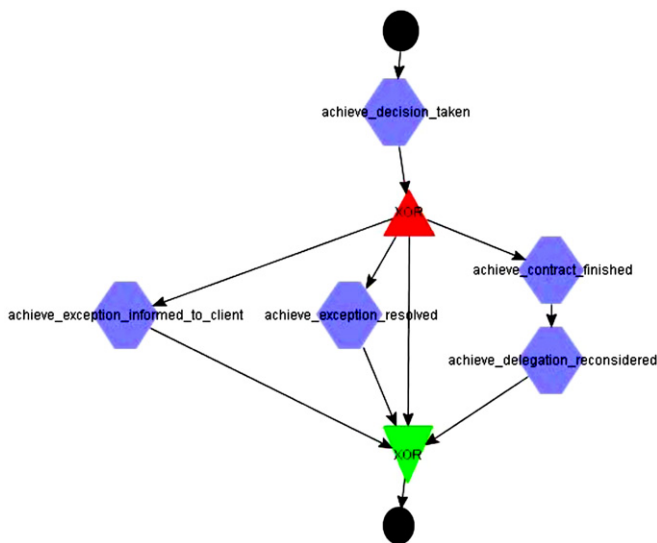


Fig. 8. *client_plan_react_to_exception*, candidate plan for goal *achieve_exception_recovery_decided*. The plan corresponds to first taking a decision (goal *achieve_decision_taken*) and depending on the outcome choosing between three different exclusive plans of action: (1) in case the client was delegated with the service and he is a provider of a super client then he can decide to inform the exception to the super client (goal *achieve_exception_informed_to_client*); (2) deal with the exception himself (goal *achieve_exception_resolved*); (3) not interfere with the way the provider is dealing with the exception. In this case *client_plan_to_exception_during_delegation* is Completed; or (4) finish the contract with the provider (goal *achieve_finish_contract*) and reconsider if he wants to re assign the task or goal or to perform it himself (goal *achieve_delegation_reconsidered*).

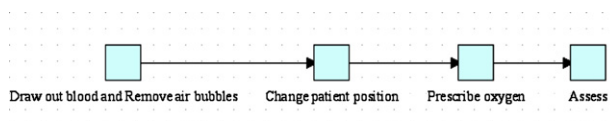


Fig. 9. Candidate plans for goal *achieve_exception_resolved*, where the exception is the *hazard symptoms_air_embolism*. The plan consists on the following sequence of tasks:

1. use Luer-lock syringes for blood draw from catheters and remove air bubbles,
2. change the patient position (Flat supine position Trendelenburg),
3. prescribe Oxygen with FiO2 100% and
4. assess the patient.

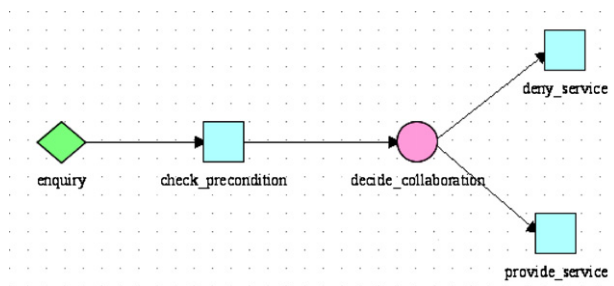


Fig. 10. Implementation in PROforma using the Tallis tool of the plan *provider_delegation_pattern*.

at the beginning of the Tallis plan *provider_delegation_patterns* to check the pattern's *precondition*. The *enquiry* allows accessing the data that is required to evaluate the *precondition* of the task *check_precondition*, such as: actor's competences, contracts signed by the actors, record of service requested, etc.

The pattern's goals *achieve_collaboration_decided* and *achieve_service_provided* are mapped into the Tallis decision *decide_collaboration* and task *provide_service*, respectively. The Tallis decision and task have as *precondition* and *postcondition*

expressions equivalent to the *precondition* and *successCondition* of the respective goals *achieve_collaboration_decided* and *achieve_service_provided*.

In the Tallis plan, the task *deny_service* is introduced to simulate the end of the plan when the provider does not agree to provide the service. The pattern's XOR scheduling constraint is simulated in Tallis by specifying two disjunctive *preconditions* for the tasks *deny_service* and *provide_service*.

With respect to the exception handling it is not fully supported by Tallis. For instance suspending obstacles cannot be simulated. For the other types of exceptions, discarding obstacles and hazards, we can define a Tallis plan whose *state trigger* attribute behaves as the exception's *condition*, activating the plan when the *condition* is satisfied. For instance we can simulate the hazard *symptoms_air_embolism* (Table 14) with the Tallis plan *Plan_hazard_symptoms_air_embolism*. The Tallis plan is specified as a decision with multiple candidate plans, one Tallis plan for each candidate plan that can be enacted to achieve the goal specified by the exception's attribute *triggeredGoal*. In the case of the hazard *symptoms_air_embolism* with *triggeredGoal=achieve_exception_informed*, if the goal has *n* candidate plans, then the plan *Plan_hazard_symptoms_air_embolism* includes a decision with *n* candidate plans, one for each candidate plan to achieve goal *achieve_exception_informed*.

Discarding obstacles can be simulated by adding to the Tallis plan that corresponds to the exception a task whose *postcondition* makes true the *abort condition* of the discarded plan. For instance when the discarding obstacle *unawarded_contract* (first row, Table 5) is triggered it discards the goal *achieve_contract_awarded*. Similarly in Tallis we can define the plan *Plan_contract_awarded* with *abort condition=(P=true)* for the corresponding goal, and the plan *Plan_discarding_obstacle_achieve_contract_awarded* with *postcondition=(P=true)* for the corresponding obstacle. Therefore when the *Plan_discarding_obstacle_achieve_contract_awarded* is completed it makes true the *abort condition* of the plan *Plan_contract_awarded*.

5. Conclusions

In Section 3.1 we proposed a UML class diagram to address problems related to team work in health care, including:

- (1) Lack of information about workers' competences. If the actor's *features are precisely specified* the function *IsCompetent* can check if an actor is competent to provide/assign/delegate a service and if an actor can play a role or participate in a group.
- (2) Service users do not necessarily receive the most appropriate care from the most appropriate practitioner. We provided the function *ObtainCompetentProviders* which can be used to check which members of staff can provide a service under specific constraints.

In Sections 3.2 and 3.4 we proposed *generic* patterns for the specifications of normal and abnormal scenarios for the assignment and delegation of tasks in health care teams. The patterns are based on the standard terminology for collaborative work, but they make explicit the responsibility and accountability of the service client and provider in abnormal scenarios. In Sections 3.3 and 3.5 we have shown that the patterns satisfy the properties that characterize service assignment and delegation.

With the Assignment and Delegation patterns we achieve a clear differentiation between delegation and assignment, addressing the problems of:

- (3) Lack of clarity to determine if the work is delegated or assigned. Using the patterns the team members can precisely determine the type of team work, if it is an assignment or delegation (with or without supervision).

- (4) Lack of precision on roles played by team members. Once an assignment and delegation contract is created between the team members the roles (client and provider), accountabilities and responsibilities both in normal and abnormal scenarios are fixed.
- (5) Incomplete or ambiguous specification of responsibilities and accountabilities. As we showed in Sections 3.3 and 3.5, the patterns provide a complete and unambiguous specification of responsibilities and accountabilities.

A prototype of the framework used for the pattern's specification [6] has been developed, using COGENT [22]. The prototype does not include the exception manager module. A simplified version of the goal-based pattern presented by Black et al. [29] and the patterns presented in this paper has been implemented and instantiated in the prototype. The simplification mainly consisted on not considering abnormal scenarios triggered by exceptions. In the near future we expect to introduce to the COGENT prototype framework the functionality required for detecting and recovering from exceptions.

Like us, others have previously defined patterns [6–10] to abstract solutions and lessons learned from the experience in specifying domain-specific scenarios.

In Russell et al. [39] the so-called workflow resource patterns are introduced to capture the various ways in which (human, non-human) resources are represented and utilized in workflows. The patterns presented here can be seen as combinations of some of their 43 basic patterns. For instance our notions of *roles* and *actors* allow implementation of the “role-based allocation pattern” that restricts at design time the roles of the resources that can perform a task, but also the “capability-based allocation” pattern that specifies that instances of task should be offered or allocated based on the resources capabilities.

In Castelfranchi et al. [14] and Huang et al. [17] agent-based approaches are proposed for modelling generic solutions to collaborative team work. In [14] two different ways of cooperation called delegation and adoption are considered. Their notions of delegation/adoption are quite generic and could potentially be refined to model the notions of assignment and delegation explained here. Like us they allow sub delegation and sub adoption of tasks and they provide agents with mechanisms to control that the delegated/adopted task has been successfully completed. Like us, the work of Huang et al. [17] addresses the problem of collaborative work in health care; we share with them the idea that agents can collaborate to achieve goals and that a decision support system is used to decide at run-time the best plan to achieve a goal. The approach from [17] to collaborative work does not differentiate between service assignment and delegation, neither have they presented any general strategy to differentiate accountability and responsibility depending on the type of collaborative work. In contrast we have focused on the problem of service assignment and delegation, defining two different generic strategies of accountability and responsibility in abnormal scenarios based on the type of collaboration. In addition, the pattern presented in [17] is based on the agent's communication, which depends on a fixed set of communication primitives; while our approach is more generic and it abstracts away implementation details (like communication primitives) focusing instead on the agent's intentions or goals. Only at run time do the agents (client and provider) decide if any communication should be used to achieve a goal and in that case they can select a communication language.

According to Mackey [11] and Royal College of Nursing [19], the problem of who is accountable for re-planning in case of abnormal events arising during the assignment or delegation of services in the medical field is crucial and can have legal consequences. In Castelfranchi et al. [14] this problem has been addressed by

providing a hierarchy of possible conflicts between client and provider during team work, but they have not specified any strategy to recover from those conflictive scenarios. In Huang et al. [17] the agents can collaborate in teams by sharing contracts and a basic catalogue of rules to detect exceptions and to enact recovery strategies of the sort: drop a contract, drop or start a task, etc. As in [17] our approach allows the specification of a catalogue of exceptions to detect and recover from arbitrary exceptions. But our catalogue of exceptions can be hierarchical, where some exceptions can be very generic and reusable for any type of assignment and delegation (like obstacle *unawarded_contract* defined in row 1 Table 5 or obstacle *unsatisfied_service_completion* introduced in row 2 Table 5), while other exceptions may be specific to the type of collaborative work (like the exceptions specific to hemodialysis: obstacle *failure_hemodialyzer* in row 4 Table 5, or hazard *symptoms_air_embolism* in row 5 Table 5). In our approach the recovery strategies triggered by the exceptions are given in terms of goals to be achieved, instead of concrete workflows to be enacted as it is the case for [17]) and most of the languages used for specifying computer-interpretable guidelines (see Peleg et al. comparative study [5]).

Our idea of defining policies (positive and negative authorizations) to enact tasks and goals and defining functions to check the actor's competences and the existence of policy conflicts is an abstraction of the existing general-purpose policy management frameworks KAoS [31] and Rei [32]. Neither Castelfranchi et al. [14] nor Huang et al. [17] have considered this issue.

Although our patterns are *abstract* specifications of team work we showed in Section 4 how they can be easily instantiated and executed in the framework used for their specification, or even be mapped into a concrete language like PROforma and specified and enacted using the Tallis [18] tool.

While our patterns are very concise and simple to understand their combination allows modelling of very complex scenarios such as the following:

1. *Open contract*: a client delegates or assigns a *goal* to a provider, and therefore gives the provider freedom to choose the best way to achieve the goal. The provider is considered autonomous. For example in Section 4 we described a scenario in which the advanced practice nurse is delegated the goal of choosing some hemodialysis treatment for decreasing the rate of creatinemia and normalizing the kalemia.
2. *Closed contract*: a client delegates/assigns a *task* to a provider, and therefore the provider is restricted to perform the task as described by the client's specification. No autonomy is given to the provider.
3. *Mixed contract*: A client delegates/assigns a *plan with goals and tasks*. The provider is semi-autonomous.
4. *Subcontracts*: A provider in one contract can become the client in a new contract involving part or all the service from the first contract. For instance a provider chooses for goal *achieve_service_provided* any of the following candidate plans: *client_assignment_pattern* or *client_delegation_pattern*. In the scenario described in Section 4.1, the GP assigns to a nephrologist the task of urgently stabilizing the patient's condition of *Acute Renal Failure*. The nephrologist assesses the patient and delegates (with supervision) the hemodialysis treatment to an advanced practice nurse. Another example is when the client of a delegation with supervision has been informed by the provider of an exception that the provider cannot manage. In this case the client can decide to delegate with supervision the responsibility of managing the exception while he keeps the accountability.
5. *Third party supervision*: An agent can delegate a service to a provider and delegate or assign the supervision of the task or goal to a third agent. The client of the delegation pattern can

choose to delegate to another agent the task of checking the outcome of the delegated service. The client can do this by selecting as *candidatePlan* for the goal *achieve_outcome_checked* the plan *client_delegation_pattern*.

6. *Propagating information related to exceptions that arise during chains of delegations*: Allowing propagation of exceptions from providers to clients in a chain of delegations until some client decides to deal with the exception rather than propagate it further.
7. *Different criteria to check service completion*: Examples of completion criteria that can be modeled using the function *completion* include checking that the task has been completed, or that the task has been completed following the client's preferences, or that the task has been literally completed (the provider did not do more than was asked for), etc.

In the specification of collaborative team interactions the representation of temporal constraints is very important. While our framework allows dealing with basic temporal constraints a future extension of this work will look at ways of extending our approach with a theoretically grounded and sound approach for representing and reasoning with complex temporal constraints. For this future extension we will take inspiration from domain-independent frameworks already available for specifying temporal constraints in medical environments, like RESUME [40] and Asbru [41].

While we have motivated the specification of the presented patterns on the solution of problems related to collaborative work in teams of health workers, the proposed patterns should be generic enough to describe collaborative work in heterogeneous teams in which humans and automated systems (i.e. decision support systems, IT applications) can collaborate to achieve common goals. These patterns could potentially be used in multi-agent based environments or service-oriented systems, where heterogeneous agents or peers adopt the role of clients and provider to collaborate on the enactment of common goals.

Acknowledgements

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

We also want to thank the reviewers for their thorough review and helpful comments.

References

- [1] Kohn LT, Corrigan JM, Donalson MS, editors. *To Err is Human: building a safer health system*, Committee on Quality of Health Care in America. Washington, DC: Institute of Medicine, National Academy Press; 1999.
- [2] Field MJ, Lohr KN, editors. *Guidelines for clinical practice: directions for a new program*. Washington, DC: Institute of Medicine, National Academy Press; 1990.
- [3] Overhage JM, Tierney WM, Zhou XH, McDonald CJ. A randomized trial of corollary orders to prevent errors of omission. *J Am Med Inform Assoc* 1997;4(5):364–75.
- [4] Shea S, DuMouchel W, Bahamonde L. A meta-analysis of 16 randomized controlled trials to evaluate computer-based clinical reminder systems for preventative care in the ambulatory setting. *J Am Med Inform Assoc* 1996;3(6):399–409.
- [5] Peleg M, Tu SW, Bury J, Ciccarese P, Fox J, Greenes RA, et al. Comparing computer-interpretable guideline models: a case-study approach. *J Am Med Inform Assoc* 2003;10(1):52–68.
- [6] Grando A, Peleg M, Glasspool D. A goal-oriented framework for specifying clinical guidelines and handling medical errors. *J Biomed Inform* 2010;43(2):287–99.
- [7] Friedland PE, Iwasaki Y. The concept and implementation of skeletal plans. *J Automat Reason* 1985;1:161–208.
- [8] Tu SW, Musen MA. A flexible approach to guideline modeling. In: Lorenzi NM, editor. *Proceedings AMIA symposium*. Philadelphia: Hanley & Belfus Inc.; 1999. p. 420–4.
- [9] Tu SW, Campbell J, Glasgow J, Nyman M, McClure R, McClay P, et al. The SAGE guideline model: achievements and overview. *J Am Med Inform Assoc* 2007;14:589–98.
- [10] Peleg M, Tu SW. Design patterns for clinical guidelines. *Artif Intell Med* 2009;47(1):1–24.
- [11] Mackey H. Assistant practitioners: issues of accountability, delegation and competence. *Int J Ther Rehabil* 2005;12(8):331–8.
- [12] Richardson G, Maynard A, Cullum N, Kinding D. Skill mix changes: substitution or service development? *Health Policy* 1998;45(2):119–32.
- [13] Grando MA, Peleg M, Glasspool D. Goal-based design pattern for delegation of work in health care teams. In: Safran C, Reti S, Marin HF, editors. *Proceedings of the 13th world congress on medical and health informatics*. Studies in Health Technology and Informatics, vol. 160. Netherlands: IOS Press; 2010. p. 299–303.
- [14] Castelfranchi C, Falcone R. Towards a theory of delegation for agent-based systems. *Robot Auton Syst* 1998;24:141–57.
- [15] Fisher M, Wooldridge M. Specifying and executing protocols for cooperative action. In: Deen SM, editor. *Proceedings of the second international working conference on cooperating knowledge based systems*. Berlin: Springer-Verlag; 1994. p. 295–306.
- [16] Dehoussé S, Liu L, Chi C, Faulkner S. Delegation models in service oriented systems. In: Di Nitto E, Hall RJ, Han J, Han Y, Polini A, Sandkuhl K, Zisman A, editors. *Second IEEE international symposium on service-oriented system engineering*. Washington, DC: IEEE Computer Society Press; 2006. p. 85–94.
- [17] Huang J, Jennings NR, Fox J. Agent-based approach to health care management. *Appl Artif Intell* 1995;9:401–20.
- [18] Tallis toolset. Available at: <http://www.acl.icnet.uk/lab/tallis/Documentation.CREDO.2002.008.htm> [accessed: 28 July 2010].
- [19] The Chartered Society of Physiotherapy, Royal College of Speech and Language Therapists, British Dietetic Association and Royal College of Nursing. (Eds.), *Supervision, accountability and delegation of activities to support workers, a guide for registered practitioners and support workers*, Intercollegiate Information Paper, United Kingdom; 2006. p. 1–27.
- [20] Shardlow S. Confidentiality, accountability and the boundaries of client-worker relationships. In: Hugman R, Smith D, editors. *Ethical issues in social work*. London: Routledge; 1995. p. 65–83.
- [21] Fox J, Johns N, Lyons C, Rahmzadeh A, Thomson R, Wilson P. PROforma: a general technology for clinical decision support systems. *Comput Methods Prog Biomed* 1997;54(1–2):59–67.
- [22] Cognitive Modeling Language (COGENT). Available at: <http://cogent.psyc.bbk.ac.uk/> [accessed: 28 July 2010].
- [23] Shahar Y, Miksch S, Johnson P. A Task-Specific Ontology for Design and Execution of time-oriented skeletal plans. In: Gaines B, Musen M, editors. *Proceedings of the tenth knowledge acquisition for knowledge-based systems workshop*. 1996. p. 9–14.
- [24] Musen MA, Tu SW, Das AK, Shahar Y. EON: a component-based approach to automation of protocol-directed therapy. *J Am Med Inform Assoc* 1996;3(6):367–88.
- [25] Tu SW, Kahn MG, Musen MA, Fagan LM, Ferguson JC. Episodic skeletal-plan refinement based on temporal data. *Commun ACM* 1989;32:1439–2155.
- [26] Musen MA, Carlson CW, Fagan LM, Deresinski SC, Shortliffe EH. T-HELPER: automated support for community-based clinical research. In: Frisse ME, editor. *Proceedings of the sixteenth annual symposium on computer applications in medical care*. New York: McGraw-Hill Inc.; 1992. p. 719–23.
- [27] Herbert SI, Gordon C, Jackson-Smale A, Renaud Salis JL. Protocols for clinical care. *Comput Methods Programs Biomed* 1995;48:21–6.
- [28] Shortliffe EH, Scott AC, Bischoff MB, Campbell AB, Van Melle W, Jacobs CD. ONCOC: an expert system for oncology protocol management. In: Hayes PJ, editor. *Proceedings of 7th international joint conference on artificial intelligence*. Los Altos: William Kaufmann; 1981. p. 878–81.
- [29] Black E, Glasspool D, Grando MA, Patkar V, Fox J. Goal-based decision for dynamic planning. In: Combi C, Shahar Y, Abu-Hanna A, editors. *Proceedings of 12th conference on artificial intelligence in medicine*. Berlin: Springer-Verlag; 2009. p. 96–100 [LNCS 5651].
- [30] Rumbaugh J, Jacobson I, Booch G. *The unified modeling language reference manual*. Massachusetts: Second Edition Addison-Wesley Object Technology Series, Addison-Wesley; 2004.
- [31] Bradshaw J, Uszok A, Jeffers R, Suri N, Hayes P, Burstein M, et al. Representation and reasoning for DAML-based policy and domain services in KAOs and Nomads. In: Rosenschein JS, Wooldridge M, Sandholm T, Yokko M, editors. *Proc. of the 2nd int. joint conf. on autonomous agents and multiagent systems*. Melbourne: ACM Press; 2003. p. 835–42.
- [32] Kagal L, Rei, a policy language for the Me-Centric Project, Technical report, HP Laboratories, Palo Alto, CA, USA; 2002 [HPL-2002-270].
- [33] Burke E, De Causmaecker P, Berghé GV, Landeghem HV. The state of the art of nurse rostering. *J Scheduling* 2004;7:441–99.
- [34] Dijkstra E, Edsger W. *A discipline of programming*. New Jersey: Prentice-Hall Series in Automatic Computation; 1976.
- [35] Owicki S, Gries D. An axiomatic proof technique for parallel programs. *Acta Inform* 1976;6:319–40.
- [37] Headley CM, Wall B. Advanced practice nurses: roles in the hemodialysis unit. *Nephrol Nurs J* 2000;27(2):177–84.
- [38] Easom A, Allbritton G. Advanced practice nurses in nephrology. *Adv Ren Replace Ther* 2000;7(3):247–60.

- [39] Russell N, ter Hofstede AHM, Edmond D. Workflow resource patterns: identification, representation and tool support. In: Pastor O, Falcao e Cunha J, editors. Proc. 17th conf. on advanced information systems engineering. Berlin: Springer-Verlag; 2004. p. 216–32 [LNCS 3520].
- [40] Shahar Y, Musen A. RESUME: a temporal-abstraction system for patient monitoring. *Comput Biomed Res* 1993;26(3):255–73.
- [41] Miksch S, Shahar Y, Horn W, Popow C, Paky F, Johnson PD. Time-oriented skeletal plans: support to design and execution. In: Steel S, Alami R, editors. *Lecture Notes in Artificial Intelligence (LNAI), recent advances in AI*

Planning, fourth European conference on planning ECP'97 (EWSP'97). 1997. p. 301–13.

Further reading

- [36] Grando MA, Mitrana V. A possible connection between two theories: grammar systems and concurrent programming. *Fundam Inform* 2007;76(3):325–36 [Special issue on developments in grammar systems].