Coping with Exceptions in Agent-Based Workflow Enactments*

Joey Sik-Chun Lam, Frank Guerin, Wamberto Vasconcelos, and Timothy J. Norman
{j.lam, f.guerin, w.w.vasconcelos, t.j.norman}@abdn.ac.uk

Department of Computing Science
University of Aberdeen, Aberdeen, U.K.
AB24 3UE

Abstract. A workflow involves the coordinated execution of multiple operations and can be used to capture business processes. Typical workflow management systems are centralised and rigid; they cannot cope with the unexpected flexibly. Multi-agent systems offer the possibility of enacting workflows in a distributed manner, by agents which are intelligent and autonomous. This should bring flexibility and robustness to the process. When unexpected exceptions occur during the enactment of a workflow we would like agents to be able to cope with them intelligently. Agents should be able to autonomously find some alternative sequence of steps which can achieve the tasks of the original workflow as well as possible. This requires that agents have some understanding of the operations of the workflow and possible alternatives. To facilitate this we propose to represent knowledge about agents' capabilities and relationships in an ontology, and to endow agents with the ability to reason about this semantic knowledge. Alternative ways of achieving workflow tasks may well require an adjustment of the original agent organisation. To this end we propose a flexible agent organisation where agents' roles, powers and normative relationships can be changed during workflow enactment if necessary. We show an example of how this combination allows certain workflow exceptions to be handled.

1 Introduction

The Workflow Management Coalition (WfMC) defines a workflow as "the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules" [36]. Workflows can be formalised and expressed in a machine readable format, and this makes it possible for them to be employed in service-oriented computing scenarios. In such scenarios we may be dealing with open heterogeneous computing systems, where errors and exceptions are likely to occur. We would like the computing systems to cope with these exceptions. Ideally we would like to be able to deal with the unexpected; while we could write specific exception handling routines to deal with some common exceptions which we expect to arise, it will be difficult to anticipate all possible exceptions. Hence it would seem that we need some type of intelligence to deal with the unexpected. Typical workflow management systems (e.g., Taverna [25], Kepler [23]) are centralised and rigid; they cannot cope with the unexpected flexibly. Moreover, they have not been designed for dynamic environments requiring adaptive

* This work is funded by the European Community (FP7 project ALIVE IST-215890).

responses [7]. To overcome this we argue that it will be necessary to use agents to control the enactment of a workflow in a distributed manner; agents can be endowed with sufficient intelligence to allow them to manage exceptions autonomously. This should bring flexibility and robustness to the process of enacting workflows.

Different types of exceptions may arise during the enactment of an agent-based workflow. We can identify different levels of adaptivity, and exceptions can occur at any level. The following are the levels of adaptivity [1]:

- Organisation level: exceptions due to changes in the environment may mean that the current organisational structure makes it impossible for a workflow to progress. The organisation must be changed to adapt to the current situation.
- Coordination level: there are exceptions due to changes in the environment, or the agents and their organisational position. For example, some roles may be empty so that the workflow cannot progress. The workflow itself must be altered, possibly to find alternate pathways on which the tasks may be completed.
- Service level: this is the lowest level at which exceptions occur, and the simplest to deal with. A web service is unavailable and an alternative must be found. This may be possible without changing the existing workflows.

It is often the case that exceptions at lower levels can be dealt with by the next higher level; this is indeed one of the main advantages of using an agent based approach rather than a typical workflow execution engine. For example, agents can be used to manage the invocation of Web services, and then they can manage the Web services in an intelligent way [5]; such techniques can also be used to cope with exceptions intelligently. A service-level exception could be one in which a required Web service has gone offline; in this case an agent can use semantic matching [26] or service composition techniques [34] to search for a replacement. For example, if the equipment supplier is not available to give a quote for the required robotics equipment, agents can search for a supplier whose services are described semantically, the replacement supplier can be either an exact match or more general than the one currently specified. Thus coping with service-level exceptions can be done to some extent with existing techniques [37].

Higher-level exceptions are more problematic. For example if the powers or prohibitions in the agent system do not allow the agents to complete the workflow and this leaves the workflow deadlocked at a certain stage. Methods for coping with such exceptions have not been addressed in the literature so far, to our knowledge. If we are to cope with these types of exceptions it would seem that we need organisational flexibility, or the ability to change the social relationships among roles as necessary. Again, higher levels can deal with exceptions at lower levels To facilitate this we make use of an institutional framework with certain speech acts which can modify the roles, powers or obligations of agents in the organisation. For example, suppose that there is a single agent a in the organisation who is empowered to authorise equipment purchases, and that agent is currently unavailable. If an agent b urgently needs an equipment purchase to be authorised, a request can be made to a manager to appoint a suitable stand-in for a, with the appropriate power. This will be further explained in Section 4.

For agents to cope with the unexpected autonomously they must have some understanding of the operations of the workflow and possible alternatives. This means that the tasks specified in the workflow cannot be meaningless labels, but must be associated with some semantic information. To satisfy this requirement we provide an OWL ontology [4] representing the background knowledge for the organisation. This allows

agents to reason about the capabilities of agents in the organisation and find alternative ways to deal with workflow tasks when exceptions arise.

In Section 2 we give an overview of our approach. In Section 3 we describe how we model the agent institution, focusing on normative notions. In Section 4 we describe our workflow specification with an example "Equipment Purchase" workflow. Section 5 illustrates an ontology which captures essential aspects of an organisation with roles and tasks. Section 6 describes how agents deal with exceptions by using the knowledge in the ontology, and also respecting the norms in the institution, and modifying norms if necessary. Section 7 looks at related work and Section 8 concludes.

2 Proposed Approach

We focus on adaptation at the organisational and coordination levels. To make our organisation flexible we adopt an institutional framework which allows roles, powers and normative relations to change dynamically, under agents' control. To endow agents with the ability to reason intelligently about how to cope with exceptions, we represent background knowledge about the organisation, and the knowledge and capabilities of its constituent agents, in an OWL ontology. This knowledge enables the agents to recommend appropriate changes when exceptions arise. Our system has two types of organisational knowledge: knowledge about the powers and the norms governing agents are represented as logical rules, as part of the "institutional facts" of the institution; knowledge about the capabilities of agents and the hierarchy of the organisation, and constraints among roles, are represented in an OWL ontology. This separation is appropriate because the former knowledge (norms, powers and roles) may include norms and powers which apply under certain conditions, where conditional rules could not be represented in OWL. This knowledge is also dynamic, being frequently changed by the agents. The latter knowledge (agent capabilities, hierarchy, constraints) is static, and easily represented in a description logic based ontology (i.e., OWL DL).

We use examples from a University institution. In our examples we use a multi-agent system to model the activities in the institution; thus it is not the case that agents are supporting humans in the institution; it is a simulation where the agents are playing the roles of the humans. This modelling is an exercise to test if our framework is able to cope with scenarios arising in human institutions. In future work we intend to use the framework to simulate exception handling in disaster management scenarios.

3 Modelling the Institution

To model our institution we build on previous work [12] where we described an agent communication framework which allows rules to be defined to describe how events (such as messages being sent) lead to modifications in the institutional facts. This is in line with the "social perspective" [32] on communication; i.e., the institution is not concerned with private mental states of agents. The rules are first-order logic clauses implemented in Prolog. We describe it here with a Prolog-style notation. In fact our institutional framework is quite similar to [27] which uses event calculus to represent the rules, and implements them also using Prolog. We simply use Prolog directly. In all examples here ee follow the Prolog convention where a string starting with an uppercase case letter is a variable.

We describe the state of an institution by a pair $F = \langle R, A \rangle$ where R describes the current *rules* in force in the institution and A describes the *state of affairs*. Collectively

these are called the institutional facts F. *Rules* describe how the speech acts of agents, or other events, lead to changes in the institutional facts. An example of an institutional fact of the *state of affairs* type is having the title "doctor"; examples of institutional rules are the rules of a University describing how the title can be awarded and by whom.

Given an institution described by facts F_0 at some instant, and a subsequent sequence of events $e_1, e_2, e_3 \ldots$, we can use the rules to obtain the description of the institutional facts after each event, obtaining a sequence of facts descriptions: $F_1, F_2, F_3 \ldots$. In our examples events are either speech acts (i.e., messages being sent) or timer events. Note that the above description allows for events which change the rules, although this will not be used in our examples. Thus all events lead to modifications of A; A is simply a list of predicates (facts which hold) or clauses (where they hold conditionally, as in the powers below, for example); events add and remove elements from this list. For convenience, A has been divided into a number of sub-lists for: *roles*, *world_facts*, *powers*, *obligations*, *prohibitions*, *pending_acts*. We will explain the types of facts which populate these sub-lists of A after we introduce our running example below. However, we will firstly explain how speech acts fit within the framework of the institution.

3.1 Speech Acts

The meaning of speech acts is defined by rules in R. These rules define how a set of facts in A is modified by a speech act event. Thus a rule is a function taking as input facts F_0 and speech ace e, and giving as output the new facts F_1. We will briefly describe some illustrative examples because there is insufficient space to give the rules for our speech acts in full. The speech acts *assign_role* and *assign_power* define a particularly simple manipulation of A: the new role or power (as specified in the content of the act) is simply added to the appropriate list. The *allocate_task* speech act causes a new obligation to be added for the agent in question. An *inform* speech act simply adds the content of the act to the *world_facts* within the list A; thus an agent can assert some facts and make them publicly available. A *request* speech act adds an obligation for the receiver to reply.

We shall illustrate the notions we introduce below with examples from a University institution (see Figure 4). Later we will define a workflow, and illustrate exception handling, using this same institution. The rules R include speech act rules which can change agents' roles. The state of affairs A includes a record of the roles of the institution, and the agents occupying each role. If agent Ag is currently occupying the role of senior lecturer, and the Head of College performs the speech acts "$assign\_role, [Ag, professor]$" and "$remove\_role, [Ag, senior\_lecturer]$", then Ag will be moved into the role of professor. In these speech acts we are just stating the performative and content (a sender and receiver are also needed to complete the speech act). Some speech acts have more elaborate rules; the act *assign_temp_role* (*college_staff*, [Ag, hod, $Duration$]) triggers two acts to be executed: the act *assign_role* (*college_staff*, [Ag, hod]) is executed immediately, and the act *remove_role* (*college_staff*, [Ag, hod]) is placed on the "*pending_acts*" list, for execution at time $Duration + Current\_time$. This "*pending_acts*" list holds actions that need to be taken at a future time. When the time is reached the action is executed, much like a speech act, however the *sender* field is blank, meaning that the act is simply executed without any checks for power or prohibition.

3.2 Normative Notions

Powers, prohibitions and obligations can be assigned to roles or to agents themselves. An agent inherits powers and norms from the roles it takes on. We will not list all the powers, prohibitions and obligations of our example scenario here, but we will give illustrative examples. Power is necessary for an agent to be able to use a speech act to change the institutional facts; if the agent does not have power, then uttering the act will not have the desired effect. There is a power (i.e. written as a fact in A) for each speech act that a role can effectively perform. The following is an example of some of the more interesting powers in our scenario:

(1) *power*(hod, *allocate_task* (Ag,[$Task$,$Time\_limit$])) if
role (Ag,cs_dept_staff)
(2) *power* (hod, *assign_temp_role* (college_staff,[Ag,hod,$Duration$])) if
role (Ag,cs_dept_staff) and *role* (Ag,professor) and $Duration \leq$21:00:00
(3) *power* (hod, *suspend_role* (college_staff,[Ag,$Role$,$Duration$])) if
role (Ag,cs_dept_staff)
(4) *power* (hoc, *assign_role* (college_staff,[Ag,hod])) if
role (Ag,cs_dept_staff) and *role* (Ag,professor)
(5) *power* (hoc, *assign_power* (college_staff,[Ag,$Power$])) if
role (Ag,college_staff)
(6) *power* (hod, *authorise_purchase* (secretary,Item))

In these powers the speech acts are written as "*performative* (*receiver*,[*content*])", the sender is added when the concrete act is performed. Note that when roles or powers are changed, the speech act is to be sent to the entire institution (a college in this case), so that all staff know of the new assignment. Line (1) means that the Head of Department (HoD) can allocate a task to anyone who takes on the role cs_dept_staff. Line (2) means that the HoD can temporarily assign the HoD role to any other professor in the department; the third parameter within the *assign_temp_role* is for the duration after which the temporary assignment will expire – this cannot exceed 21 days. Line (3) allows the HoD to temporarily suspend some agent's membership of a role; it lasts for a time defined by the third parameter. These two powers (2 and 3) are to be invoked when the HoD goes on vacation; the HoD agent will temporarily suspend its own occupancy of the HoD role, and appoint a replacement. Line (4) allows the head of college (HoC) to permanently assign the HoD role to any professor in the CS department. Line (5) allows the HoC to assign a power to an individual agent, or a role.

Obligations are a type of norm. Obligations are always defined with a time limit before which they must be carried out. Some example obligations are

(7) *obliged* (secretary,
complete_task ("upgrade webserver"),"`05-June-12:00`", 103)
(8) *obliged* (technician,
web_service (T, supplier_service, request, [quote, Equipment], Result),
"`09-Sept-18:00`", 103)

After the agent completes a task this is reported as completed in the *world_facts* in A; this means that compliance with the obligation can be checked by looking at the facts in A. The final parameter above (103) is the "sanction code" which applies if the

obligation is broken. Following [27] we associate a 3-figure "sanction code" with each norm violation (similar to the error codes used in the Internet protocol HTTP). The sanction codes gathered by each agent as it commits offences are merely recorded in a list. The use of codes is just a convenient way to record sanctions without yet dealing with them; we would require a separate component to impose some form of punishment.

Obligations do not usually have a condition (as some powers had). If we wish to model the situation where an agent is obliged to reply if it receives a *request*, then we must ensure that the performance of the request creates an obligation to reply; i.e. we make the *request* speech act add an obligation. The condition of any norm cannot be that a speech act is sent because conditions only check facts in A, not events. Thus obligations tend to have a more temporary existence than powers; they are added until they are fulfilled or expire. Obligations cannot have a negative content; i.e., we cannot state that an agent is obliged not to do something. To achieve this effect we use prohibitions. The following is a sample prohibition:

(9) *prohibited* (hoc, *assign_power* (college_staff,[hoc,$Power$]),103)

Again, the final parameter is a sanction code. Lines (5) and (8) model the situation where the head of college is prohibited from assigning new powers to him/herself but is nevertheless empowered to do so (i.e., if the prohibition is violated the assignment of power is still effective).

3.3 Updating the Institutional Facts

We are now ready to present the algorithm which is used to update the institutional facts when a speech act event happens (shown in Figure 1).

algorithm UPDATE-INSTITUTIONAL-FACTS

1. Input: a speech act with *Sender, Receiver, Performative, Content*
2. Check if *Sender* (or one of the roles he occupies) is empowered to send this speech act: If not, discard the act and exit this algorithm.
3. Check if there is a prohibition for *Sender* (or one of the roles he occupies) sending this speech act: If not, go to the next step; If so, apply the specified sanction.
4. Check if there is an obligation which requires that *Sender* (or one of the roles he occupies) send this speech act. If so remove the obligation from the state of affairs, A.
5. Process the act as normal (i.e., follow the rules specified for the act).
6. If the speech act is part of a workflow in progress, then check the workflow specification at the place which control now passes to, and add obligations for the receiving agent to execute each task required at that place.

Fig. 1: Algorithm to Handle Normative Relations when processing a speech act

It is in this algorithm that roles are consulted to retrieve the names of the agents occupying the roles; e.g., when checking if an agent who has just sent a message is obliged, the algorithm will consult the facts to see what roles the sending agent occupies. Processing speech act events is part of the procedure used to maintain the institutional facts. The main loop of the institution program is executed repeatedly, and invokes the above update algorithm as well as doing some housekeeping:

– For each obligation check if it has timed out. If so, apply the sanction to the agent (or all agents occupying the obliged role) and remove the obligation from A.
– For each pending_act check if it is due. If so, execute it and remove it from A.
– If there have been any events, then UPDATE-INSTITUTIONAL-FACTS

This completes the description of our institutional infrastructure. Note that our model of an institution is minimal, possessing only those essential features required to illustrate our approach. There are other more complete and sophisticated proposals for representing societies of software agents which could have been used instead. Some of these proposals are, for example, *Open Norm-Governed Computational Systems* [2], *electronic institutions* [11], *virtual institutions* [8] and organisations for agents [9] – such proposals could have also been used in this paper instead, however it would require more space to introduce them, and for this reason we have opted for our own minimal approach which allows us to devote more attention to the handling of exceptions. Some of the features of our approach have very clear counterparts in those proposals. For instance, in our algorithm to update institutional facts (Figure 1), the "censoring" of unauthorised utterances which takes place in step 2 corresponds to the *governor agents* of electronic institutions [11]; such agents intermediate all communications between external (foreign) agents and the institution/society. Governor agents check if the messages that the external agents want to send are indeed pertinent to the current state of the interactions; if this is the case, the illocution is forwarded to the appropriate receiver, otherwise the message is discarded.

4 Workflows

We extend the basic framework above with the definition of workflows, allowing us to define structured interaction patterns for agents. A workflow can be enacted by a workflow engine [18] or it can be controlled by individual agents. In our case we are relying on the intelligence of agents to take appropriate actions if the workflow enactment encounters an exception which prevents it from progressing. For this reason we will have the workflow executing in a distributed fashion, controlled at each stage by the agent responsible for that stage.

4.1 Workflow Specification Language

We have developed a simple workflow language. The workflow language assumes that a workflow has a finite number of numbered *places*, with transitions between them (Figure 2 shows a workflow, with places depicted as relationships). A workflow which is currently in progress may occupy one or more places. If the workflow has no branches it will only occupy one place at any time, but if it branches several places may be occupied simultaneously. Figure 3 gives a concrete example of a workflow specification, using a Prolog style notation. A workflow specification has a *place* predicate for each numbered place. The first parameter of a place predicate specifies the number identifying this place in the workflow; this is followed by the role which is responsible for executing the statements in this place, with an identifier in parenthesis for the agent who is taking up that role (e.g., "secretary(D)"). There then follows a sequence of statements enclosed by square brackets; these are to be executed in order by the agent taking up the role. Statements may be *variable assignments*, or *actions*, or *if…then …else* constructs. Actions include any action the agent can take such as performing a speech act, querying a Web service, etc.; some actions (e.g., calling a Web service) may return a result.

Variable assignments may include (as their right hand side) actions which return results. The final parameter of a place predicate is a deadline, indicating the maximum amount of time which is allowed to pass between the workflow reaching this place, and the control passing to the next workflow place.

Speech acts within our framework typically have four parameters: sender, receiver, performative, content; however, when a speech act is sent as part of a workflow we add information so that the recipient knows which workflow is being executed and what place it is at. Thus every message which passes control to another agent includes the number of the next place to be executed. This ensures that an agent receiving a message can look up the workflow specification and find what is to be done for this workflow place. It also serves to disambiguate between potentially confusing places: it is possible that a workflow might have two different places where the message being sent to the next agent is the same, thus an agent receiving the message would be unsure about what point had been reached in the workflow. In our example workflow shown in Figure 3 it can be seen that every speech act sent includes, as the final parameter, the name of the workflow ("ep"= equipment purchase) and the state that has been reached within it (e.g., "[ep,2]"). Speech acts are written in the form *speechAct(sender, receiver, performative, content, workflow)*.

For example in the workflow below (Figure 3) `2:technician(T)` means that this is workflow place 2 and is to be carried out by some agent occupying the role of technician, and that `T` is the variable to be used to hold the name of the actual technician who is carrying out this workflow place. For example, when the secretary receives the message from the technician, she assigns the variable `T` to be the name of the technician, and can subsequently send messages to `T`. On the other hand, if the secretary performs a speech act with `technician` as the receiver, then it can be sent to any agent occupying the role, and not necessarily the same technician identified as `T` previously.

4.2 Example: The Equipment Purchase Workflow

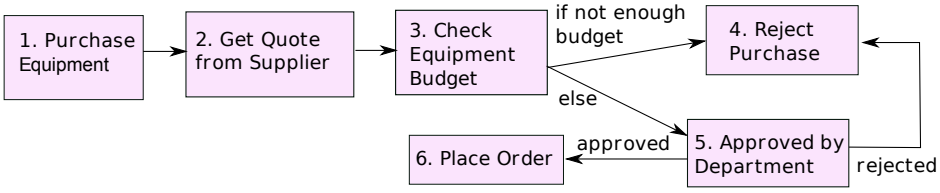

Fig. 2: Diagram of "Equipment Purchase" Workflow

We consider a simple workflow example – a purchasing equipment workflow in a university – to give an idea of how agents do reasoning to deal with unexpected circumstances during the enactment of workflows. Figure 2 graphically depicts the example. Firstly a research staff issues a request of purchasing a Robotics equipment (1), this request goes to the technician. The technician gets a quote from a supplier via a web service (2), and then passes the request to the department secretary. The secretary checks the equipment budget (3). If the budget is not enough for purchasing, then the purchase request is rejected and the workflow terminates (4). Otherwise, the purchase

request is then passed to the HoD for approval (5). The HoD is responsible for deciding to authorise this purchase (6) or reject it (4).

Apart from speech acts, there are three other types of actions that can be required at a place of the workflow: *web_service*, *local_service*, *query_expertise*. Only *speech_act* changes the institutional facts, the other acts give a result which only changes the agent's internal mental state. The *web_service* simply invokes an external web service, while the *local_service* action queries a local computer system, for example the department's finance system. The *query_expertise* action is used when an agent needs to query his own internal knowledge base; in the example below the HoD must execute this action, to query his own internal expertise in robotics equipment and make a decision about whether the robotics equipment proposed is in fact useful for the experiment proposed. This set of actions is known to all agents, although not all agents can carry out all actions; certain actions require certain expertise. For example, to execute *query_expertise* actions to answer queries about a particular topic will require that the agent has expert knowledge in its own knowledge base.

```
place(1,research_staff(R),[
 speech_act(R, technician, request, [purchase, Equipment, Experiment], [ep,2])
                          ]).

place(2,technician(T),[
 web_service(T, supplier_service, request, [quote, Equipment], Cost),
 speech_act(T, secretary, request, [Equipment, Cost, Experiment], [ep,3]),
                       ],24:00:00).

place(3,secretary(D),[
 local_service(D, dept_data, query, [equipment_budget], Budget),
 (if C > Budget
  then
    speech_act(D, T, inform, [reject, Equipment], [ep,4])),
 (if C <= Budget
  then
    speech_act(D, hod, request, [approve, Equipment, Experiment], [ep,5]))
                     ],5:00:00).

place(4,secretary(D),[
 speech_act(D, T, inform, [reject, Equipment])
                     ],1:00:00).

place(5,hod(H),[
 query_expertise(H, Equipment, is_appropriate_equipment(Equipment, Experiment), Useful),
 (if Useful
  then
    speech_act(H, D, authorise_purchase, [Equipment], [ep,6])
  else
    speech_act(H, D, reject_purchase, [Equipment], [ep,4]))
                ],48:00:00).

place(6,secretary(D),[
 web_service(D, supplier_service, request, [place_order, Equipment], _)
                     ],2:00:00).
```

Fig. 3: Specification of "Equipment Purchase" Workflow

The workflow is a coordination device for agents, in the same way as agent interaction protocols; the powers and prohibitions of agents are not overridden by the workflow – agents are still bound by them. This means that the workflow designer must take care to design workflows where all the tasks which the various roles carry out must not vi-

olate the existing norms which those roles are bound by. The designer must also make sure that the agents executing the workflow have the appropriate powers to perform their respective parts; this may sometimes require that the organisation be adjusted to add new powers to certain roles so that they can execute the tasks which the designer will allocate to them. In addition to these normative relations, during the enactment of a workflow, Step 6 of Algorithm 1 ensures that the next agent in the workflow must carry out the tasks at that place in the workflow. According to Step 6 of Algorithm 1, when a speech act includes the workflow parameter, then the algorithm inspects the workflow and adds obligations for the receiving agent to perform its place of the workflow. These obligations all include a time limit which is obtained by adding the final deadline parameter from the workflow place to the current time.

So far agents have not been given any knowledge about each others' capabilities, or about the capabilities required to perform tasks in the workflow. In order for agents to make intelligent decisions about what action to take when a workflow breaks down, agents will need to know about tasks and capabilities, so that tasks may be reassigned to others; without this knowledge they can only blindly follow a workflow. We encode the required knowledge in an ontology and endow agents with the ability to reason with the knowledge in this ontology, in order to make intelligent decisions when workflows break down.

5 An OWL Ontology

An ontology [35] formally captures a shared understanding of certain aspects of a domain: it provides a common vocabulary, including important concepts, properties and their definitions, and constraints regarding the intended meaning of the vocabulary, sometimes referred to as background assumptions.

More formally, An ontology $\mathcal{O}$ consists of a set of *terminology* axioms $\mathcal{T}$ (TBox) and assertional axioms $\mathcal{A}$ (ABox), that is, $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$. An axiom in $\mathcal{T}$ is either of the form $C \sqsubseteq D$ or $C \doteq D$, where C and D are arbitrary concepts; an axiom in $\mathcal{A}$ is either of the form $C(a)$ (where C is a concept and a is an individual name; a belongs to C), or of the form $R(a, b)$ (where a, b are individual names and R is a role/property name; b is a filler of the property R for a). The OWL-DL [16] ontology language is a variant of $\mathcal{SHOIN}(\mathbf{D})$ [17] Description Logic, which provides constructs for full negation, disjunction, a restricted form of existential quantification, and reasoning with concrete datatypes. OWL DL benefits from many years of DL research, the benefits include well defined semantics, well-studied reasoning algorithms, highly optimised systems, and well understood formal properties (such as complexity and decidability) [3].

The organisation in the university is represented by the OWL DL ontology, University.owl[1] (part of which is shown in Figure 4). The ontology models persons, the role hierarchy, constraints (such as mutually exclusive roles, cardinality, prerequisite roles) [31], and the range and domain of properties (which are represented by the arrow and black dot in the diagram). A person can take more than one role; a role can have many persons. An example of mutually exclusive roles is that the head of college cannot be the head of department simultaneously (see axiom 1 below); a course organiser supervising a student's project cannot mark that student's project (see axiom 2 below). Maximum and minimum cardinality constraints are also used. For example, only one person can fill the role of the head of department; a student has to take at least one

[1] http://www.csd.abdn.ac.uk/∼jlam/University.owl

course (see axiom 3 below). The concept of prerequisite roles means a person can be assigned to role $r1$ only if the person already is assigned to role $r2$. The Role hierarchy is also modeled in the ontology to reflect authority. More powerful roles are shown toward the top of the hierarchy and less powerful roles toward the bottom. This role hierarchy is consulted by the agents when an exception occurs in a workflow and the agent encountering the problem needs to report to a higher authority. The agent with higher authority may appoint agents to different roles, or change powers to overcome the problem; this is why it is important that the agents have knowledge of the relevant constraints on such appointments.

The following shows some of the axioms of the ontology and SWRL rules which are pertinent to our example exception, and its resolution.

1. HoD $\sqsubseteq \neg$ HoC
2. supervises(?staff,?stu) $\wedge$ doesProject(?stu,?proj) $\wedge$ Project(?proj) $\wedge$ marksProject(?staff,?proj) $\rightarrow$ owl:Nothing(?staff)
3. Student $\sqsubseteq \geq 1$ takesCourse
4. teaches(?staff,?crs) $\wedge$ Person(?staff) $\wedge$ Course(?crs) $\rightarrow$ hasExpertise(?staff,?crs)
5. hasRole(dave, ITProfessor) // dave is an IT professor
6. teaches(dave, Robotics) // dave teaches Robotics
7. $\top \sqsubseteq \forall$ teaches.Course // range(teaches) = Course
8. query_expertise $\sqsubseteq \exists$ doneBy.(Person $\sqcap \exists$ hasExpertise.Expertise)
9. Equipment(?q) $\wedge$ Expertiment(?x) $\wedge$ isNeededBy(?q,?x) $\rightarrow$ is_appropriate_equipment(?q,?x)
10. supplier_service $\sqsubseteq \exists$ supplies.(Equipment $\sqcap \exists$ hasPrice.Cost) $\sqcap \exists$ hasRating.xsd:int
11. supplier_service(?s) $\wedge$ hasRating(?s, ?r) $\wedge$ swrlb:lessThan(?r,2) $\wedge$ PlaceOrder(?act) $\wedge$ ordersFrom(?act, ?s) $\rightarrow$ owl:Nothing(?act)

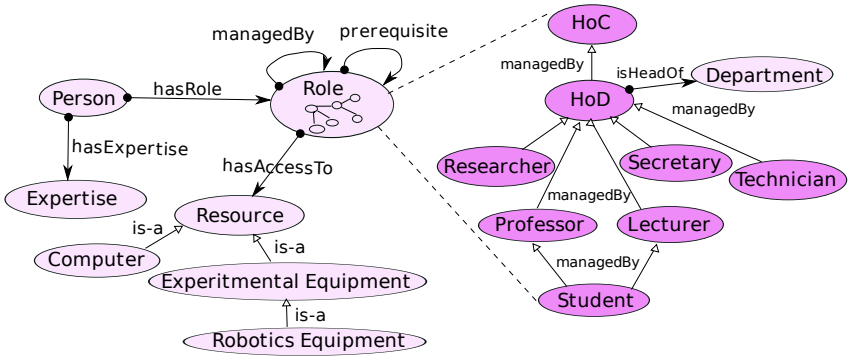

Fig. 4: Parts of the University.owl ontology

6 Exception in Equipment Purchase Workflow

Exceptions can occur in workflows easily, such as message delivery failure, resource unavailability, constraint violation, deadline expiry, etc. There are a range of possible ways in which an exception may be dealt with; it is usual to specify what should be done if exceptions arise and what recovery action will be taken to resolve the effects of exceptions at design time. To deal with exceptions flexibly, the approach chosen for

handling exceptions needs to be as generic as possible to ensure that it has the broadest applicability [30].

We now describe our generic approach to handling exceptions. Common exceptions can be broadly categorised under the following four headings: (1) Failure of message delivery due to agent unavailability: this is reported to an upper management level; an appropriate agent should replace the unavailable agent. (2) No resource can be found which meets the specified allocation criteria for the task during the execution of workflow: the agent tries to search for alternative resources for replacement. (3) Agents violate their norms such as performing some prohibited actions, thus preventing the workflow from progressing: a sanction is applied or remedy should be taken depending on specific situations. (4) A task is not completed before its deadline: the ways of dealing with it could include: complete the task as soon as possible; abort the whole workflow; restart the workflow; reassign agents or resources to this task, etc. In this paper we will focus on the first two types of exception handling, because the latter two types are highly dependent on specific domains.

The algorithm for handling exceptions is sketched in Figure 5; we describe it informally now. When an agent attempts to send a message to an unavailable agent Ag_2 whose role is R, then another agent with the same role R is sought as a replacement. If no agent is found, then we search for agents whose role has the same obligation or power as R; otherwise we search for agents which are not prohibited to do the task. If the task needs expertise (or capability), then we search for agents with such expertise. If the best matching agent has no power, but the task needs power, then the agent is granted such power and the task is allocated to this agent. If the task does not need any expertise or power, we can allocate this task to any agent which is not prohibited to do it. On the other hand, if the resource of a task is not available, we search for sibling resources in the ontology for replacement. The replacement is checked to see if it fulfills the constraints of the task. We replace the alternative resource to the task (by updating the ontology) and check the consistency of the ontology. If the ontology is consistent, then the resource can be used for replacement. We can ilustrate this resource replacement using our workflow example. We view a supplier as a type of resource. If a supplier has no equipment to match what is requested, then alternative sibling suppliers are sought. In this case, supplier_service is the resource for place_order. However, we cannot choose suppliers whose rating is less than two, otherwise the ontology will be inconsistent (see axiom (11) above).

We now consider a scenario where an exception arises during the enactment of the Equipment Purchase (EP) workflow. As described above, when the HoD goes on vacation he/she is supposed to appoint another member of the department to temporarily act as HoD to cover the vacation period. We describe how to deal with the situation where the HoD has failed to do this. The EP workflow then gets stuck at place 5. The exception is picked up by the secretary who is notified (by the agent platform) of a delivery failure on her message (which should pass control to the HoD and enter workflow place 5). The exception handling routine is invoked, and this requires that the secretary propagate the problem to the next higher authority above the HoD. The ontology $\mathcal{O}$ is queried to get the manager of HoD (i.e., the next higher agent in the hierarchy). This query in SPARQL [28] can be done as follows:

algorithm EXCEPTION-HANDLING
// A message from Ag_1 cannot be sent to Ag_2 (whose role is R)
// a task T with ID and a time limit in a workflow WF,
// an upper management agent Ag_h is informed.
If message_delivery(Msg) == failed,
  Ag_h := RunQuery("SELECT ?Ag_h WHERE {"Ag_2" uni:managedBy ?Ag_h }")
  speech_act(Ag_1, Ag_h, $inform$, Msg, [WF, ID])
  // search for an alternative agent with the same role R
  Ag_X := RunQuery("SELECT ?Ag WHERE {?Ag uni:hasRole uni:R }")
  if Ag_X != null,
    allocate_task(Ag_X, [T, Time_limit]), return;
  // search for an alternative agent Ag_Y with the same obligation, or power
  if power(Ag_Y, T) or obliged(Ag_Y, T),
    allocate_task(Ag_Y, [T, Time_limit]), return;
  if not prohibited(Ag_Y, T),
    if T needs expertise,
      AgList := RunQuery("SELECT ?Ag WHERE {?Ag uni:hasExpertise uni:Ep}")
      if $Ag \in$ AgList
        if T needs power && not power(Ag_X, T),
          assign_power(Ag_Y, T)
          allocate_task(Ag_Y, [T, Time_limit]), return;
        else return;
    else if T needs power,
      assign_power(Ag_Y, T)
      allocate_task(Ag_Y, [T, Time_limit]), return;
    else allocate_task(Ag_Y, [T, Time_limit]), return;

// if the resource for task T is not available, a sibling resource is searched
If isAvailable(Resource) == false,
  ResSiblings := QueryOntology(sibling(Resource))
  For each r in ResSiblings
    if consistentOntology(T, r) == true,
      assign_resource(T, r)
      return;

Fig. 5: Algorithm of Exception Handling

```
Prefix  uni: ⟨http://www.csd.abdn.ac.uk/∼jlam/University.owl⟩
SELECT  ?staff
WHERE {?staff uni:hasRole ?role .
          ?hod uni:managedBy ?role .
          ?hod rdf:type uni:HoD}
```

The secretary sends the undelivered message to the HoC, and this allows the HoC to know the relevant variable bindings; in this case the HoC will know that the "Equipment" is robotics equipment. This HoC agent then inspects the tasks in the workflow at this place, in order to find a suitable agent who can perform them. As mentioned above, there are only four types of task: querying expertise, a local service query, a web service, or the sending of a speech act. Local services and Web services could be done by any agent, however querying expertise can only be done by an agent with the

required expertise. Therefore when the manager agent inspects the tasks which need to be delegated to a new agent, for each "query_expertise" task, the manager must find an agent that has the required capability. The second parameter of *query_expertise* indicates the type of expertise required (the variable "Equipment" is bound to "robotics"). Thus when the HoC inspects this part of the workflow by querying the ontology (see axiom 8 above), she knows that in order to execute this it is required that the agent must have expertise in robotics. The HoC will perform the following ontology query to find an agent with the appropriate expertise. Axioms 4 to 7 implicitly encode the knowledge that "dave" has expertise in robotics, thus "dave" will be the result of the query.

```
Prefix  uni: ⟨http://www.csd.abdn.ac.uk/∼jlam/University.owl⟩
SELECT  ?person
WHERE { ?person uni:hasExpertise "Robotics" }
```

A further problem is that no agent is empowered to authorise an equipment purchase, except the HoD (see line (6) in Section 3.2). The HoC must rectify this situation by nominating a suitable agent who could authorise the purchase. In general for speech acts that need to be delegated to a new agent, the manager may search the institutional facts to find a suitably empowered agent. In some cases, the manager of a group of agents may be empowered to grant new powers to the team he manages, and this is an alternative which can be used to ensure that a nominated agent can fulfill all the tasks required at a place in the workflow. In our example the HoC can grant the appropriate power to any agent according to power (5) above.

7 Related Work

Many research efforts have been undertaken on distributed workflow enactment mechanisms based on the agent paradigm, their aim is to support flexible and adaptive workflows in open and dynamic environments. In this section, we focus on exception handling in multi-agent systems and workflow management systems.

7.1 Multi-agent Systems

Exception handling in the agent community has been researched in order to build more reliable multi-agent systems. Klein and Dellarocas [20] proposed the use of specialised agents that handle exceptions. Their approach focuses on observing agent behaviour, diagnosing the possible fault and taking appropriate remedial action. The exception handling service is a centralised approach; the service is characterised as a kind of coordination doctor which actively diagnoses agents' illnesses and prescribes specific treatment procedures. The aim is to simplify agent development and have the agent infrastructure provide the fault-tolerance. Klein and Dellarocas [22] constructed a semi-formal Web-accessible repository of exception handling expertise for learning purposes. They firstly identified an exception taxonomy which is a hierarchy of exception types, and then described the exception management meta-process. The meta-process specifies which handlers should be used when for what exceptions. Klein et al. [21] describe a domain-independent exception handling services approach to increasing robustness in open agent systems. A directory of agents is used to keep track of the "death" of agents, so that exception which arise due to "agent death" problems can be handled with minimal resource wastage. All of these works use some device which is added into the system to deal with exceptions, for example: specialised agents, an exception repository, or a directory to keep track of agents. In contrast, our approach aims to en-

dow the agents of the system themselves with the ability to deal with exceptions by querying ontologies and changing the organisational structure.

7.2 Workflow Management Systems

The standard approach to representing workflows in business is the Business Process Execution Language (BPEL) [19]. Buhler and Vidal [6] proposed the use of the Business Process Execution Language for web Services (BPEL4WS) as a specification language for expressing the initial social order of a multi-agent system, which can then intelligently adapt to changing environmental conditions. Since BPEL4WS describes the relationship between Web services in the workflow, agents representing the Web service would know their relationships a priori. Buhler and Vidal [5] further proposed to integrate agent services into BPEL4WS-defined workflows. The strategy is to use the Web Service Agent Gateway to slide agents between a workflow engine. The workflow engine calls the target agent instead of the Web service directly; the agent can be configured to respond flexibly. From the above mentioned papers, Buhler and Vidal describe approaches to create adaptive workflow capability through decentralised workflow enactment mechanisms that combine Web services and agent technologies; they claim that that agents representing semantic Web services can organise themselves to enact workflows flexibly.

Similarly, Guo et al. [13–15] address the downside of current workflow engines which are centralised and suffer from single point-of-failure weakness; they describe the development of a distributed multi-agent workflow enactment mechanism from a BPEL4WS specification. They proposed a syntax-based mapping between some of main BPEL4WS constructs to the Lightweight Coordination Calculus (LCC). The authors claim that with their approach, a BPEL4WS specification can be used directly for constructing a multi-agent system using Web services composition; therefore, its benefit is that existing workflow development methodologies and business process models can be used as much as possible for MAS development. The papers do not cover how their approach deals with unexpected circumstances during the enactment of workflows at runtime.

In [10, 29] the authors address the same problems of workflow management systems which have rigid, centralised architectures that do not offer sufficient flexibility for distributed organisations. In their system, Coloured Petri Nets (CPNs) are used to represent the workflow. A process agent executes a workflow instance by assigning tasks to resource agents which can be seen as representing Web services and can be dynamically discovered. Their aim is to assign suitable resource dynamically to a task. However the resulting solution is still centralised to some extent, as an agent is managing the enactment of the workflow and calling on resources to do the individual tasks; a truly agent-based enactment should allow each agent to control their part of the workflow (as in the other related works above). In our work, the workflow tasks are dealt by autonomous agents without a central control from a manager agent, resulting in systems that exhibit decentralised flow control.

Similar to our approach, the above mentioned works aim to enable the flexibility of decentralised multi-agent workflow-enactment to deal with dynamic Web services; agents are able to intelligently diverge from prescribed workflows when needed. Although the above works sometimes do not give the details of how they would deal with exceptional circumstances, their techniques can be extended to deal with exceptions. We further include organisational knowledge, such as agents' capabilities represented

in OWL ontologies; agents are then able to modify the roles, powers, obligations in the organisation. We believe that the use of ontologies to describe aspects of the organisation and domain can be valuable here, as agents are part of an organisation and will be unable to deal with exceptions entirely on their own.

7.3 The Commitment Approach

Singh and Huhns [33] propose interaction-oriented programming (IOP) as a technique for engineering multi-agent systems to flexibly enact workflows. With the emphasis on facilitating agent autonomy and flexibility in interactions, IOP describes interactions using high-level primitives. The high-level primitive which Singh and Huhns focus on is the "commitment". By making this a first class object agents are able to reason about their commitments to others and vice versa, and can make autonomous decisions about how to act. With commitments capturing the high level meaning of an interaction, agents have the opportunity to intelligently reason about alternative ways of satisfying the high-level goals of an interaction. This approach potentially allows a much greater flexibility than our approach above, however the challenge is to develop an appropriate agent reasoning mechanism to enable such adaptive behaviour. This is an interesting area for future investigation.

Mallya and Singh [24], building on the commitment approach, have proposed novel methods to deal with exceptions in a protocol. They distinguish between expected and unexpected exceptions. Unexpected exceptions are closest to the types of exceptions we tackle here. Mallya and Singh's solution makes use of a library of sets of runs (sequences of states of an interaction) which could be spliced into the workflow at the point where the exception happens. Mallya and Singh do not describe how these sets of runs can be created, but it is likely that one would need access to observed sequences from enactments of similar workflows. The aim of the commitment approach is in line with our work, as it endows the agents with some understanding of the meaning of the workflow they are executing, by giving them knowledge of the commitments at each workflow place. This would make it possible for agents to find intelligent solutions when exceptions arise. Similarly, in our approach, agents are endowed with semantic knowledge (represented in an ontology) about the capabilities and hierarchy of the other agents so that they can find suitable candidates to execute tasks in the case of exceptions. In the future work, we would like to merge our approach with the commitment protocols approach to model business processes, in which commitments among roles and business policies can be described.

8 Summary, Discussion & Conclusion

We have shown how workflow exceptions at the coordination or organisational level could be handled by agents. Our solution has two components: (1) a flexible agent institution, so that when an impasse arises in a workflow, the agents can reorganise to find an alternative path to circumvent the problem; by "flexible" we mean that the institution must include speech acts which allow agent roles, powers and normative relations to be altered at run-time; and (2) agents endowed with semantic knowledge about the capabilities and hierarchy of the other agents so that they can find suitable candidates to execute tasks that are preventing the workflow from progressing; we provided this knowledge via an OWL ontology. We illustrated our approach with a University institution example to illustrate how exceptions can be dealt with by agents via ontology reasoning, a decentralised collection of agents in an organisation cooperate to maintain

the workflow's integrity. In general the type of exception we can handle is one where a task which needs to be done cannot be done because an agent is unavailable, or available agents are lacking some attribute. In this case agents perform ontological reasoning or querying to find alternatives. In our example we simply adjusted the powers of an agent so that he could fulfil the task. More generally we may allow agents in an organisation to decide to outsource a task, or to hire a consultant, or send a member of the organisation for training. To tackle more varied and more complex types of exceptions we foresee that agents will need to be given more knowledge about their domain and the tasks and capabilities available. This will require a combination of more ontological knowledge, and also appropriate reasoning mechanisms so that agents can exploit the knowledge.

Our solution has split the knowledge of the system in two parts: the institutional facts (powers, roles, speech act processing rules, etc.) are implemented with Prolog rules, while the knowledge about the tasks the agents will be able to undertake (knowledge of capabilities of the agents, constraints on roles, and the hierarchy of the organisation) are represented with an OWL ontology. In fact some knowledge is represented twice; the knowledge about membership of roles is included in both the ontology and the institutional facts. This information is important both for deciding the institutional updates, and for finding suitable candidates when coping with exceptions; however, this duplication is not optimal and will be addressed in future work.

References

1. H. Aldewereld, F. Dignum, L. Penserini, and V. Dignum. Norm dynamics in adaptive organisations. In *3rd International Workshop on Normative Multiagent Systems (NorMAS 2008)*, July 2008. (to appear).
2. A. Artikis. *Executable Specification of Open Norm-Governed Computational Systems*. PhD thesis, University of London, November 2003.
3. F. Baader, I. Horrocks, and U. Sattler. Description logics as ontology languages for the semantic web. In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning: Essays in Honor of Jörg Siekmann on the Occasion of His 60th Birthday*, number 2605 in Lecture Notes in Artificial Intelligence, pages 228–248. Springer, 2005.
4. S. Bechoffer, F. van Harmlen, J. Hendler, I. Horrocks, D. McGuinnes, P. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language Reference, February 2004. http://www.w3.org/TR/owl-ref/.
5. P. Buhler and J. M. Vidal. Integrating agent services into BPEL4WS defined workflows. In *Proceedings of the Fourth International Workshop on Web-Oriented Software Technologies*, 2004.
6. P. Buhler and J. M. Vidal. Towards adaptive workflow enactment using multiagent systems. *Information Technology and Management Journal*, 6(1):61–87, 2005.
7. Paul Buhler, José M. Vidal, and Harko Verhagen. Adaptive workflow = web services + agents. In *Proceedings of the International Conference on Web Services*, pages 131–137. CSREA Press, 2003.
8. O. Cliffe, M. De Vos, and J. Padget. Answer Set Programming for Representing and Reasoning about Virtual Institutions. In Katsumi Inoue, Satoh Ken, and Francesca Toni, editors, *Computational Logic for Multi-Agents (CLIMA VII)*, volume 4371 of *LNCS*, pages 60–79. Springer-Verlag, May 2006.
9. V. Dignum. *A Model for Organizational Interaction: Based on Agents, Founded in Logic*. PhD thesis, University of Utrecht, Utrecht, The Netherlands, 2004.
10. L. Ehrler, M. Fleurke, M. Purvis, and B. T. R. Savarimuthu. Agent-based workflow management systems(WfMSs) : Jbees- a distributed and adaptive wfms with monitoring and controlling capabilities. *Information Systems and E-Business Management*, 4(1):5–23, 2006.

11. M. Esteva. *Electronic Institutions: from Specification to Development*. PhD thesis, Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, 2003. IIIA monography Vol. 19.
12. F. Guerin and W. Vasconcelos. Component-based standardisation of agent communication. In Matteo Baldoni, Tran Cao Son, M. Birna van Riemsdijk, and Michael Winikoff, editors, *Declarative Agent Languages and Technologies V (DALT)*, volume 4897 of *Lecture Notes in Computer Science*, pages 227–244. Springer, 2007.
13. L. Guo, D. Robertson, and Y. Chen-Burger. Enacting the distributed business workflows using bpel4ws on the multi-agent platform. In *Third German Conference on Multiagent System Technologies, MATES 2005*, pages 35–46, Koblenz, Germany, 2005.
14. L. Guo, D. Robertson, and Y. Chen-Burger. A generic multi-agent system platform for business workflows using web services composition. In *2005 IEEE Intelligent Agent Technology*, pages 301–307, Compiegne University, France, 2005.
15. L. Guo, D. Robertson, and Y. Chen-Burger. Using multi-agent platform for pure decentralised business workflows. *Journal of Web Intelligence and Agent System*, 6(3), 2008.
16. I. Horrocks and P. F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. In Dieter Fensel, Katia Sycara, and John Mylopoulos, editors, *Proc. of the 2nd International Semantic Web Conference (ISWC 2003)*, number 2870 in Lecture Notes in Computer Science, pages 17–29. Springer, 2003.
17. I. Horrocks and U. Sattler. A tableaux decision procedure for $\mathcal{SHOIQ}$. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*, pages 448–453, 2005.
18. IBM. BPWS4J, 2004. http://www.alphaworks.ibm.com/tech/bpws4j.
19. IBM, BEA Systems, Microsoft, SAP AG, and Siebel Systems. Business process execution language for web services version 1.1. Technical report, July 2003. http://www.ibm.com/developerworks/library/specification/ws-bpel/.
20. M. Klein and C. Dellarocas. Exception handling in agent systems. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents*, pages 62–68, New York, NY, USA, 1999. ACM.
21. M. Klein, J. Rodriguez-Aguilar, and C. Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):179–189, 2003.
22. Mark Klein and Chrysanthos Dellarocas. Towards a systematic repository of knowledge about managing multi-agent system exceptions. Technical Report ASES Working Report ASES-WP-2000-01, Massachusetts Institute of Technology, 2000.
23. B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, 2006.
24. A. U. Mallya and M. P. Singh. Modeling exceptions via commitment protocols. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 122–129. ACM, 2005.
25. T. Oinn, M. J. Addis, J. Ferris, D. J. Marvin, M. Senger, T. Carver, M. Greenwood, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workows. *Bioinformatics Journal IEEE Computer*, 20(17):3045–3054, 2004.
26. M. Paolucci, T.Kawamura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 333–347, London, UK, 2002. Springer-Verlag.
27. J. Pitt, L. Kamara, M. Sergot, and A. Artikis. Formalization of a voting protocol for virtual organizations. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 373–380, New York, NY, USA, 2005. ACM.
28. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 15 January 2008. Available at `http://www.w3.org/TR/rdf-sparql-query/`.

29. M. Purvis, B. T. R. Savarimuthu, and M. Purvis. A multi-agent based workflow system embedded with web services. In *second international workshop on Collaboration Agents: Autonomous Agents for Collaborative Environments (COLA 2004)*, pages 55–62, Beijing, China, 2004. IEEE/WIC Press.
30. N. Russell, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Workflow exception patterns. In *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE 06)*, volume 4001 of *LNCS*, pages 288–302. Springer-Verlag, 2006.
31. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.
32. M. P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, 1998.
33. M. P. Singh and M. N. Huhns. Multiagent systems for workflow. *International Journal of Intelligent Systems in Accounting, Finance and Management*, 8:105–117, 1999.
34. K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan. Automated discovery, interaction and composition of semantic web services. *Journal of Web Semantics*, 1(1):27–46, September 2003.
35. M. Uschold and M. Gruninger. Ontologies: Principles, Methods and Applications. *The Knowledge Engineering Review*, 11(2):93–136, 1996.
36. WfMC. Workflow management coalition terminology and glosary. Technical Report WFMC-TC-1011, Workflow Managemtn Coalition, 1999.
37. K. Wiesner, R. Vaculín, M. Kollingbaum, and K. Sycara. Recovery mechanisms for semantic web services. In René Meier and Sotirios Terzis, editors, *DAIS*, volume 5053 of *Lecture Notes in Computer Science*, pages 100–105, 2008.