Building Multi-Agent Systems for Workflow Enactment and Exception Handling*

Joey 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. Workflows represent the coordination requirements of various distributed operations in an organisation; workflows neatly capture business processes, and are particularly suitable for cross-organisational enterprises. 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, via software agents which are intelligent and autonomous, and respect the constraints in a norm-governed organisation. Agents should bring flexibility and robustness to the workflow enactment process. In this paper, we describe a method for building a norm-governed multi-agent system which can enact a set of workflows and cope with exceptions. We do this by providing agents with knowledge of the organisation, the domain, and the tasks and capabilities of agents. This knowledge is represented with Semantic Web languages, and agents can reason with it to handle exceptions autonomously.

1 Introduction

The Workflow Management Coalition 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" [21]. 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 we need intelligence to deal with the unexpected; or, more accurately, we can use Artificial Intelligence techniques to deal with a set of possible exceptions, where the response to each situation is not hand-coded at design-time, but rather worked out at run-time by reasoning with hand-coded knowledge. Typical workflow management systems (e.g., [12, 13]) are centralised and rigid; they cannot cope with the unexpected flexibly. Moreover, they have not been designed for dynamic environments requiring adaptive responses. To overcome

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

this we argue that it is 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.

We are interested in building multi-agent systems (MASs) which simulate the operations in large organisations, and so must adhere to constraints defined by the organisation[1]. In this paper we explore the issue of building agent systems which can enact workflows in a distributed fashion, and which can cope with exceptions as they arise. We propose a method for building such agent systems, given the appropriate knowledge as an input. The knowledge input to the system is divided into three main components, as illustrated in Figure 1. Firstly there

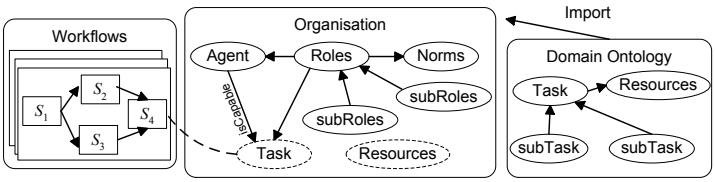

Fig. 1: Overview of the Proposed Approach

is the *organisational knowledge*, consisting of such things as roles, norms, role classification, and resources available. Secondly there are the *workflows*, describing the tasks to be executed, and appropriate flow of control, and also including variable definitions which are used to control flow. Thirdly there is the *domain ontology*, describing concepts of the world including tasks and resources. Some tasks are atomic and can be directly executed by agents, but some tasks require a workflow to be executed. In this case the name of the task matches the name of a workflow. As indicated by the dashed line there are links between the workflow tasks and the organisation and imported domain ontology. Tasks and resources are not described in the organisation directly; the domain ontology is imported to the organisation ontology. This knowledge allows us to firstly allocate tasks in an agent system where workflow tasks are distributed among the agents in a way which is consistent with the organisational constraints. Secondly, it allows the agents to enact the workflows, updating the organisation as appropriate. Thirdly it allows agents to cope with exceptions as they arise, because the agents can query the ontology to find alternative agents or tasks when problems arise.

In our system the workflow specifications and the ontologies are available to be queried by any of the agents in the system, and the ontologies can also be updated as events add or modify instances in the ontology. This requires a centralised service which maintains the knowledge, and updates it, when instructed to by an agent. This centralised approach is somewhat undesirable in a distributed agent system (lack of robustness, and scalability), however additional robustness could be introduced by having multiple copies of the knowledge, and some synchronisation processes to ensure that all copies are identical. Both of

[1] Our focus for the moment is simulation, but eventually we envisage that our agents will support real human users in executing tasks as part of human-agent teams.

these possibilities entail challenges which go beyond the scope of the current paper; we will merely assume the knowledge is available.

This paper is structured as follows. In Section 2 we briefly introduce Semantic Web languages. In Section 3 we describe norm-governed organisations represented in Semantic Web languages. We describe the representation of workflows and explain allocation of tasks in Sections 4 and 5 respectively. The details of agents enacting workflows and dealing with exceptions are given in Sections 6 and 7 respectively. Section 8 looks at related work and Section 9 concludes and proposes future work.

2 Semantic Web Languages

The OWL-DL [6] ontology language is a variant of the $\mathcal{SHOIN}(\mathbf{D})$ Description Logic [7], which provides constructs for full negation, disjunction, a restricted form of existential quantification, cardinality restrictions, and reasoning with concrete datatypes. We make use of the open world assumption, which requires that something is false if and only if it can be proved to contradict other information in the ontology. Since we assume a MAS as an open system, its knowledge of the world is incomplete, and the knowledge is extendable. If a formula cannot be proved true or false, we draw no conclusion[2].

Formally, an ontology $\mathcal{O}$ consists of a set of *terminology* axioms $\mathcal{T}$ (TBox), *role* axioms (RBox) and assertional axioms $\mathcal{A}$ (ABox), that is, $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$. An axiom in $\mathcal{T}$ is either of the form $C \sqsubseteq D$ or $C \doteq D$, where C and D are arbitrary concepts (aka. classes in OWL); the RBox contains assertions about roles (such as functional, transitive roles) and role hierarchies; 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 (aka. instances in OWL) and R is a role name (aka. a datatype or object property in OWL); b is a filler of the property R for a). The meaning of concepts, roles and individuals is given by an interpretation. An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a non-empty set of individuals (the *domain* of the interpretation) and an *interpretation function* $(\cdot^{\mathcal{I}})$, which maps each atomic concept $\mathsf{CN} \in \mathbf{C}$ ($\mathbf{C}$ is a set of concept names) to a set $\mathsf{CN}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and each atomic role $R \in \mathbf{R}$ ($\mathbf{R}$ is a set of role names) to a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The interpretation function can be extended to give semantics to concept descriptions. An interpretation $\mathcal{I}$ is said to be a model of a concept C, or $\mathcal{I}$ models C, if the interpretation of C in $\mathcal{I}$ is not empty. A concept A is *unsatisfiable* w.r.t. a terminology $\mathcal{T}$ if, and only if, $A^{\mathcal{I}} = \emptyset$ for all models of $\mathcal{I}$ of $\mathcal{T}$. An ontology $\mathcal{O}$ is *inconsistent* if it has no models.

OWL DL benefits from many years of DL research, leading to well defined semantics, well-studied reasoning algorithms, highly optimised systems, and well understood formal properties (such as complexity and decidability) [3].

The Semantic Web Rule Language (SWRL)[3] extends the set of OWL axioms to include Horn-Clause-like rules that can be expressed in terms of OWL classes and that can reason about OWL instances. SWRL provides deductive reasoning

[2] We can reason in an inconsistent ontology by tolerating a limited number of contradictions. A formula is *undefined* (or *undetermined*) if it entails neither true nor false; a formula is *overdefined* (or *over-determined*) if it entails both true and false.

[3] http://www.w3.org/Submission/SWRL/

capabilities that can infer new knowledge from an existing OWL knowledge base. However, OWL DL extended with SWRL is no longer decidable. To make the extension decidable, Motik et al. [15] propose DL-safe rules where the applicability of a rule is restricted to individuals explicitly named in a knowledge base. For example: $\mathsf{parent(x,y)} \wedge \mathsf{brother(y,z)} \wedge \mathcal{O}(\mathsf{x}) \wedge \mathcal{O}(\mathsf{y}) \wedge \mathcal{O}(\mathsf{z}) \rightarrow \mathsf{uncle(x,z)}$ where $\mathcal{O}(x)$ must hold for each explicitly named individual x in the ontology. Hence, DL-safe rules are SWRL rules that are restricted to known individuals.

The language SPARQL-DL [18] supports mixed TBox/RBox/ABox queries for OWL-DL ontologies. Throughout the paper we will use qnames to shorten URIs with rdf, rdfs, and owl prefixes to refer to standard RDF (Resource Description Framework), RDF-S (RDF Schema) and OWL namespaces, respectively. We also use the prefix dom and org to refer to the namespace of the Domain and Organisation ontologies. Our agents will be able to query the ontology to access the knowledge. For example, agents can search for all roles working in the finance department which are obliged to perform task "WriteReport" by using this query[4]:

Type(_:x, ?role), PropertyValue(_:x, org:worksIn, _:y), Type(_:y, org:FinanceDept),
PropertyValue(_:x, org:isObliged, _:z), Type(_:z, dom:WriteReport)

3 Norm-Governed Organisations

In this section, we describe how to represent roles, role classification, and norms using OWL and SWRL. Our specification in this section is adequate to allow agents to query an organisation at a certain point in time and ask questions such as "is agent x prohibited from doing task t", or "is agent y empowered to do task b". However, we do not provide a specification for how the organisation is changed as a result of agents performing speech acts, or as a result of other events. We assume that the system provides other specifications for this purpose, and we focus only on the specifications necessary for workflow enactments.

We represent the agent organisation using Semantic Web languages. The agent literature has many examples of different approaches to the specification of norm-governed organisations, using languages such as the event calculus or C+, for example [2]. Semantic Web languages are not as expressive as these, but they have the advantage of having very efficient DL reasoners, and being standardised. To specify the organisational knowledge relevant to our workflows we can get by without the expressiveness of more sophisticated languages; for example we only need to do static queries on current knowledge, and we do not require the ability to reason over different time intervals. We do allow modalities to change over time, but we only represent the new (changed) ontology; agents can only query the current version, not any prior states.

3.1 Roles and their Constraints

The ontology we propose in this paper models the concepts of a role, its role classification(s), restrictions on roles (such as mutually exclusive roles, cardinality, prerequisite roles) [16], and other aspects of the organisation. Roles are modelled in a classification to reflect the subsumption of role descriptions. Sub-roles inherit the properties from the super-roles; the properties of a sub-role override

[4] Type(?a,?C) gives the most specific classes an instance belongs to.

those of its super-roles if the sub-role has more restrictive properties (the sub-role cannot be less restrictive or the ontology would be inconsistent). Cardinality restrictions can be used for example to restrict the number of agents a task can be assigned to. Disjointness axioms can represent separation of duty restrictions.

We now give an example specification to illustrate these ideas. In Figure 2, a role classification is shown. Sub-roles inherit the properties from super-roles. For example, Staff are obliged to work from 9am to 5pm during weekdays; its sub-roles inherit this obligation. The properties of a sub-role override those of its super-role. Manager is permitted to employ staff; its sub-roles inherits this property. However, this property of the AccountingManager is more restrictive; it is only permitted to employ AccountingStaff (see axioms 4 and 5 below). For the cardinality restrictions, we can model that only one agent can fill the role of the general manager (see axioms 11 and 12 below); a member of staff works in exactly one department (see axiom 6 below). An example of mutually exclusive roles is that a department manager cannot be a general manager simultaneously (see axiom 8 below). An example of separation of duty is that a staff submitting a project proposal is prohibited from approving the proposal (see axiom 9 below). Prerequisite roles means that a person can be assigned to role $r1$ only if the person already is assigned to role $r2$ (see axiom 10).

(1) Programmer $\sqcup$ Manager $\sqcup$ Secretary $\sqsubseteq$ Staff

(2) DeptManager $\sqcup$ GeneralManager $\sqsubseteq$ Manager

(3) AccountingManager $\sqcup$ ITManager $\sqsubseteq$ DeptManager

(4) Manager $\sqsubseteq \exists$ isPermitted.($\exists$ employs.Staff) $\sqcap \forall$ isPermitted.($\exists$ employs.Staff)

(5) AccountingManager $\sqsubseteq \exists$ isPermitted.($\exists$ employs.AccountingStaff) $\sqcap \forall$ isPermitted.($\exists$ employs.AccountingStaff)

(6) Staff $\sqsubseteq =1$ worksIn

(7) range(worksIn) $=$ Department

(8) DeptManager $\sqsubseteq \neg$ GeneralManager

(9) Staff(x) $\wedge$ ProjectProposal(p) $\wedge$ ApproveProjectProposal(act) $\wedge$ submits(x,p) $\wedge$ approves(act,p) $\wedge\mathcal{O}$(x) $\wedge\mathcal{O}$(p) $\wedge\mathcal{O}$(act) $\rightarrow$ isProhibited(x,act)

(10) AccountingManager $\sqsubseteq \exists$ prerequisites.Accountant

(11) GeneralManager $\sqsubseteq =1$ takenBy

(12) range(takenBy) $=$ Agent

(13) canDelegate $\sqsubseteq$ hasPower

(14) Staff $\sqsubseteq \exists$ isObliged.($\exists$ works.(Weekdays $\sqcap$ OfficeHour)) $\sqcap \forall$ isObliged.($\exists$ works.(Weekdays $\sqcap$ OfficeHour))

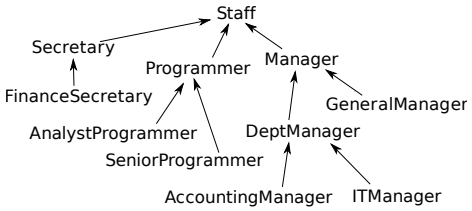

Fig. 2: Roles and a Role Classification

3.2 Normative Notions

We firstly describe norms concerning agents performing some task Task. We model permission[5], obligation, prohibition and power as isPermitted, isObliged, isProhibited and hasPower OWL object properties to relate roles in the organisation and tasks. Their domain and range is Role and Task respectively. Permissions allow the agent to achieve a state of affairs or perform an action (see for example axioms 4 and 5 above). Permission and prohibition are distinct from power because a member may be empowered to do something even though he is prohibited from doing it. Prohibitions forbid the agent from achieving a state of affairs or performing an action (see for example axiom 9 above). An obligation indicates that some act has to be done (see for example axiom 14 above). It is common to specify a time-limit or a condition for obligations. The axiom 14 above states a conditional obligation, such that staff have to work during office hours and weekday. Due to limited space, we will not describe time-limit constraints in the paper. We now model the relations between the basic notions; the relations can be equivalence, compatibility or incompatibility (or conflict) [20]. The following rules list some of these relations.

(1) If an act is permitted and prohibited then there is a conflict.

$$\mathsf{isPermitted(x,act)} \wedge \mathsf{isProhibited(x,act)} \wedge \mathcal{O}(\mathsf{x}) \wedge \mathcal{O}(\mathsf{act}) \rightarrow \mathsf{owl{:}Nothing(x)}$$

(2) If an act is obligatory, then it is permitted.

$$\mathsf{isObliged(x,act)} \wedge \mathcal{O}(\mathsf{x}) \wedge \mathcal{O}(\mathsf{act}) \rightarrow \mathsf{isPermitted(x,act)}$$

(3) If an act is obligatory and prohibited then there is a conflict.

$$\mathsf{isObliged(x,act)} \wedge \mathsf{isProhibited(x,act)} \wedge \mathcal{O}(\mathsf{x}) \wedge \mathcal{O}(\mathsf{act}) \rightarrow \mathsf{owl{:}Nothing(x)}$$

(4) If a prohibited act is performed then there is a violation.

$$\mathsf{performed(x,act)} \wedge \mathsf{isProhibited(x,act)} \wedge \mathcal{O}(\mathsf{x}) \wedge \mathcal{O}(\mathsf{act}) \rightarrow \mathsf{violated(x,act)}$$

Compared to standard deontic logic, here these deontic notions are being given quite a different interpretation by the Semantic Web languages. For example in deontic logic we could say that "obliged" is equivalent to "not permitted not to", however in Semantic Web languages we cannot express this. SWRL does not allow us to negate the atoms within the scope of isPermitted(...). Nevertheless, the Semantic Web version of these norms seems to be adequate for representing simple agent scenarios, as illustrated in our examples below. One thorny issue is contraposition; a DL axiom such as $A \sqsubseteq B$ entails $\neg B \sqsubseteq \neg A$. This entailment is not desirable in exceptional situations where there is a special condition such that some type of A is not a B. The only way we can deal with this is to explicitly state all exceptions in the axioms, for example $\mathsf{Bird} \sqcap \neg \mathsf{Penguin} \sqcap \neg \mathsf{Ostrich} \sqsubseteq \mathsf{CanFly}$.

In this paper we distinguish between institutional tasks (such as authorising a purchase), and physical tasks (such as printing a document). An institutional task can only be performed by an agent which has power to do that task. For example we say that action 'manager x employs staff y' is valid if x is empowered to employ a staff at that time, therefore y is now a member of staff. Otherwise, it is an invalid action due to its lacking of institutional power. The following axioms mean that a manager has the power to employ staff; when the manager performs EmployStaff, the person will become a staff:

[5] Explicitly defined permission means *strong* permission in our system. Undefined permission axioms represent *weak permission*.

EmployStaff $\doteq$ $\exists$ employ.Staff $\sqcap$ $\forall$ employ.Staff

Manager $\sqsubseteq$ $\exists$ hasPower.EmployStaff

Manager(m) $\wedge$ Person(p) $\wedge$ EmployStaff(act) $\wedge$ hasPower(m,act) $\wedge$ employ(act,p) $\wedge$ performed(m,act) $\wedge\mathcal{O}$(m) $\wedge\mathcal{O}$(p) $\wedge\mathcal{O}$(act) $\rightarrow$ Staff(p)

4 Workflows

We now introduce a representation for workflows. A common way to represent a workflow is using Petri Nets [19] or BPEL (Business Process Execution Language) [1]. In this paper we represent a workflow as a digraph, which is a simplified and minimalistic way to capture the basic concepts of workflows.

Definition 1. *A workflow is a tuple* $\langle N, \mathbf{S}, \mathbf{E}, s_0, \mathbf{S}_f \rangle$*, where*

1. N *is the name of the workflow,*
2. $\mathbf{S}$ *is a finite set of states of the form* $\langle id, T \rangle$*, where* id *is the number identifying this state, and* T *is the task;*
3. $\mathbf{E}$ *is a set of edges linking states. Edges take the form* $\langle id_1, l, v, id_2 \rangle$*, where* id_1 *is the state this edge leaves from,* id_2 *is the state this edge arrives at,* v *is the variable associated with the edge, and* l *is a label indicating the type of edge. There are four types of edge,* $l \in \{AND, OR, JOIN\text{-}AND, JOIN\text{-}OR\}$ *where AND-edges and OR-edges describe exclusive-or branches, and JOIN-AND-edges and JOIN-OR-edges describe joins. For any pair of states with multiple edges linking them, the edges must be of the same type;*
4. s_0 *is the initial state of the workflow, and* $\mathbf{S}_f \subseteq \mathbf{S}$ *is the set of final states.*

We consider a travel request workflow example in a company. Figure 3 graphically depicts the example. Figure 4 shows the representation of the workflow. The figure annotates edges with the name of the variable associated with the edge. We refer to *input* and *output* variables of the workflow; for example the state $\langle$2,checkRequestForm$\rangle$ has input variable "TravelRequestForm" and output variable "CorrectTravelRequestForm". In this example, we assume the workflow is triggered by an agent who issues a travel request. Firstly a travel request form is issued; the request form should be checked to have correct information. The checked form is then passed to be approved. The output of "approveTravelRequest" is either "ApprovedForm" or "RejectedForm". If the output variable is "ApprovedForm", the order for the travelling can be placed; otherwise, the request is rejected.

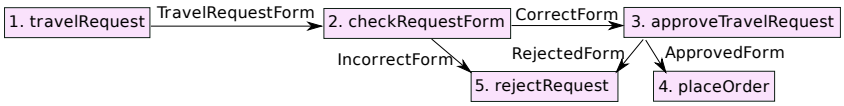

Fig. 3: Travel Request Workflow Example

5 Allocating Tasks to Agents

In this section we describe how we allocate tasks to software agents which, together, will enact a set of workflows. Agents are parameterised by the roles they take up – these roles dictate the tasks agents become responsible for.

$W = \langle$travelRequest_WF, $\mathbf{S}, \mathbf{E}, 1, \{4, 5\}\rangle$
$\mathbf{S} = \{\langle$1,travelRequest$\rangle$, $\langle$2,checkRequestForm$\rangle$, $\langle$3,approveTravelRequest$\rangle$, $\langle$4,placeOrder$\rangle$, $\langle$5,rejectRequest$\rangle\}$
$\mathbf{E} = \{\langle$1,AND,TravelRequestForm , 2$\rangle$, $\langle$2,OR,CorrectTravelRequesForm, 3$\rangle$,
$\langle$2,OR,IncorrectTravelRequesForm, 5$\rangle$ $\langle$3,OR,ApprovedForm, 4$\rangle$, $\langle$3,OR,RejectedForm, 5$\rangle\}$

Fig. 4: Travel Request Workflow as a Digraph

The input to the ontological creation of agents is a set of workflows $\mathcal{W}$ and an ontology $\mathcal{O}$, and the output is an updated ontology with a set of software agents introduced as subclasses of the Agent concept, with roles and tasks associated with them. In Figure 5 we show how we create agents in our ontology. The

algorithm *agent_creation*($\mathcal{W} = \{W_1, \ldots, W_n\}$, $\mathcal{O}$)
  $\mathcal{T} = \bigcup_{i=1}^{n} \mathbf{S}_i$, $\langle N_i, \mathbf{S}_i, \mathbf{E}_i, s_{0_i}, \mathbf{S}_{f_i}\rangle \in \mathcal{W}$
  for each role R_i in $\mathcal{O}$ **do**
    for each $\langle id, T\rangle \in \mathcal{T}$ **do**
      if $R_i \sqsubseteq \exists$ isObliged.T **then**
        $\mathcal{T} := \mathcal{T} \setminus \{\langle id, T\rangle\}$; $\mathcal{T}_i := \mathcal{T}_i \cup \{T\}$
      else if $R_i \sqsubseteq \exists$ hasPower.$T \sqcap \exists$ isPermitted.T **then**
        $\mathcal{T} := \mathcal{T} \setminus \{\langle id, T\rangle\}$; $\mathcal{T}_i := \mathcal{T}_i \cup \{T\}$
      else if $R_i \sqsubseteq \exists$ isPermitted.T **then**
        $\mathcal{T} := \mathcal{T} \setminus \{\langle id, T\rangle\}$; $\mathcal{T}_i := \mathcal{T}_i \cup \{T\}$
  if $\mathcal{T} \neq \emptyset$ **then fail** // org. cannot enact a workflow
  else
    for each role R_i in $\mathcal{O}$ with $\mathcal{T}_i = \{T_0^i, \ldots, T_m^i\}$ **do**
      $\mathcal{O} := \mathcal{O} \cup \{Ag_i \sqsubseteq$ Agent$\}$
      $\mathcal{O} := \mathcal{O} \cup \{Ag_i \doteq \exists$ hasRole.$(R_i) \sqcap \exists$ isCapable.$(T_0^i \sqcup \cdots \sqcup T_m^i)\}$
return $\mathcal{O}$;

Fig. 5: Creation of Agents in $\mathcal{O}$

algorithm collects in $\mathcal{T}$ all tasks of the workflows and distributes them among the roles of the organisation. The distribution gives priority to *i)* obligations, then *ii)* institutionalised power and permissions, and finally *iii)* permissions, captured in the algorithm by the order of the nested **if** constructs. All tasks should be distributed among roles, otherwise the algorithm fails, that is, the organisation represented in the ontology cannot enact one of the workflows. If all tasks have been assigned to roles, then for each role R_i we create in $\mathcal{O}$ a subclass Ag_i of Agent, with tasks $\mathcal{T}_i = \{T_0^i, \ldots, T_m^i\}$ associated to the agent via isCapable.

For each Ag_i in $\mathcal{O}$ we start up an independent computational process – a software agent – which will support the enactment of workflows. Each software agent, upon its bootstrapping, will use the definition of the subclass as the parameterisation of its mechanisms: the role and tasks associated to the agent will guide its behaviour, explained in Section 6. For simplicity, in our algorithm above, we assume that each agent will enact exactly one role; however this could easily be changed if required.

6 Enactment of Workflows

After allocation, the next step is that agents take up roles in the organisation and enact workflows. Agents plan their actions in real-time. The workflow provides an outline plan, but many of the details need to be decided by agents. During the enactment, an ontology is used by the agents to check what actions they can perform. There is a relationship between the tasks and variables in the

workflows, and the concepts and instances in the ontology. Each time the agents perform workflow tasks or assign values to variables, they update the instances in the ontology. Some agent actions will involve the consumption of organisational resources, in which case the agent will update the instances in the ontology. Thus the ontology maintains a record of the current status of the workflow enactment, as well as relevant aspects of the organisation. In this paper we avoid details of how the implementation could work, but the update of the ontology can be done by the agents whenever they are about to do a task; the agent sends an update instruction to the service which maintains the ontology.

We will detail the relationship between the ontology and the workflow enactment; this is easiest to illustrate by referring to an example. We continue with the travel request example from Section 4, and we add to it an ontology (see the axioms below) which describes roles, norms, and descriptions of tasks. The following axioms state the norms governing agents and the tasks to be executed.

(1) Secretary $\sqsubseteq \exists$ isObliged.checkRequestForm
(2) Manager $\sqsubseteq \exists$ hasPower.approveRequest
(3) DeptManager $\sqsubseteq \exists$ hasPower.approveTravelRequest
(4) Manager(m) $\wedge$ TravelRequestForm(f) $\wedge$ approveTravelRequest(act) $\wedge$ requestedFrom(f,m) $\wedge$ approves(act,f) $\wedge\mathcal{O}$(m) $\wedge\mathcal{O}$(m) $\wedge\mathcal{O}$(f) $\rightarrow$ isProhibited(m,act)
(5) checkRequestForm $\doteq \exists$ checks.(TravelRequestForm $\sqcap \exists$ isCorrect.xsd:boolean)
(6) Staff(s) $\wedge$ requestedFrom(f,s) $\wedge$ hasName(s,n) $\wedge$ filledName(f,n) $\wedge$ hasStaffID(s,id) $\wedge$ filledStaffID(f,id) $\wedge\mathcal{O}$(s) $\wedge\mathcal{O}$(f) $\wedge\mathcal{O}$(n) $\wedge\mathcal{O}$(id) $\wedge \cdots \rightarrow$ isCorrect(f, {"true"^^⟨ xsd:boolean⟩})
(7) CorrectTravelRequestForm $\doteq$ TravelRequestFrom $\sqcap \exists$ isCorrect.{"true"^^⟨ xsd:boolean⟩}
(8) InCorrectTravelRequestForm $\doteq$ TravelRequestFrom $\sqcap \exists$ isCorrect.{"false"^^⟨ xsd:boolean⟩}
(9) checkRequestForm $\sqsubseteq \exists$ hasInput.TravelRequestForm $\sqcap$ $\exists$hasOutput.(CorrectTravelRequestForm $\sqcup$ InCorrectTravelRequestForm)
(10) functional(isCorrect)
(11) ApprovedForm $\doteq$ TravelRequestForm $\sqcap \exists$ isApproved.{"true"^^⟨ xsd:boolean⟩}

Each state in a workflow is mapped to a task in the domain ontology by matching the same name (i.e. each task is a concept in the ontology). For example, state 2 in the workflow (Figure 4) is mapped to checkRequestForm in the domain ontology. When a task in a workflow is about to be executed by an agent, an instance of the corresponding task in the ontology is created by the agent (in real-time).

Similarly, each input (or output) variable from a workflow task maps to a concept in the ontology. For example, the variable "ApprovedForm" in the workflow (Figure 4) is mapped to the concept ApprovedForm (see axiom 11 above) in the domain ontology. Every time an agent assigns a value to an input (or output) from a workflow task, then a new instance is also created in the ontology, corresponding to the value of the variable.

The creation of instances in the ontology allows an agent to check if its next action complies with the constraints of the organisation. The agent who is about to execute a workflow task first tentatively creates an instance in the ontology, and then calls the DL reasoner to check the ontology's consistency and also to check if violations have been introduced. If the ontology is inconsistent, then the agent knows that the task it was about to execute is an error; on the other hand, if the ontology entails violated(x,task) for some agent "x", then the agent knows that the task it was about to execute would cause it to violate norms. Thus the agent should not carry out the task, and should revert to the ontology before the instance was added. This type of check can pick up on such things as an agent performing a prohibited action (see axiom 4 in subsection 3.2) or axiom (4)

above which forbids a manager from approving his own travel request. Of course an autonomous agent may choose to execute the task regardless, in which case the instance is added, and the ontology may now have recorded violations (in the case of broken prohibitions) or may be inconsistent (in the case of an agent updating wrong information). In the second case, the inconsistent ontology can still be used thereafter for agents to check proposed actions. Ontology reasoners can reason with inconsistent ontologies by selecting consistent subsets [8]. In our case, the reasoner can identify the set of "Minimal Unsatisfied Preserving Sub-Ontologies" (MUPSs) in an inconsistent ontology [17]; each MUPS is a minimal set of problematic axioms. Thus, given an inconsistent ontology, an agent can add an instance and check if the number of MUPSs has increased; if so, then this instance has caused further inconsistencies, otherwise the instance (and hence the proposed workflow task) is acceptable.

We now describe how the agent's behaviour is related to the ontology and the workflow using the "travel request" workflow in Figure 4. Let us look at the second state of the workflow in Figure 4, i.e., the "checkRequestForm" task. When control passes to this state there already exists an instance of a TravelRequestForm in the ontology, say this is TF124. Now the above axioms (1) states that the Secretary is obliged to perform the "checkRequestForm" workflow task. The secretary is going to perform this action, so the secretary agent creates an instance of checkRequestForm, say this is CRF54. Axiom (5) states that checkRequestForm is defined as having at least one checks relationship, and therefore the secretary agent must also add an ABox axiom for the relationship checks(CRF54, TF124); the agent can also deduce that the form TF124 should be correct or incorrect (i.e. boolean). Now due to axiom (6), assuming the form has been filled correctly, then its isCorrect property should have a true value; this corresponds to ABox axiom isCorrect(TF124, {"true"^^⟨ xsd:boolean⟩}), which can now be inferred from the ontology. Now from axiom (7) it can be inferred that the travel request form TF124 is an instance of the concept CorrectTravelRequestForm. Finally axiom (9) tells us that this instance TF124 is the value to be assigned to the output variable "CorrectTravelRequesForm" of this workflow task. However, the secretary might erroneously decide to assign the output TF124 as the output value "CorrectTravelRequesForm" or "IncorrectTravelRequesForm" in the workflow. As mentioned above, whenever an agent assigns a value to a workflow input (or output) variable, the agent must add an instance of the corresponding concept to the ontology. Thus the secretary's choice is between adding the ABox axiom CorrectTravelRequestForm(TF124) or InCorrectTravelRequestForm(TF124). Of course if the latter choice is made, then the ontology becomes inconsistent, because the form passed all the tests of axiom (6), hence CorrectTravelRequestForm(TF124) can already be inferred. Thus the secretary knows that declaring the form incorrect breaks the organisation's constraints. Likewise, if the form was incorrectly filled, the secretary would break the organisation's constraints by declaring it to be correct. Axiom (10) states that the isCorrect property can only have one value (i.e. the form's correctness cannot be both true and false).

7 Dealing with Exceptions

During the enactment of workflows, exceptions may occur easily, for example due to unavailable resources, or agent failures. One way to deal with exceptions

is to classify exceptions into classes and pre-define rules or policies to handle each case; specialised agents then perform defined remedial actions [9]. However, exceptions are difficult to predict during design, especially in open and dynamic environments. It is preferable to program agents with intelligence and adaptivity so they can accommodate unexpected changes in their environment. To satisfy this requirement we have associated the workflow tasks with semantic information in the OWL ontology, and we have also represented the background knowledge for the organisation. This allows agents to reason about the description of tasks and agents in the organisation and find alternative ways to deal with workflow tasks when exceptions arise.

Exceptions often involve the inability to execute some particular task in the workflow. This can be repaired by breaking off from the execution of the workflow at that point, and executing some sequence of actions which can act as a substitute for the problematic task. The appropriate sequence of actions may itself involve another workflow which is nested inside the original workflow. A simple example of this could be when a member of an organisation is absent and unable to execute a task in a workflow; then another member of the organisation may repair this problem by invoking a delegation workflow to delegate the absent member's duties to another suitable member of staff.

We provide exception handlers for the following two types of exception: (1) an agent exception, where an agent may have crashed, or is not performing for some reason; and (2) a task exception, where a task is unachievable.

The "Agent_Exception_handler" in Figure 6 handles Exceptions regarding unavailable agents. The input to this routine consists of the problematic workflow state and the agent who is handling the exception (Ag_H); this is the agent who completed the preceding workflow state, and was unable to pass control to the next agent. This routine first tries to find an alternative agent which has the appropriate capability (and institutional power if necessary), and to delegate the task to that agent. If no suitable agent can be found, then a substitute task is sought; the subroutine "Find_Alternative_Task" (Figure 8) tries to find a task with the same inputs and outputs. If no suitable task can be found, then the Agent_Exception_handler tries to procure additional staff, and this is done via a nested workflow for staff procurement. The agent uses its own internal procedure "callWorkflow" to initiate a workflow to procure a new member of staff who can do task T_p; this procedure returns true if the workflow successfully procures new staff. We do not detail the procurement workflow, but it will make a selection between either hiring contract staff or recruiting new staff, depending on the organisational rules governing that class of staff.

The "Task_Exception_handler" in Figure 6 deals with unachievable tasks. Its inputs are the problematic workflow state and the agent who is handling the exception (Ag_H); in this case this is the agent who attempted to execute the problematic task, and was unable to. The routine begins by checking if the task has failed due to the unavailability of a required resource. If so, an alternative resource is sought. This is done by finding siblings of the original resource in the ontology. This could for example replace a black and white laser printer with a colour laser printer for a simple print job; the colour printer is less desirable as it is more costly, but it can do the job. If no suitable substitute resource can be found, then a substitute task is sought; the subroutine "Find_Alternative_Task"

(Figure 8) tries to find a task with the same inputs and outputs. If no suitable task can be found, then the Task_Exception_handler tries to procure the resource, and this invokes a nested workflow for resource procurement. This workflow is shown in Figure 9, it will make a selection between either hiring the equipment or purchasing it, depending on the organisational rules governing that type of equipment (we do not give the details of these rules).

algorithm Agent_Exception_handler($\langle id, T_p \rangle, Ag_H$)
  // Try to find an alternative agent who has the capability to do T_p
  if Find_Alternative_Agent($\langle id, T_p \rangle, Ag_H, \mathcal{O}$) **return** true;
  // Try to find an alternative task (or workflow) with the same input/output as T_p
  else if Find_Alternative_Task($\langle id, T_p \rangle, Ag_H, \mathcal{O}$) **return** true;
  else if callWorkflow(procureStaff_WF,T_p) **return** true; // initiate the staff procurement workflow
  else return false;

algorithm Task_Exception_handler($\langle id, T_p \rangle, Ag_H$)
  // if something is missing try alternative resources
  if there exists some resource R such that
        $T_p \sqsubseteq \exists \mathsf{uses}.R$ and ! available (R)
    then for each R_i, where sibling(R, R_i)
      let r be an instance of R
      let r_i be an instance of R_i
      let t be an instance of T_p
      $\mathcal{O}' := \{\mathsf{uses}(t, r_i)\} \cup \mathcal{O} \setminus \{\mathsf{uses}(t, r)\}$
      if consistent($\mathcal{O}'$) then $\mathcal{O} := \mathcal{O}'$; **return** true;
    end for
  // Try to find an alternative task (or workflow) with the same input/output as T_p
  if Find_Alternative_Task($\langle id, T_p \rangle, Ag_H, \mathcal{O}$) **then return** true;
  // If resources are missing
  if there exists some resource R such that $T_p \sqsubseteq \exists$ uses.R and ! available (R)
    then if callWorkflow(procureResource_WF,R) **return** true
    // initiate the resource procurement workflow
  return false;

Fig. 6: Dealing with Exceptions

algorithm Find_Alternative_Agent($\langle id, T_p \rangle, Ag_H, \mathcal{O}$)
  $Ag_H = \langle \mathcal{R}_H, \mathcal{T}_H \rangle$
  // if T_p is a type of institutional task, then it needs institutional power
  if $T_p \sqsubseteq$ Institutional_Task $\in \mathcal{O}$
  // then find agents having the capability and power to do T_p
  then Agent_list := RunQuery("Type(_:y, ?agent), PropertyValue(_:x, org:isPermitted, _:z),
        PropertyValue(_:x, org:hasPower, _:z), PropertyValue(_:y, org:isCapable, _:z),
        PropertyValue(_:y, org:hasRole, _:x), Type(_:z,dom:T_p)")
  // else find agents having the capability to do T_p
  else Agent_list := RunQuery("Type(_:y, ?agent), PropertyValue(_:y, org:isCapable, _:z), Type(_:z,dom:T_p)")
  for each Ag_i in Agent_list do
    // if Ag_H has power to delegate to Ag_i, then Ag_H gives him the order to do it
    if $\exists Role \in \mathcal{R}_H$ such that canDelegate($Role, Ag_i$)
      then SpeechAct(Ag_H, Ag_i,order, T_p) **return** true;
    else // Ag_H requests Ag_i to take the obligation
      if callWorkflow(request_WF, Ag_H, Ag_i, T_p) **then return** true;
  end for
  return false;

Fig. 7: Finding Alternative Agents

We now describe dealing with exceptions with examples. We consider the workflow example shown in Figure 9 which deals with printing a finance report. When a finance report is written, its format is then checked. Next, the report is

algorithm Find_Alternative_Task($\langle id, T_p \rangle, Ag_H, \mathcal{O}$)
  $Ag_H = \langle \mathcal{R}_H, \mathcal{T}_H \rangle$
  // find any task T with same input/output
  Task_list := RunQuery("Type(_:t, ?task),Type(_:z, dom:T_p), PropertyValue(_:z, dom:hasInput, _:xi),
        PropertyValue(_:z, dom:hasOutput, _:xo), PropertyValue(_:t, dom:hasInput, _:xi),
        PropertyValue(_:t, dom:hasOutput, _:xo)")
  for each $T \in$ Task_list // check the consistency for each alternative task
    let t be an instance of T_p
    $\mathcal{O}' := \{T(t)\} \cup \mathcal{O} \setminus \{T_p(t)\}$
    if consistent($\mathcal{O}'$) then $\mathcal{O} := \mathcal{O}'$
      if T is a workflow, then Ag_H initiates T
      else if T is a task // then check if the Ag_H can do T
        if $T_p \sqsubseteq$ Institutional_Task $\in \mathcal{O}$ // then find agents having the capability and power to do T_p
          if $\exists Role \in \mathcal{R}_H$ such that isCapable(Ag_H, T)$\wedge$ hasPower($Role, T$)$\wedge$ isPermitted($Role, T$)
          then $Ag_H := \langle \mathcal{R}_H, \mathcal{T}_H \cup \{T\} \rangle$; **return** true;
        else if $\exists Role \in \mathcal{R}_H$ such that isCapable(Ag_H, T)
          then $Ag_H := \langle \mathcal{R}_H, \mathcal{T}_H \cup \{T\} \rangle$; **return** true;
        // otherwise call routine for finding alternative agent
        if Find_Alternative_Agent($\langle id, T \rangle, Ag_H, \mathcal{O}$) **then return** true;
  end for
  return false;

Fig. 8: Finding Alternative Task

to be approved, and printed locally, finally the report is posted. Below we list some of the ontology axioms which are relevant to this workflow.

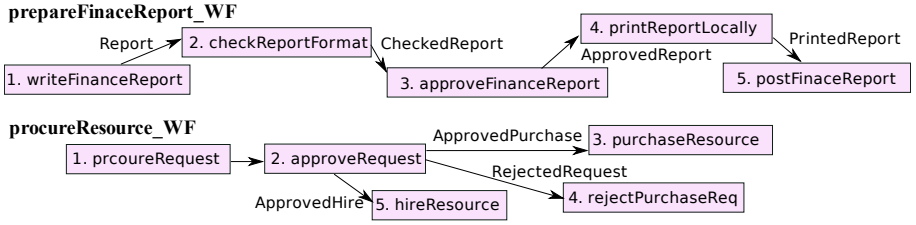

Fig. 9: Finance Report and Purchase Workflows

(1) ApproveReport $\sqsubseteq$ Institutional_Task
(2) Manager $\sqsubseteq \exists$ hasPower.ApproveReport $\sqcap \forall$ hasPower.ApproveReport $\sqcap \exists$ isCapbale.ApproveReport $\sqcap$ $\exists$ isPermitted.ApproveReport
(3) ITManager $\sqsubseteq \exists$ hasPower.ApproveITReport $\sqcap \forall$ hasPower.ApproveITReport
(4) FinanceManager $\sqsubseteq \exists$ hasPower.ApproveFinanceReport $\sqcap \forall$ hasPower.ApproveFinanceReport
(5) FinanceManager $\sqcup$ ITManager $\sqcup$ GeneralManager $\sqsubseteq$ Manager
(6) ApproveFinanceReport $\sqcup$ ApproveITReport $\sqsubseteq$ ApproveReport
(7) ApproveFinanceReport $\sqsubseteq \neg$ ApproveITReport
(8) Secretary $\sqsubseteq \exists$ isObliged.checkReportFormat

The FinanceManager has the power to do ApproveFinanceReport. If the FinanceManager is not available, then control returns to state 2 of the workflow, where the secretary must handle the exception using the routine in Figure 6. This leads to the routine "Find_Alternative_Agent" (in Figure 7) being run. After running the query within "Find_Alternative_Agent", the agent whose role is GeneralManager is returned as an appropriate candidate to carry out the approval task. This is because GeneralManager inherits power from its superclass Manager, while ITManager is restricted to approve IT reports only. The final stage of the exception handling is for the secretary to initiate a "request" workflow, to request that the general manager take on the obligation to approve the

finance report. If this is successful, then the workflow can resume with the new substitute agent.

The next step is to print the report. To show an example of an unachievable task, let us assume no printer or toner is available; therefore printReportLocally cannot be implemented. As the secretary is responsible for this unachievable task, the secretary must execute of the routine "Task_Exception_handler" (in Figure 6), which reveals that a resource is missing, and no suitable alternative resource exists in the organisation. The routine then searches for alternative tasks and finds that the task printReportCommercially is a sibling task of printReportLocally, has the same input ApprovedReport and output PrintedReport. However, if the report is sensitive, it is not allowed to print it commercially (see axiom 6 below). Assume that printRpt322 is the instance of printReportLocally, which is to be replaced by printReportCommercially; according to "Find_Alternative_Task" (in Figure 8) we move the instance printRpt322 from printReportLocally to printReportCommercially. If the report is sensitive, i.e., printRpt322 is also an instance of SensitiveReport, then from axiom 6 below we could infer that the agent performing the commercial print act is violating norms. Hence, printReportCommercially should not be executed. If the agent chooses not to violate the norms, then "Find_Alternative_Task" fails and control returns to "Task_Exception_handler" (in Figure 6). Having failed to find an alternative resource or task, this routine now tries to procure the resource required for the task. This means that the secretary must initiate the "procureResource_WF" workflow (in Figure 9), and if successful, the printing resource is available, and the workflow can progress.

(1) printReportLocally $\doteq$ $\exists$ print.(Report $\sqcap$ $\exists$ printedBy.(Printer $\sqcap$ $\exists$ hasToner.(Toner $\sqcap$ $\exists$ hasAmt.GreaterThanZero)))
(2) Printer $\sqcup$ Toner $\sqsubseteq$ Resource
(3) procureRequest $\doteq$ $\exists$ requests.(Resource $\sqcap$ $\exists$ hasAmt.LessThanOne $\sqcap$ $=$ 1 hasPrice)
(4) printReport $\sqsubseteq$ $\exists$ hasInput.ApprovedReport $\sqcap$ $\exists$ hasOutput.PrintedReport
(5) printReportLocally $\sqcup$ printReportCommercially $\sqsubseteq$ printReport
(6) Staff(s) $\wedge$ SensitiveReport(r) $\wedge$ printReportCommercially(act) $\wedge$ print(act,r) $\wedge$ performed(s,act) $\wedge\mathcal{O}$(s) $\wedge\mathcal{O}$(r) $\wedge\mathcal{O}(act)$ $\rightarrow$ violated(s,act)
(7) Secretary $\sqsubseteq$ $\exists$ isObliged.printReportLocally

8 Related Work

Various works use agents to enact workflows. Buhler and Vidal [4] 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 and the Web services it calls. Thus the workflow execution is managed centrally rather than by the agents. On the other hand Guo et al. [5] 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). This work however does not address organisational or normative aspects of an agent system; we believe that these high level aspects are important for agent systems that are to model real processes in human organisations; such simulations can be useful to reveal potential problems in organisational and normative specifications for a system, for example in a crisis management scenario. Furthermore we have shown how the use of ontologies to describe aspects of the organisation and domain can be valuable in exception handling, as agents are part of an organisation and will be unable to deal with exceptions entirely on their own.

Klein and Dellarocas [9] explicitly deal with the issue of exceptions; they propose the use of specialised agents that handle exceptions. The exception handling service is a centralised approach, in which a coordination doctor diagnoses agents' illnesses and prescribes specific treatment procedures. Klein and Dellarocas [10] identified an exception taxonomy which is a hierarchy of exception types, and then described which handlers should be used for what exceptions. Klein et al. [11] describe a domain-independent but protocol-specific exception handling services approach to increasing robustness in open agent systems. They focus on "agent death" in the Contract Net protocol. We would argue that our approach is more generic in that it is neither domain-specific, nor protocol-specific. When seeking alternative ways to achieve workflow tasks, our agents can use the same handling routine regardless of the workflow in progress. A further distinction between our work and the above related 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 endow the agents of the system themselves with the ability to deal with exceptions by querying ontologies to find alternative ways.

Perhaps the closest approach to our work in the literature is from Mallya and Singh [14]. Building on the commitment approach, they 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. This is similar to the way our exception handling can sometimes include a nested workflow in place of a failed task, to repair the workflow. However, they do not describe how these sets of runs can be created, but it is likely that one would need access to observed sequences from previous 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 stage. 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 norms of the other roles so that they can find suitable candidates to execute tasks in the case of exceptions.

9 Conclusions & Future Work

In this paper we have described a method by which an agent system could be constructed to enact a set of given workflows, while respecting the constraints of a given organisation. We have shown how Semantic Web languages can be used to describe the organisational knowledge, as well as domain knowledge which can be used by the agents if exceptions arise during the enactment of a workflow; the agents can then use this knowledge to make intelligent decisions about how to find alternative ways to complete the workflow.

Some issues which have not been addressed include the updating of the organisational knowledge by agent activities outwith the workflows, for example speech acts that may change norms in the system. Also, we have not included any mechanism to detect when an obligation is violated. This could be addressed

by associating time limits with obligations and including timer events which are triggered when obligations time out, and then checking if they have been fulfilled; this remains for future work.

Our aim has been to allow agents to deal with unexpected exceptions, rather than coding specific exception handlers for a predefined set of expected exceptions. Nevertheless we have had to define exception handling routines for some predefined situations, such as agent exceptions, or task exceptions. However, our predefined situations cover a broad class of exceptions, and there can be many possible solutions if the ontological knowledge is suitably rich.

References

1. IBM, BEA Systems, Microsoft, SAP AG and Siebel Systems, business Process Execution Language for Web Services version 1.1. Technical report, July 2003.
2. A. Artikis. *Executable Specification of Open Norm-Governed Computational Systems*. PhD thesis, Imperial College London, 2003.
3. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
4. 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.
5. L. Guo, D. Robertson, and Y.-H. Chen-Burger. Using multi-agent platform for pure decentralised business workflows. *Web Int. and Agent Systems*, 6(3), 2008.
6. I. Horrocks and P. F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. In *Proc. of the 2nd International Semantic Web Conference (ISWC 2003)*, number 2870, pages 17–29. Springer, 2003.
7. I. Horrocks and U. Sattler. A tableaux decision procedure for $\mathcal{SHOIQ}$. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence*, pages 448–453, 2005.
8. Z. Huang, F. van Harmelen, and A. ten Teije. Reasoning with inconsistent ontologies. In Kaelbling and Saffiotti, editors, *IJCAI'05*, 2005.
9. M. Klein and C. Dellarocas. Exception handling in agent systems. In *AGENTS '99: 3rd Annual Conference on Autonomous Agents*, pages 62–68, 1999.
10. M. Klein and C. 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.
11. 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. *Auton. Agents and Multi-Agent Systems*, 7(1-2):179–189, 2003.
12. A. Lanzén and T. Oinn. The taverna interaction service: enabling manual interaction in workflows. *Bioinformatics*, 24(8):1118–1120, 2008.
13. 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.
14. 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.
15. B. Motik, U. Sattler, and R. Studer. Query Answering for OWL-DL with Rules. In *Proc. of the 3rd Int. Semantic Web Conf.*, pages 549–563. Springer, 2004.
16. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.
17. S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 355–362, 2003.

18. E. Sirin and B. Parsia. SPARQL-DL: SPARQL query for OWL-DL. In *3rd OWL Experiences and Directions Workshop (OWLED-2007)*, 2007.
19. W. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
20. G. H. von. Wright. Deontic logic. *Mind, New Series*, 60(237):1–15, January 1951.
21. WfMC. Workflow management coalition terminology and glosary. Technical Report WFMC-TC-1011, Workflow Management Coalition, 1999.