An Ontological Framework for Dynamic Coordination

Valentina Tamma[1], Chris van Aart[2], Thierry Moyaux[1],
Shamimabi Paurobally[1], Ben Lithgow-Smith[1], and Michael Wooldridge[1]

[1]Dept of Computer Science
University of Liverpool
Liverpool L69 7ZF, UK

[2]Acklin BV
Taxandriaweg 12b
5142 PA Waalwijk, The Netherlands

Abstract. Coordination is the process of managing the possible interactions between activities and processes; a mechanism to handle such interactions is known as a coordination regime. A successful coordination regime will prevent negative interactions occurring (e.g., by preventing two processes from simultaneously accessing a non-shareable resource), and wherever possible will facilitate positive interactions (e.g., by ensuring that activities are not needlessly duplicated). We start from the premise that effective coordination mechanisms require the sharing of knowledge about activities, resources and their properties, and hence, that in a heterogeneous environment, an ontological approach to coordination is appropriate. After surveying recent work on dynamic coordination, we describe an ontology for coordination that we have developed with the goal of coordinating semantic web processes. We then present a implementation of our ideas, which serves as a proof of concept for how this ontology can be used for dynamic coordination. We conclude with a summary of the presented work, illustrate its relation to the Semantic Web, and provide insights into future extensions.

1 Introduction

Coordination is one of the fundamental problems in systems composed of multiple interacting processes. Such processes will need to coordinate their activities if ever there is a possibility that these activities may interact with one-another. As an example, imagine two processes making use of a non-shareable resource. If both processes attempt to use the resource simultaneously, we will naturally have problems - a lost update at best, perhaps damage to the resource at worst. The processes thus need to *coordinate* their activities, to make use of the non-shareable resource. Although such a scenario represents the best-known type of possible coordination interaction, there are many other less obvious ways in which coordination may be mutually beneficial. For example, imagine two e-science processes carrying out some computational task, where both processes require the results of some intermediate computation; then, it makes sense for them to adopt a policy of pro-actively exchanging information that may be of use to other processes. Here, coordination is not *required* for the agents to be successful in their tasks, but there is a global benefit to be gained by adopting this rule.

Coordination in the limited sense of synchronisation (preventing scenarios such as simultaneous access to a non-shareable resource) has long been a central topic of research in the concurrency community [1]. However, the pre-dominant approach to

handling coordination has been to *hard-wire* the coordination mechanism into the system structure (for example by means of semaphores, monitors, or locks). In more open systems, where the processes and resources of which the system is comprised are not known at design time, such an approach is often impossible. In such systems, it may be desirable to allow the relevant processes to communicate their intentions with respect to future activities and resource utilisation, and get them to *reason* about coordination at run time, with the goal of preventing negative interactions, and facilitating positive interactions. This is a *dynamic* approach to coordination, since the coordination requirement is handled at *run-time*, rather than design time. Note that the communication implied by this approach requires an agreed common vocabulary for coordination, with a precise semantics, and hence we have an ontological approach to dynamic coordination, in short.

Our goal in this paper is to describe such an ontological approach to coordination, and present our results with respect to a proof-of-concept implementation of the approach. We begin in the following section with a brief survey of previous work on coordination, which has been carried out largely within the multi-agent systems community. In section 3, we give an informal overview of our coordination ontology; the key concepts, their attributes, and their relationships. In section 4, we present a proof-of-concept implementation of the ontological approach to coordination, in which multiple processes detect coordination relationships using a Jess/Protégé implementation of the ontology. We conclude with some conclusions and pointers to further work.

2 Background

Coordination is perhaps the defining problem in cooperative working. Since much work on coordination (and in particular, the precursors to our own work) arises from the multi-agent systems community [2], we will adopt the convention of referring to the processes which need to coordinate as "agents". The coordination problem is that of *managing relationships between the activities of agents* [3]. Coordination is essential if the activities that agents engage in can *interact* in any way. Consider the following examples.

- *You and I both want to leave the room, and so we independently walk towards the door, which can only fit one of us. I graciously permit you to leave first.* In this example, our activities need to be coordinated because there is a resource (the door) which we both wish to use, but which can only be used by one person at a time.
- *I intend to submit a grant proposal, but in order to do this, I need your signature.* In this case, my activity of sending a grant proposal depends upon your activity of signing it off – I cannot carry out my activity until yours is completed. In other words, my activity *depends* upon yours.
- *I obtain a soft copy of a paper from a Web page. I know that this report will be of interest to you as well. Knowing this, I pro-actively photocopy the report, and give you a copy.* In this case, our activities do not strictly need to be coordinated – since the report is freely available on a Web page, you could download and print your own copy. But, by pro-actively printing a copy, I save you time.

Notice that coordination, defined in this way, subsumes the well-known (and widely studied) concept of *synchronisation* [1]. Synchronisation is generally concerned with the rather restricted case of ensuring that processes do not destructively interact with one another. While solving this problem certainly requires coordination, the concept of coordination is actually much broader than this. Standard solutions to synchronisation problems involve *hard-wiring* coordination regimes into program code. Thus, for example, a JAVA method may be flagged as `synchronized` by a programmer, indicating that a certain access regime is enforced whenever this method is invoked. However, in large-scale, dynamic, open systems, of the kind we are concerned within this project, such hard-wired regimes are too limiting. We ideally want computational processes to be able to *reason about* the coordination issues in their system, and resolve these issues *autonomously*.

In order to build agents for semantic web applications that can reason about coordination issues dynamically, we must first identify the possible interaction relationships that may exist in these applications. Hence, the goal, here, is to derive and formally define the possible interaction relationships that may exist between activities. There is some prior work on this topic — von Martial [4] puts forward a high-level typology for coordination relationships. He suggested that, broadly, relationships between activities could be either *positive* or *negative*. Positive relationships "are all those relationships between two plans from which some benefit can be derived, for one or both of the agents plans, by combining them" [5, p. 111]. Such relationships may be *requested* (I *explicitly* ask you for help with my activities) or *non-requested* (it so happens that by working together we can achieve a solution that is better for at least one of us, without making the other any worse off). Von Martial distinguishes three types of non-requested relationships:

The action equality relationship: We both plan to perform an identical action, and by recognizing this, one of us can perform the action alone, and so, save the other effort.

The consequence relationship: The actions in my plan have the side-effect of achieving one of your goals, relieving thus you of the need to explicitly achieve it.

The favour relationship: Some part of my plan has the side effect of contributing to the achievement of one of your goals, perhaps by making it easier (e.g., by achieving a precondition of one of the actions in it).

Another major body of work on this issue is that on *Partial Global Planning* [6]. The basic idea of partial global planning is that agents develop and exchange plans of local activity in order to identify possible interactions (positive or negative). The ideas were refined in Decker's subsequent work on *Generalised Partial Global Planning* (GPGP) in the TÆMS testbed [7]. GPGP makes use of five techniques for coordinating activities:

– *Updating non-local viewpoints*: Agents have only local views of activities, and so, sharing information can help them achieve broader views. In his TÆMS system, Decker uses three variations of this policy: communicate no local information, communicate all information, or an intermediate level.

– *Communicate results*: Agents may communicate results in three different ways. A minimal approach is where agents only communicate results that are essential to satisfy obligations. Another approach involves sending all results. A third is to send results to those with an interest in them.
– *Handling simple redundancy*: Redundancy occurs when efforts are duplicated. This may be deliberate – an agent may get more than one agent to work on a task because it wants to ensure the task gets done. However, in general, redundancies indicate wasted resources, and are therefore to be avoided. The solution adopted in GPGP is as follows. When redundancy is detected, in the form of multiple agents working on identical tasks, one agent is selected at random to carry out the task. The results are then broadcast to other interested agents.
– *Handling hard coordination relationships*: "Hard" coordination relationships are essentially the "negative" relationships of von Martial. Hard coordination relationships are thus those that threaten to prevent activities being successfully completed. Thus a hard relationship occurs when there is a danger of the agents' actions destructively interfering with one another, or preventing each others actions being carried out. When such relationships are encountered, the activities of agents are rescheduled to resolve the problem.
– *Handling soft coordination relationships*: "Soft" coordination relationships include the "positive" relationships of von Martial. Thus, these relationships include those that are not "mission critical", but which may improve overall performance. When these are encountered, then rescheduling takes place, but with a high degree of "negotiability": if rescheduling is not found possible, then the system does not worry about it too much.

Based on all this body of work, we have designed an ontology for coordination, which is presented in the next section. Although ontologies for service based computing have been developed, such as OWL-S [8] and WSMO [9], they mainly focus on describing the services and their orchestration/composition. We argue that our ontology is complementary to existing efforts. Coordination is indeed an important aspect of service based computing, however it addresses the way in which *independent*, and possibly conflicting agents choreograph with others. While in efforts like OWL-S and WSMO the interaction and composition of processes are modelled as a workflow that is determined *a priory* and that is executed by a workflow execution component, in agent-based coordination, the choreography is determined by the exchange of messages among the agents that need to interact (*protocol*). However, OWL-S first order logic representation of process theory based on PSL [18] could be integrated in our ontology, in a future implementation.

3 An Ontology for Coordination

As described above, we define an ontology for coordination. The basic idea is to enable agents to reason about the relationships of their activities to the activities of other agents. So, the fundamental purpose of the ontology is to answer the following questions:

– what is a *coordinable activity*?

– what *coordination relationships* such activities have to one another?

In the sub-sections that follow, we give an overview of the ontology: the key concepts, the slots associated with these concepts, the relationships between these concepts, and axioms. In the interests of comprehensibility, we do not present all the components of the ontology. Also note that our presentation is informal: we aim to give an overview of the ontology, rather than present all the low-level technical details. The "definitive" version of the ontology is maintained using Protégé [10] and is illustrated in in Figure 1.

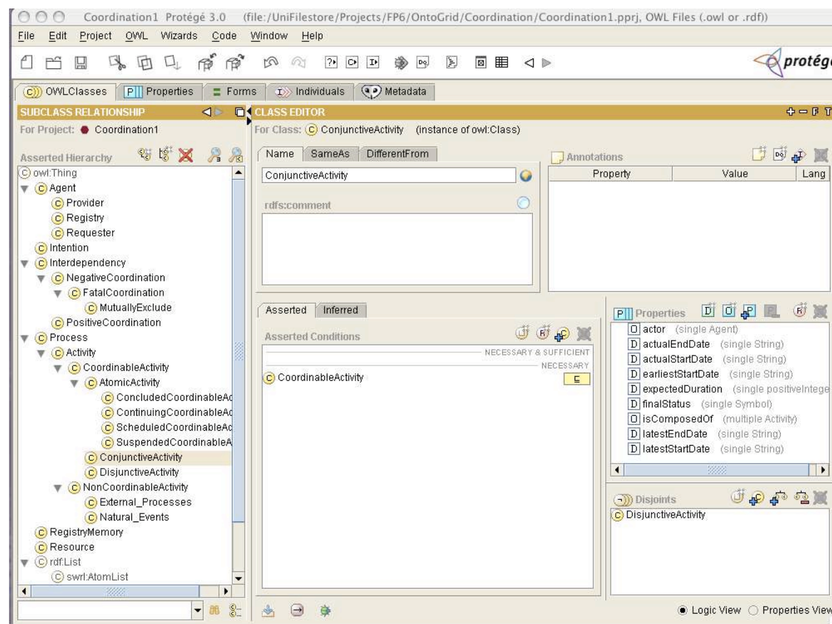

Fig. 1. The Protégé version of the coordination ontology

3.1 Agents

Our starting concept is *Agent*. The idea is, obviously enough, that this concept relates to the agents in the system, i.e., the things that do the actions in the system needing to be coordinated. For the purposes of the coordination ontology, agents have just one slot: *id*, which is a string representation of the unique identifier for the agent (e.g., a URI).

3.2 Processes and Activities

Our next concept is *Process*. A process is an activity that changes the state of the environment in some way. It may be terminating or non-terminating, and be carried out by a human or other agent, or be a natural (physical) process.

The process concept has two sub-classes: the most important of which is that of a *CoordinableActivity*. A coordinable activity is a process that can be managed in such a way as to be coordinated with other coordinable activities. For example, executing the process of invoking a web service would be a coordinable activity, in the sense that the invocation of such a service can be managed so as to coordinate with other invocations. For example, suppose we have two agents, both of which want to invoke the same web service, with different parameters. Then, in general, the agents could manage their invocations so as not to interfere with one another.

Not all processes of interest to a system are coordinable – hence we have the *NonCoordinableActivity* concept. We intend this concept to capture all those processes whose coordination is not possible by the agents within the system to which a particular knowledge base refers. This will include at least the following two types of process (although we do not represent these as concepts):

– *Natural events*: These are physical processes that will take place irrespective of what any agent in the system does. An extreme example would be the decay of an atom, caused by essentially random quantum events. Clearly, such processes cannot be coordinated with other processes: they will take place (or not take place) irrespective of what the agents in the system do.
– *External processes*: These are processes – either physical world processes or natural processes – which are simply outside the control of the system, in that they cannot be managed by the agents in the system. Notice that such processes may be coordinated by entities *outside* the system: the point is, that for the purposes of the system to which the knowledge base refers, they cannot be coordinated.

Another way of thinking about the distinction between a coordinable and a non-coordinable activity is that there is always an agent (i.e., a software agent within the system) associated with a coordinable activity, whereas there is no such agent associated with a non-coordinable activity.

We think of particular *CoordinableActivity* as being arranged into an and/or tree hierarchy of activities, with *AtomicActivity*s as leaves of the tree. Thus a *CoordinableActivity* is *composedOf* possibly many other *Activity*s, and may be:

– a *ConjunctiveActivity*: in this case, it is composed of a number of other activities, which must *all* be successfully completed in order for the overall activity to be completed;
– *DisjunctiveActivity*: it is composed of a number of other activities, of which *at least one* must be successfully completed in order for the overall activity to be completed; or
– *AtomicActivity* – in which case the activity is composed of *no* further activities. (The set of *CoordinableActivity*s of which this activity is composed is empty.)

In future work, it may be interesting to compare these notions with those of the OWL-S model of processes, and one possibility is to attempt to align them in some way [8][1]. We can further identify the following sub-classes of *AtomicActivity*:

[1] The point is that there may be some relation at this point to the process model in OWL-S, so perhaps an *AtomicActivity* is a sub-class of an OWL-S process, and similarly for OWL-S composite processes.

– *ConcludedCoordinableActivity*: an activity that has taken place in the past, and is now fully concluded;
– *ContinuingCoordinableActivity*: this is an activity that is currently in progress;
– *ScheduledCoordinableActivity*: this is an activity that it is expected *will* take place, in the sense that it is scheduled for execution by some agent[2];
– *SuspendedCoordinableActivity*: this is an activity that whose status is undetermined.

Let us briefly consider slots and properties of our concepts. A *CoordinableActivity* will have the following slots:

– *actor*: an *Agent*, i.e., the agent that intends to carry out, or has carried out this activity;
– *earliest start date*: either a date or `null`, with a date indicating the earliest date at which the activity may begin; `null` indicates that this information is not known;
– *latest start date*: either a date or `null`, with a date indicating the latest date at which the activity may begin; `null` indicates that this information is not known;
– *expected duration*: either a natural number, indicating the number of milliseconds the activity is expected to take, or `null` indicates an unknown duration;
– *latest end date*: either a date or `null`, with a date indicating the latest date at which the activity may end; `null` indicates that this information is not known;
– *actual start date*: either a date or `null`, with a date indicating the date at which the activity actually began; `null` indicates that this information is not known;
– *actual end date*: either a date or `null`, with a date indicating the date at which the activity actually ended; `null` indicates that this information is not known;
– *final status*: an enumeration type, either `succeeded`, `failed`, or `null`.

There are a number of axioms that may be introduced at this point. With respect to *Conjunctive* and *DisjunctiveActivity*s, we have the following:

– a *ConjunctiveActivity* has successfully terminated if all its components have successfully terminated;
– a *DisjunctiveActvity* has successfully terminated if at least one of its components has successfully terminated.

With respect to the relationship between scheduled activities and their successful completion, we have the following:

– if an activity is scheduled, then it should have a `null` actual start date and actual end date.
– if an activity is concluded, then the final status must be non-`null`;
– if an activity started before its earliest start date, then it has `failed`;
– if an activity started after its latest start date, then it has `failed`.

[2] We do not worry about exactly what "scheduled for execution" means: we simply assume that some agent is expected to carry out the activity, or that the activity appears in some agent's plan.

3.3 Resources

Next, we have the *Resource* concept. The idea of this concept, as we discussed in the introduction, is that a resource is something that may be required to expedite an activity. Thus, we have a one-to-many relationship between *AtomicActivity*s and *Resource*s. Note that we regard this set as being fixed, for any given activity. The *Resource* concept has the following slots:

– *viable*: a Boolean value, indicating whether the resource is still in a state to be used; a value of `false` here would indicate that the resource could not be used by any activity (even if these activities *Require* it). Another simple way to think about *viable* is that it indicates whether a resource is "broken" or "working" or not.
– *consumable*: a Boolean value, which indicates whether the use of the resource will reduce subsequent availability of the resource in some way; more precisely, whether the repeated use of the resource in activities would make the resource non-viable.
– *shareable*: a Boolean value, indicating whether a resource may be used by more than one agent at any given time.
– *cloneable*: a Boolean value, indicating whether or not the resource is cloneable (= `true`), or unique and not-cloneable (= `false`). An example of a cloneable resource would be a dataset or a digital document. An example of a unique resource would be a physical artefact produced as the output of a particular experiment, or a human being.
– *owner*: either an *Agent* (in which case this is the agent that owns the resource), or `null` (in which case the semantics are that the resource may be used by any agent at no cost). If a resource is owned by an agent, and another agent wishes to use this resource, then it may be necessary to enter into negotiation over the exploitation of the resource.

3.4 Interdependencies Between Activities

We now turn to the interrelationships that exist between activities. Our first concept is that of an *Interdependency*. The interdependency concept has the following slots:

– *source* and *target*: both slots are *Activities*, the idea being that these are the two activities which are interdependent.
– *isHard*: a Boolean value, which indicates whether the relation is "soft" (= `false`) or "hard" (= `true`), with the following semantics:
  • a *hard* relation is one which will materially affect the success or otherwise of the activities;
  • a *soft* relation is one which *may* affect the activities, positively or negatively, but will not affect whether they are successful or not.

Sub-classes of *CoordinationRelation* are:

– *NegativeCoordination*: an interaction which, if it occurs, will lead to a reduction in the quality of the solution or the utility of the participants;

- *PositiveCoordination*: an interaction which, if it occurs, will lead to an increase in the utility of the participants or the quality of the solution.

We have a further sub-class of *NegativeCoordination*: *FatalCoordination* is a hard coordination relationship which, if it occurs, will inevitably lead to the failure of one or more of the component activities. Note that instances of *FatalCoordination* relationships are always *hard*. As sub-classes of *FatalCoordination*, we have:

- *MutuallyExclude*: an instance of this relationship will exist between two *AtomicActivity*s iff:
  1. they both *Require* some resource *r*,
  2. the actual or scheduled usage of *r* by both activities overlaps;
  3. *r* is non-shareable.

  The idea is thus that these two activities will be mutually exclusive, in the sense that they cannot possibly both succeed, as they require access to a resource that cannot be shared.
- *ResourceContention*: an instance of this relationship will exist between two *AtomicActivity*s iff:
  1. they both *Require* some resource *r*;
  2. resource *r* is consumable.

  The idea here is thus that one of the activities (the earlier one) could prevent the successful completion of the other activity, by depleting it or rendering it unviable. We do not require that *ResourceContention* relationships are *hard*,although, of course, they could be.
- *Disables*: one activity will disable another if the occurrence of it will definitively prevent the occurrence of the other.

Sub-classes of *PositiveCoordination* are:

- *ConditionallyFeeds*: in such a coordination, the occurrence of activity A_1 will subsequently make possible the occurrence of activity A_2, but it is nevertheless possible that A_2 could occur (i.e., the occurrence of A_1 is a sufficient but not necessary event for the occurrence of A_2);
- *Enables*: the occurrence of activity A_1 is both necessary and sufficient for the occurrence of A_2;
- *IsSubsumedBy*: activity A_1 is subsumed by activity A_2 if A_2 contains all the activities of A_1.;
- *Subsumes*: the inverse of *IsSubsumedBy*;
- *Favors*: an activity A_1 favors another activity A_2 if its prior occurrence will subsequently improve the overall quality of A_2. We include this as a “catch all”. This is a *soft* relationship.

3.5 Operational Relationships

In order to *resolve* a coordination relationship between two activities, we may have to appeal to the *operational relationships* that exist between the agents that will carry

them out. Intuitively, operational relationships exist between agents that carry out activities, and by understanding these relationships, it can help to resolve the coordination relationships. The main concept here is *OperationalRelationship*. This concept has two slots, both of which are *Agents*: *source* and *target*. Sub-classes of *OperationalRelationship* include:

- *LegalAuthority*: this sub-class indicates that *source* has legal authority over *target* (of course, this begs the question of what "legal authority" means in the context of semantic web services and processes, but this is outside the scope of our current work, and is left as a placeholder for the future);
- *ContractualAuthority*: this indicates that *source* has contractual authority over *target* (i.e., that both agents "belong" to the same organisation, and that in the context of this organisation, *source* should take precedence over *target*);
- *ProducerConsumer*: this indicates that *source* is the *owner* of a *Resource* that is to be used by *target*;
- *ConsumerProducer*: the inverse of *ProducerConsumer*;
- *Peer*: two agents that work as peers, i.e., that neither has any authority over the other. We have developed a prototype as a proof-of-concept for our ontology. The current state of its development is now presented.

4 Implementation

We have implemented our prototype with the plug-in JessTab 1.1 [11] in Protégé 3.0. JessTab is a plug-in integrating the inference engine Jess (in its version 6.1p7 [12] in our case) with Protégé, so that Jess can carry out inferences on the knowledge base in Protégé. More precisely, JessTab enables Jess to work with a Protégé knowledge base, i.e., Jess can (i) access the ontology and the instances represented in Protégé, (ii) directly manipulate these ontology and instances, (iii) infer new facts deduced from these ontology and instances, and (iv) perform all the other programming tasks permitted by Jess, such as calculating or launching Java operations.

In our prototype, we use these capabilities of JessTab in the following way. We first design an ontology for our agents in OWL [13] using Protégé. For this proof of concept we restrict our attention to few concepts and types of coordination and we do not implement the whole ontology described in Section 3. In our implementation, concepts in the ontology are translated into Jess facts, whilst the coordination strategy is translated into a set of Jess rules. In our ontology , we create the class *Agent*, with subclasses *Provider*, *Requester* and *Registry*, as well as the classes required by these three types of *Agent*s, i.e., *RegistryMemory*, *Intention* and *Resource*, which are now outlined.

A *RegistryMemory* is related to an instance of *Registry* by the property "hasMemory". Every instance of *RegistryMemory* represents either a *Requester* and one of its *Intention*s, or a *Provider* and one of its capabities and associated *Resource*s.

The second element, *Intention*, is related to instances of *Agent* to describe one of the activities planned by a particular *Agent*. Besides an *Agent*, an *Intention* may also be linked to a *RegistryMemory* to enable a *Registry* to memorize an *Agent*'s *Intention*.

The third class is *Resource*, which contains the name of a resource (we assume this name is a unique identifier for this resource) and the flag “isShareable”.

After the creation of the classes of this ontology, we populate the ontology with instances. In our example, we instantiate one resource *Provider*, two resource *Requesters* and one *Registry*. These first two steps related to ontology building do not require JessTab, but only Protégé. Finally, we add Jess rules to “animate” our instances. These rules implement the choreography between the instanciated *Agent*s. These three stages are detailed in the following subsections.

4.1 Classes in the ontology

As noted, we basically deal with three different classes, namely *Provider, Requester* and *Registry*. Each time a *Provider* or a *Requester* registers to a *Registry*, this *Registry* records the information sent by this *Provider/Requester* by creating a *RegistryMemory*, which means that a *RegistryMemory* is very similar to a *Provider/Requester* and thus to an *Agent* (of course, only from an ontological viewpoint, like everything in this subsection!). To record a *RegistryMemory*, *Registry* has an object property called “hasMemory” listing instances of *RegistryMemory* used by this *Registry*. “hasMemory” is the only property of interest in a semantic *Registry*, even if a *Registry* inherits all the properties of an *Agent*, namely “hasName”, “hasCapabilities”, “hasGoals”, “hasIntention” and “hasResource”.

It is worth noting the difference we make between “hasGoals” and “hasIntention”: in a similar way to the BDI (Belief, Desire and Intention) architecture [14], an agent desires to achieve its multiple goals, and as a result, this agent selects and adopts the appropriate intention (which is a plan of actions in the BDI architecture). In our ontology, we translate this in the following way: an *Agent* has a property “hasGoals” pointing to several instances of *Intention*, and one property “hasIntention” pointing to one of these instances of *Intention*. This latter *Intention* represents what current action this *Agent* currently tries to achieve. Indeed, this is the main difference between a *RegistryMemory* and an *Agent*: only an *Agent* has a property “hasIntention”, while both *RegistryMemory* and *Agent* have a property “hasGoals”. A semantic *Registry* uses the property “hasGoals” to register in one of its *RegistryMemory*s what it knows about an *Agent*’s planned activities. In other words, there is one *RegistryMemory* per *Agent* (normally, this *Agent* should be a *Requester*), and as many properties “hasGoals” per *RegistryMemory* as the *Agent* communicates to the *Registry*.

Finally, all *Agent*s may have object properties “hasResource” of type *Resource*, and “hasCapabilities” of type *string*. As previously stated, there is one *RegistryMemory* per *Agent* (this *Agent* should now be a *Provider*), and as many properties “hasResource” and “hasCapabilities” per *RegistryMemory* as the *Agent* communicates to the *Registry*.

A *Resource* describes a resource, such as a CPU, a hard drive, a printer, . . . and datatype properties “hasCapabilities” of type string, e.g., saving-information, calculating, printing, etc. In addition, *RegistryMemory* has two datatype properties “agentName” and “agentRef” to respectively record the name (which is the string an *Agent* records in its datatype property “hasName”) and the address of the agent.

4.2 An example of instances for our ontology

As a case study, we have implemented a system with four agents, in which "Requester 1" and "Requester 2" look for the non-shareable resource called "Printer", while "Provider Printer" manages this resource. Requester 1 has "intention1", which describes the fact that this agent has scheduled to use Printer from the date 5 and for a duration of 10 time units. Figure 2 displays this Requester 1's intention to use Printer, as well as Requester 2's.

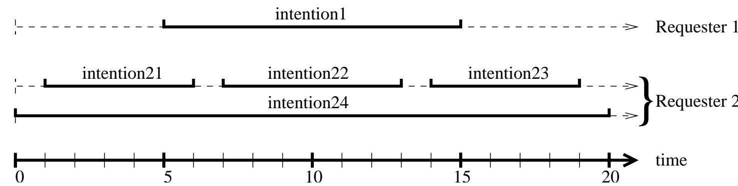

Fig. 2. Gantt chart of Requester 1 and Requester 2's schedules for Printer.

In this figure, we can also see that the property "hasGoals" of Requester 2 points to four intentions, namely intention21, 22, 23 and 24, and that intention24 is at the same time as intention21, 22 and 23. We assume that Agent2 has not seen this overlapping in its own schedule, and the registry should thus detect this clash among Agent2's goals. The registry should also detect the clashes between Agent2's goals and Agent1's.

4.3 Orchestration implemented in the prototype

The Jess program roughly adopts the following four steps. By "roughly", we mean that these steps are interlaced in practice, while we are now presenting them sequentially:

- *Step 1*: The Protégé classes, instances and templates are translated into Jess. This is performed by the JessTab command `(mapclass :THING)` that translates the root node of the Jess ontology, as well as all its children up to the instances, into Jess. Note that this command is an addition of JessTab to Jess.
- *Step 2*: Every agent sends (i.e., asserts) a registration message to every registry. This message contains the description of this agent, one of its capabilities, one of its goals and one of its resources. The agent sends several messages to register all its capabilities, goals and resources, and can write "none" if it does not have one of these features.
- *Step 3*: Every registry receives these messages and saves their content by creating *RegistryMemory*s. One *RegistryMemory* is created for each registering *Agent*, and this *RegistryMemory* is almost a copy of *Agent* reconstructed from the registration messages.
  In practice, the JessTab commands `make-instance` and `slot-set` add instances and slots in the Protégé base, and then `mapinstance` converts this information into Jess, so as the consistency is maintained between Jess and Protégé knowledge bases.

– *Step 4*: Every semantic *Registry* detects clashes between non-shareable resources. Intention21, 22, 23 and 24 in Figure 2 represent the four possible types of clash with intention1. For example, intention1/intention21 is a conflict in which the beginning of the time interval represented in intention1 overlaps the end of intention21. This conflict is characterized by the following conjunction: (i) the starting date requested by Requester 1 is later (greater) than the starting date requested by Requester 2, (ii) the starting date requested by Requester 1 is earlier (lower) than the ending date requested by Requester 2, (iii) the ending date requested by Requester 1 is later (greater) than the the ending date requested by Requester 2. Notice that (i) and (ii) mean that Requester 1's starting date is in the time interval requested by Requester 2, while (ii) and (iii) mean that Requester 2's ending date is in the time interval requested by Requester 1.
  A separate Jess rule is programmed to detect each of these four possible clashes. We call *1* the rule detecting the conflict intention1/intention21, *2* for intention1/intention22, etc.

4.4 Results

The execution trace in JessTab is displayed in Figure 3, in which we can see seven conflicts, each one beginning with the name of the rule that detected it followed by some explanations. For example, the first conflict was detected by the rule *2*, and is thus

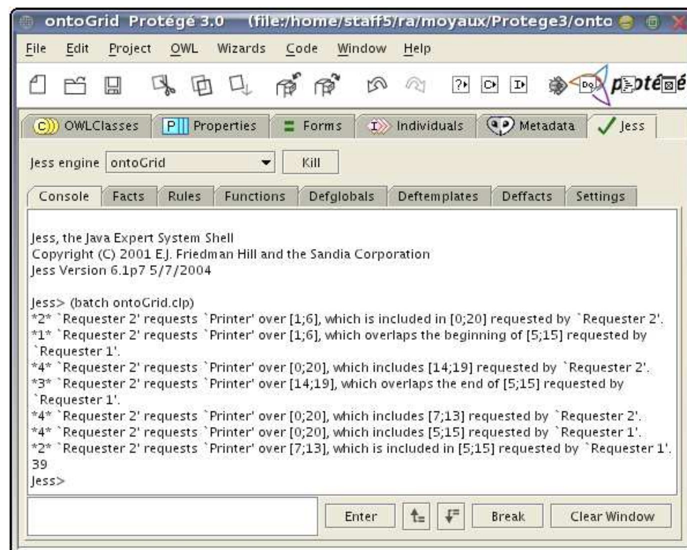

Fig. 3. JessTab detects the conflicts in the schedule of the non-shareable resource.

of the type intention1/intention22, but this first conflict is not between Requester 1's intention1 and Requester 2's intention22. In fact, this conflict is due to the fact that

Requester 2 wants to use Printer both over [1;6] and [0;20], and thus, the former time interval is included in the latter while Printer is non-shareable.

Conversely to this, the second clash is between two different requesters. In other words, inter-agent as well as intra-agent conflicts are detected. We have checked that it is possible to add more instances of resources, providers, requesters and registries and JessTab still detects the conflicts.

5 Conclusions

The effectiveness of the Semantic Web relies on enabling technologies that permit the various components – ontologies, reasoning engines, and agents – to work harmoniously together. The interactions need to be managed according to a theory that is understood and agreed upon by all the components (in the paper we loosely referred to these components as *agents*). Coordination is the process of managing the possible interactions between activities and processes. The premise of the work presented in this paper is that effective coordination requires the sharing of knowledge about activities, resources and their properties. Typically, this sharing is achieved statically, by hard-coding at design time the coordination mechanism in the agents. However, in more open systems, where the processes and resources of which the system is comprised are not known at design time, such an approach is often impossible. A viable alternative in this type of systems would be a *dynamic* approach, in which the coordination requirement is handled at *run-time*, rather than design time. Such approach allows the relevant processes to communicate their intentions with respect to future activities and resource utilisation, and gets them to "reason" about coordination at run time, with the goal of preventing negative interactions, and facilitating positive interactions. The communication implied by this solution requires an agreed common vocabulary for coordination, with a precise semantics, that is, an ontological approach to dynamic coordination.

This paper describes such an ontological approach to coordination, and presents our results with respect to a proof-of-concept implementation of the approach, in which multiple processes detect coordination relationships using a Jess/Protégé implementation of the ontology. This prototype is only intended to show how an inference engine may be used to perform coordination tasks in Semantic Web Services. In fact, inference engines have already been being used to check semantic consistency, e.g., Racer [15] both checks semantic consistency and improves the organization of Protégé knowledge bases.

This work is still at an embryonic stage, the results obtained by the proof-of-concept implementation are very promising and encourage us to proceed towards the development of a representation of coordination mechanisms using Semantic Web rule languages. One possible implementation strategy consists of representing the rules in SWRL [16] and using a reasoner such as Vampire [17], to reason about the coordination rules. This strategy raises a number of research issues, such as whether the choice of rules will fail to provide a definitive answer due to the undecidability of SWRL. As an improvement on our prototype, we could:

– *Make every agent register to only one registry* instead of every possible registry;

– *Add other protocols*: e.g, some methods of clash resolution, and the update of *Registrys* by *Agents*;
– *Add message format*: at the moment, we only handle a basic registration message;
– *Looking for the maximum size of a semantic registry*: how many resources, providers, and requesters make the semantic registry too slow (in a dynamic environment, unlike our prototype).

Acknowledgements

The work presented in this paper is partially funded by the FP6 EU project Ontogrid (FP6-511513). The authors would like to thank Sean Bechofer and Terry Payne for their insightful comments on this work.

References

1. Ben-Ari, M.: Principles of Concurrent and Distributed Programming. Prentice Hall (1990)
2. Wooldridge, M.: An Introduction to Multiagent Systems. John Wiley & Sons (2002)
3. Malone, T.W., Crowston, K.: The interdisciplinary study of coordination. ACM Computing surveys **26** (1994) 87–119
4. von Martial, F.: Coordinating Plans of Autonomous Agents (LNAI Volume 610). Springer-Verlag: Berlin, Germany (1992)
5. von Martial, F.: Interactions among autonomous planning agents. In Demazeau, Y., Müller, J.P., eds.: Decentralized AI — Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-89), Elsevier Science Publishers B.V.: Amsterdam, The Netherlands (1990) 105–120
6. Durfee, E.H.: Coordination of Distributed Problem Solvers. Kluwer Academic Publishers: Dordrecht, The Netherlands (1988)
7. Decker, K., Lesser, V.: Designing a family of coordination algorithms. In: Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, CA (1995) 73–80
8. OWL-S: OWL Semantic Web Services (2004): `http://www.daml.org/services/`.
9. WSMO: Web Service Modelling Ontology (2004): `http://www.wsmo.org`.
10. Stanford Medical Informatics: Protégé (2005) `http://protege.stanford.edu/` (accessed 31 March 2005).
11. Eriksson, H.: JessTab (2005) `http://www.ida.liu.se/~her/JessTab/` (accessed 31 March 2005).
12. Friedman-Hill, E.: Jess (2005) `http://herzberg.ca.sandia.gov/jess/`
13. W.W.W. Consortium: Web site for the specification of OWL (2004) `http://www.w3.org/2004/OWL/`.
14. Agent Oriented Software Group: Company web site (2004).
15. Haarslev, V., Möller, R.: Web site for the software Racer (2005) `http://www.cs.concordia.ca/Ehaarslev/racer/download.html`
16. SWRL. (`http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/`)
17. Tsarkov, D., Riazanov, A., Bechofer, S., Horrocks, I.: Using Vampire to reason with OWL. In McIlraith, S.A., Plexousakis, D., van Harmelen, F., eds.: Proc. of the 2004 International Semantic Web Conference (ISWC 2004). Number 3298 in Lecture Notes in Computer Science, Springer (2004) 471–485
18. PSL, process specification language. (`http://www.mel.nist.gov/psl/`)