Towards service-oriented ontology-based coordination.

Thierry Moyaux¹
Ben Lithgow Smith¹
Shamimabi Paurobally¹ *
Valentina Tamma¹
Michael Wooldridge¹

¹ Department of Computer Science, University of Liverpool

* Shamimabi Paurobally is now based within the School of Informatics, University of Westminster


This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Westminster’s products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

The WestminsterResearch online digital archive at the University of Westminster aims to make the research output of the University available to a wider audience. Copyright and Moral Rights remain with the authors and/or copyright owners. Users are permitted to download and/or print one copy for non-commercial private study or research. Further distribution and any use of material from within this archive for profit-making enterprises or for commercial gain is strictly forbidden.

Whilst further distribution of specific materials from within this archive is forbidden, you may freely distribute the URL of the University of Westminster Eprints (http://www.wmin.ac.uk/westminsterresearch).

In case of abuse or copyright appearing without permission e-mail wattsn@wmin.ac.uk.
Towards Service-Oriented Ontology-Based Coordination

Thierry Moyaux, Ben Lithgow Smith, Shamimabi Paurobally, Valentina Tamma and Michael Wooldridge

Department of Computer Science
University of Liverpool
Liverpool L69 7ZF, U.K.
{moyaux, ben, sha, valli, mjw}@csc.liv.ac.uk

Abstract

Coordination is a central problem in distributed computing. The aim is towards flexible coordination, managed at run-time, in open, dynamic environments. This approach would benefit from an explicit common vocabulary for coordination and hence, in a previous paper, we modelled coordination in an ontology, describing the activities carried out and the interdependencies among these activities. The purpose of this paper is to show how such an ontology can be used alongside a set of rules to perform coordination by managing the interdependencies among activities. The ontology and rules can then be used to provide a general purpose coordination tool in the form of a web service.

1. Introduction

Coordination is the management of interdependencies among activities [7]. It is a crucial problem for collaborative working, affecting many different paradigms of distributed computing, not least multi-agent systems, where coordination has been the subject of extensive research [9]. Our aim is to build on this research and adapt it for other distributed technologies such as web services and Grid computing, thereby developing a common coordination tool.

Coordination is often interpreted in a limited way, as synchronisation [2], which is generally concerned with the rather restricted case of ensuring that processes do not destructively interact with one another. Thus it is often handled by low-level hard-wired coordination mechanisms, e.g. semaphores, monitors, or locks [2]. However, this interpretation only highlights one aspect of coordination and becomes insufficient when considering how it might be achieved in more open systems, where the processes and resources composing the system may be unknown at design time [3] and may constantly evolve.

In such systems, we ideally want computational processes to be able to reason about the coordination issues in their system, and resolve these issues autonomously [3], with the goal of facilitating positive interactions (e.g. only performing a computation once though it may have been requested twice), whilst preventing negative interactions (e.g. preventing conflicts over the use of a non-shareable resource). One way to achieve this is to enable the relevant processes to communicate their intentions with respect to future activities and resource utilisation.

Such communication would require an agreed common vocabulary, with a precise semantics, and is therefore suitable for representation as an ontology. Based on previous work by the multi-agents systems community [3, 7, 10, 14] such an ontology has been developed [13]. It defines coordination in terms of agents (or "actors") and the possible relationships between them. It models the notion that these agents carry out activities involving resources owned by other agents. Here, the term agent is used loosely to refer to any process that might perform an action or own a resource; such processes need not embody the full set of capabilities classically used to denote agenthood such as autonomy, social ability and proactiveness [15].

To complement this ontology, we have developed a set of rules, which can be used to govern the actual process of coordination. These rules provide other vital functions; they are used to refine the definitions in the ontology, and to check the capabilities of the model represented by the ontology.

To address the general feasibility of an ontology based approach to dynamic coordination, we have also developed a prototype web service as a proof-of-concept. This prototype uses a Prot´eg´e [11] implementation of the coordination ontology and a Jess [5] implementation of the rules.

In related work, WS-Coordination [8] specifies a coordination service consisting of three kinds of sub-service: an Activation Service used by service providers to create the coordination context of their service; a Registration Service used by service requesters to inform the coordination service of their future need for the service; and several Protocol Services that perform the actual coordination. Essentially, it describes what a coordination service should look like and how to interact with it (in particular, describing the messages to be used in such interactions), but nothing is said about how the Protocol Services should perform the actual coordination.

Our goal in this paper is to discuss how the coordination ontology can be used to perform such coordination at run-time. In brief, the two research questions addressed in this paper are: How can the coordination ontology be used? And what can be achieved with the ontology?

An overview of the coordination ontology is given in Section 2. Section 3 presents the rules which implement the coordination mechanism. The prototype is then presented in Section 4, after which we conclude with a discussion, and some pointers to future work.
2. Coordination ontology

In general, an ontology is a formal description of a domain of discourse, which usually consists of a finite list of concepts and relationships between these concepts [12]. Figure 1 represents a part of our coordination ontology, that is, this figure represents the main concepts (i.e., classes such as OperationalRelationship) with all their inheritance relations (e.g. Peer is a subclass of OperationalRelationship), and all the relationships linking them to one another (object properties, e.g. every instance of OperationalRelationship has a property, hasTargetAgent, pointing to an instance of Agent). In our ontology the starting concept is Agent, which is used to denote resource owners, along with the entities in the system that perform activities requiring coordination. Agent is specialised into the following two subclasses, each denoting a role that an agent can play in the system at a particular time:

- **Provider** is the class of service providers. The class Resource has an object property owner of type Provider, which describes the fact that a resource is managed by a service provider.
- **Requester** is the class of service requesters. Every requester can be the actor of a CoordinableActivity.

An agent may play the role of requester and provider at the same time, therefore these two classes are not disjoint.

The class Interdependency describes the type of interrelationships existing between activities. This class is described by the object properties hasSource and hasTarget that both take an Activity as their value, and by the two datatype properties hasDuration and type. Composite activities, either conjunctive or disjunctive (Conjunctive- and DisjunctiveActivity), allow this class to be related to several activities. The type property is used to distinguish between hard and soft interdependencies. Hard interdependencies are those for which the source activity directly affects the success of the target activity. Conversely, soft interdependencies are those in which the source activity affects the progress (rather than the success) of the target activity. In other words, hard interdependencies are "mission critical", and must always be handled, whilst soft interdependencies merely affect the efficiency of the system.

Interdependencies can either be **Positive** or **Negative**:
- **PositiveCoordination** occurs when the interaction leads to an increase in the utility of the participants or the quality of the solution, while **NegativeCoordination** denotes an interaction which, if it occurs, will lead to a reduction in the quality of the solution or the utility of the participants.
- **NegativeCoordination** has the following subclasses:
  - **MutuallyExcludes** (hard **NegativeCoordination**): the two involved activities cannot occur at the same time, for example, because they both need to access the same non-shareable resource.
  - **Impedes** (soft **NegativeCoordination**): the activity denoted in hasSource may impede the activity denoted in hasTarget, but it does not make it impossible.

Agents are related to one another via the notion of an OperationalRelationship, which describes the priorities among them. For example, the subclass ContractualAuthority indicates that, in the context of an organization, the agent instantiating hasSourceAgent should take precedence over the agent instantiating hasTargetAgent. Similarly, LegalAuthority represents the same priority among agents, but according to norm and regulations, rather than according to an organization. Peer indicates that neither agent has any authority over the other.

An OperationalRelationship might be used to reschedule some CoordinableActivities, in order to manage the interdependencies defined among them. In particular, by moving a CoordinableActivity in time we might reduce or completely eliminate NegativeCoordination, while increasing the number of interactions that cause PositiveCoordination.

To illustrate these definitions, let us consider the five following activities which will be used throughout the paper:

- 'compute-phi' makes the resource named CPU-UNIQUE calculate the first digits of the Golden Number $\phi$.
- 'compute-pi-short' makes CPU-UNIQUE calculate the first fifty digits of $\pi$.
- 'compute-pi-long' calculates the first hundred digits of $\pi$.

1 In Antiquity, the Golden Number $\phi$, also called the Divine Proportion, was said to be the proportion of everything beautiful. It is the positive solution to $\phi^2 - \phi - 1 = 0$. 

Figure 1. An overview of the coordination ontology.
• ‘print-pi-short’ makes the printer called ‘PRINTER-UNIQUE’ print the first fifty digits of \( \pi \), and

• ‘print-pi-long’ prints the first hundred digits of \( \pi \).

Several instances of Interdependency should be managed, such as: (i) ‘compute-pi-short’ Enables ‘print-pi-short’, (ii) ‘compute-pi-long’ Subsumes ‘compute-pi-short’ (but it is not the case that ‘print-pi-long’ Subsumes ‘print-pi-short’), (iii) ‘print-pi-short’ MutuallyExcludes ‘print-pi-long’ and (iv) ‘compute-phi’ Impedes ‘compute-pi-long’. Interdependency (i) illustrates the meaning of the type hard: it is not possible to print \( \pi \) if it has not been previously calculated – this deals with more than just the efficiency of an activity. Similarly, (ii) is soft because the result of ‘compute-pi-short’ is assumed to be the beginning of ‘compute-pi-long’, i.e., it is possible to accelerate the operation of the overall system by using the expected result of ‘compute-pi-short’ inside of the result of ‘compute-pi-long’, but not exploiting this possible improvement would have no impact on the success of these two activities.

We can now describe the class CoordinableActivity representing these activities. We can see from Figure 1 that CoordinableActivity is related to the Requester who wants to carry out the considered activity, and also that its subclass AtomicActivity has a property requires. (Note that for the sake of the implementation in Section 4, AtomicActivity has two subclasses Compute and Print.) In addition to these two object properties, CoordinableActivity has eight further datatype properties:

- actualStartDate, actualEndDate, earliestStartDate, latestStartDate, latestEndDate and expectedDuration all contain a date, which is a string in our representation in OWL.

- shareableResult is a boolean representing whether the outcome of an activity, or a part of that outcome, can be used as the outcome of another activity. In the example (ii) above, we noted that ‘print-pi-long’ does not subsume ‘print-pi-short’, which is due to the fact that the outcome of these two activities is assumed to be non-shareable. In general, the outcome of a Compute is always shareable, while the outcome of a Print depends on the circumstances. For example, if you print a document only to read it, you may allow me to read it as well, in which case the shareableResult property of Print will have the value true. But if the printed document is a crossword that you fill in, shareableResult is now false because I do not want your finished crossword!

- status takes one of the values presented in Figure 2. As we can see, a CoordinableActivity initially has the status requested. It then becomes scheduled if no coordination with other activities is necessary, otherwise a change is proposed to the requester. The status eventually becomes either scheduled or failed, depending on whether an agreement has been reached with the requester. When the current date corresponds to the actualStartDate of a scheduled activity, the status changes to continuing, indicating that the activity is being performed. The activity may be suspended during its execution, but it will eventually take the status succeeded or failed.

![Figure 2. Life cycle of the property status of a CoordinableActivity.](image)

Note that any (datatype or object) property may not be associated with a value, and therefore, status may also take the value null, but this should not occur in our model. More details on the other classes of this ontology may be found in [13].

Finally, each Resource is managed by one of the CoordinationMechanisms indicated by the property isCoordinationWith. Every instance of CoordinationMechanism is a set of coordination rules, which are implemented in Jess in our system.

We divide the CoordinationRules into three classes, namely, CheckActivities, FindInterdependencies and CoordinationMechanism:

1. CheckActivities are fired by activities in order to modify them. These modifications aim at either completing these activities (e.g. we may infer information about one activity from another activity that subsumes it), or detecting inconsistencies among the composite conjunctive and disjunctive activities.

2. FindInterdependencies are fired by activities in order to add instances of positive and negative interdependencies.

3. CoordinationMechanism contains rules fired by interdependencies (asserted either at run-time by rules in FindInterdependencies or at design-time) in order to modify (or propose modifications to) activities. More precisely, these modifications aim at managing positive and negative interdependencies, so that activities are coordinated.

We discuss the first two classes now, and the most important, CoordinationMechanism, in the next section.

### 2.1. Check activities

The rules in CheckActivities define basic properties of CoordinableActivity. They capture some of the basic axiomatic properties of the ontology, and are independent of any coordination mechanism. For example, one rule (number 2 in the list below) checks whether a continuing activity has passed its latest start date, and if so, defines its status as failed. These rules should be fired before the rules in FindInterdependencies and CoordinationMechanism, and thus, they should have the highest priority. Our prototype currently uses the following rules:

1. **coordinableActivity1**: If an activity has the status continuing and its actualEndDate is less that its earliestStartDate, then its status should be changed to failed.

2. **coordinableActivity2**: Similarly to coordinableActivity1, if a continuing activity started after its latestStartDate, then it has failed.
3. conjAct-succeeded: A ConjunctiveActivity has successfully terminated if all its inner activities have successfully terminated. In practice, this is equivalent to: if a ConjunctiveActivity has inner activities listed in its property isComposedOf and none of these inner activities has a status which is not succeeded, then the status of the ConjunctiveActivity has to be set to succeeded (i.e., we use two negations to implement a “for all”).

4. disjAct-succeeded: A DisjunctiveActivity has successfully terminated if at least one of its inner activities has successfully terminated. The implementation is a simplification of the implementation of conjAct-succeeded, since we do not require either of the negations.

5. conjAct-fails: A ConjunctiveActivity fails as soon as one of its inner activities fails (rule similar to disjAct-succeeded).

6. disjAct-fails: A DisjunctiveActivity fails when all its inner activities fail (rule similar to conjAct-succeeded).

7. conjAct-actualStartDate (respectively, disjAct-actualStartDate): The actualStartDate of a ConjunctiveActivity (respectively, DisjunctiveActivity) is the actualStartDate of its inner activity which starts first. More precisely, if a ConjunctiveActivity (respectively, DisjunctiveActivity) has inner activities listed in its property isComposedOf, and one of these inner activities has an actualStartDate which has the value \( V \), and there are no other inner activities having an actualStartDate \( V_{\text{any}} \) so that \( V > V_{\text{any}} \) (respectively, \( V < V_{\text{any}} \)), then the actualStartDate of the ConjunctiveActivity (respectively, DisjunctiveActivity) has to be set to \( V \).

8. conjAct-actualEndDate (respectively, disjAct-actualEndDate): The actualEndDate of a ConjunctiveActivity (respectively, DisjunctiveActivity) is the actualEndDate of the inner activity that ends last (respectively, ends first).

9. conjAct-expectedDuration: The expectedDuration of a ConjunctiveActivity is the difference between the actualEndDate of its latest finishing inner activity and the actualStartDate of its earliest beginning inner activity.

10. disjAct-expectedDuration: The expectedDuration of a disjunctive activity is the expectedDuration of the inner activity that has the shortest expectedDuration.

11. conjAct-latestEndDate (respectively, disjAct-latestEndDate): The latestEndDate of a ConjunctiveActivity (respectively, DisjunctiveActivity) is the latestEndDate of the inner activity finishing last (respectively, finishing first).

12. conjAct-latestStartDate (respectively, disjAct-latestStartDate): The latestStartDate of a ConjunctiveActivity (respectively, DisjunctiveActivity) is the latestStartDate of the inner activity which has the earliest (respectively, latest) latestStartDate.

13. conjAct-earliestStartDate (respectively, disjAct-earliestStartDate): The earliestStartDate of a ConjunctiveActivity (respectively, DisjunctiveActivity) is the earliestStartDate of the inner activity which has the latest (respectively, earliest) earliestStartDate.

2.2. Find additional interdependencies

The rules in FindInterdependencies infer new instances of Interdependency from Activities. Such interdependencies are asserted either at run-time by the rules now presented, or at design-time. The first four rules add NegativeCoordinations which are hard (MutuallyExcludes) or soft (Impedes):

14. mutually-excludes1 (respectively, impedes1): When the end of a time slot for an activity \( A2 \) overlaps the beginning of the time slot for another activity \( A1 \), and \( A1 \) and \( A2 \) both require a non-shareable (respectively, shareable) resource, then record this by asserting an instance of MutuallyExcludes (respectively, Impedes). More precisely, if Activity \( A1 \) has actualStartDate \( SD1 \), actualEndDate \( ED1 \) and requires \( R12 \), Activity \( A2 \) has actualStartDate \( SD2 \), actualEndDate \( ED2 \) and requires \( R12 \), the property shareable of Resource \( R12 \) is false (respectively, true), and \( SD1 > SD2 \), \( ED1 > ED2 \) and \( ED2 > SD1 \), then assert an instance of MutuallyExcludes (respectively, Impedes) with hasSource \( A1 \), hasTarget \( A2 \) and hasDuration \( ED2 − SD1 \).

15. mutually-excludes2 and impedes2: When the time slot for an activity is included in the time slot for another activity, record this by asserting a MutuallyExcludes or an Impedes, depending on the shareable property of the resource used by both activities.

16. mutually-excludes3 and impedes3: When the beginning of the time slot for an activity overlaps the end of another time slot, assert an instance of MutuallyExcludes or Impedes, depending on the shareable property of the resource used by both activities.

17. mutually-excludes4 and impedes4: When the time slot for an activity includes another time slot, assert an instance of MutuallyExcludes or Impedes, depending on the shareable property of the resource used by both activities.

The next rule finds instances of PositiveCoordination. There is only one such rule and it deals with the positive interdependency Enables:

18. enables-compute->print: if a composite ConjunctiveActivity both contains an instance of Compute and of Print, then we assume that the result of the compute is to be printed, and thus the Print should be after the Compute.

In this paper, we also include another PositiveCoordination, namely Subsumes, but the description of activities in our ontology is not yet sufficiently detailed to state that an activity subsumes another, that is, that all the actions carried out in an activity are also carried out in another activity. As future work, we may use a more detailed description for CoordinationActivity, e.g., by using a representation in BPEL4WS [6] (Business Process Execution Language) instead of our class. With such an extension, we expect to be able to see that ‘compute-pi-long’ Subsumes ‘compute-pi-short’, while there is no Subsumes relation between ‘compute-phi’ and ‘compute-pi-long’, or between ‘compute-phi’ and ‘compute-pi-short’.
3. A coordination mechanism

This section presents the core of the paper, that is, the coordination rules that manage interdependencies among activities. Here, we illustrate the use of our ontology with an example set of rules for a CoordinationMechanism. Other coordination mechanisms are also possible, depending on how the designer would like to coordinate her system.

Every rule in CoordinationMechanism is fired by a single (positive or negative) Interdependency, and modifies an AtomicActivity. This modification consists of changing only the properties actualStartDate, actualEndDate and status. To choose the time to which an Activity should be moved, the right-hand side of a coordination rule queries the knowledge base to find a “space” among the atomic activities for which this AtomicActivity may be fitted in. The invocation of queries to find such a “hole” is put in functions. These functions and the queries they use are presented in Subsection 3.1. Once an Interdependency has been managed by moving some activities, this Interdependency is removed as well as all the other interdependencies involving the moved AtomicActivity, in order to maintain the consistency of the knowledge base. Note that the process of coordination only deals with moving atomic activities, i.e., conjunctive and disjunctive activities are only there to manipulate atomic activities in a simpler way and to add some relations among activities.

Since coordination only deals with managing interdependencies, we could write a generic coordination rule. However, it seems important that hard interdependencies be coordinated before soft ones, since the former deal with the success of the operation of the system while the latter deal only with efficiency. This is why we introduce a first division between rules managing hard interdependencies (which have a higher priority) and rules managing soft interdependencies.

These two sets of rules are further split to take different kinds of operational relationships between agents into account. We focus in this paper on these pairs of rules:

1. the coordination rules managing interdependencies involving a single Agent. There is one rule for hard Interdependency and one for soft.
2. the rules taking a ContractualAuthority (a subclass of OperationalRelationship) into account. As previously stated, a ContractualAuthority indicates that the Agent denoted by the property hasSourceAgent has contractual authority over the Agent denoted by hasTargetAgent; that is, both “belong” to the same organisation, and in the context of this organisation, hasSourceAgent should take precedence over hasTargetAgent. Again, there is one rule for hard Interdependency and one for soft.
3. the rules taking a LegalAuthority into account. These rules are essentially similar to rules for ContractualAuthority, except that the OperationalRelationship is due to the law rather than an organisation.

When the coordination process is time-constrained and it is good practice to avoid priorities in rule-based systems, our prototype in Section 4 does not use them.

To summarise, we are left with the following six coordination rules: manage-softInterdependency-contractualAuthority, manage-softInterdependency-legalAuthority and manage-softInterdependency-singleRequester, manage-hardInterdependency-contractualAuthority, manage-hardInterdependency-legalAuthority and manage-hardInterdependency-singleRequester. The first three rules might have the same priority as the rules in FindInterdependencies that find soft interdependencies, while the last three rules might have the same priority as the rules in FindInterdependencies that find hard interdependencies. Note that these rules do not make a distinction between rules managing Positive- and NegativeCoordination. The remainder of this section details the implementation of these six coordination rules.

3.1. Functions and queries

To simplify the right-hand side of the coordination rules, i.e. the part that solves interdependencies, we make use of two functions, namely find-a-free-schedule and delete-interdependency. These two functions use queries to get facts from the knowledge base, with a query being a special kind of rule with no right-hand side fired by program control (normal rules are fired automatically by Jess) [5].

The first of our two functions is find-a-free-schedule(?duration, ?resource12) which returns the beginning of the next time slot of ?resource12 which is free for more than ?duration. The function first checks if it is possible to insert a new schedule among the activities already scheduled for ?resource12 by invoking the query list-free-schedules(?duration, ?resource12). This query returns a list of pairs of activities. Each pair represents a sufficiently long free slot in the schedule and is comprised of the activity that precedes that slot and the activity that it follows. If the query successfully returns one or more pairs of activities, the function returns the actualEndDate of the first activity of each pair. If list-free-schedules returns nothing, the function has to schedule the new activity at the end of the current schedule. For this purpose, the query list-all-schedules(?resource12) returns all the activities scheduled for ?resource12, and the function returns the latest actualEndDate of all the instances of Activity returned by list-all-schedules.

The function delete-interdependency(?activity2remove) deletes all deprecated instances of Interdependency. This function is used to maintain the consistency of the knowledge base when an activity is removed or scheduled at a different time, in which case the previously found interdependencies no longer hold. This function calls the query list-interdependencies2remove(?activity2remove) which returns the list of all interdependencies in which ?activity2remove appears in either hasSource or hasTarget. Next, the function deletes all these interdependencies.

These two functions are just short-hands to make coordination rules simpler, but their content could have been put in the code we now present.

3.2. Detail of the coordination rules

The coordination rules are fired by instances of Interdependency which are asserted when the corresponding In-
Interdependency is violated. When two Requesters are involved, the coordination process uses operational relationships to select the AtomicActivity to be moved, otherwise it moves the shortest AtomicActivity.

19. manage-softInterdependency-singleRequester: When an Agent requests two AtomicActivities which conflict with each other, the shortest one is rescheduled later by this rule whose salience is zero when priorities are used. More precisely, when there is an Interdependency with hasSource pointing to Act1 and hasTarget pointing to Act2 and type is soft, an AtomicActivity Act1 has expectedDuration $E_{D1}$ and status different from proposed or continuing, an AtomicActivity Act2 has expectedDuration $E_{D2}$, requires $R_2$ and status different from proposed or continuing, and $E_{D1} > E_{D2}$, then move Act2 (we only need Act2's Resource $R_2$ to find when this Resource can carry out Act2 at another moment). Moving Act2 consists in setting its actualStartDate to the result of find-a-free-schedule ($E_{D2}, R_2$), and its actualEndDate to the sum of $E_{D2}$ and of the result of find-a-free-schedule($E_{D2}, R_2$).

20. manage-hardInterdependency-singleRequester: This rule is the same as manage-softInterdependency-singleRequester, except that it is fired by a hard Interdependency, and its salience is higher when priorities are used.

21. manage-softInterdependency-contractualAuthority: If two agents request to carry out two activities related by a ContractualAuthority, the OperationalRelationship among these agents is exploited to find out which activity has precedence over the other. More precisely, when there is an Interdependency with hasSource pointing to Act1, hasTarget pointing to Act2 and type being soft, an AtomicActivity Act1 having actor $A_1$ and status different from proposed or continuing, an AtomicActivity Act2 having actor $A_2$, status different from proposed or continuing, requires $R_2$ and expectedDuration $E_{D2}$, a ContractualAuthority with hasSourceAgent $A_1$ and hasTargetAgent $A_2$, then move Act2. Here, the expectedDuration $E_{D2}$ and Resource $R_2$ are necessary to move Act2. As for manage-softInterdependency-singleRequester, the actualStartDate of Act2 is set to the result of find-a-free-schedule($E_{D2}, R_2$), and its actualEndDate to the sum of $E_{D2}$ and the result of find-a-free-schedule($E_{D2}, R_2$).

22. manage-hardInterdependency-contractualAuthority: This rule is the same as manage-softInterdependency-contractualAuthority, except that it is fired by a hard Interdependency and its salience is higher.

23. manage-softInterdependency-legalAuthority: This rule is similar to manage-softInterdependency-contractualAuthority, except that it is fired by LegalAuthority instead of a ContractualAuthority. This is due to the fact that these two subclasses of OperationalRelationship have the same meaning, but with different strength.

24. manage-hardInterdependency-legalAuthority: This rule is the same as manage-softInterdependency-legalAuthority (and thus, similar to manage-hardInterdependency-contractualAuthority), except that is is fired by a hard Interdependency and its salience is higher.

3.3. Integration of the rules in the ontology

At the moment, all the rules are implemented in Jess and we have not yet discovered any necessary rules that could not be written in Jess. However, we need to translate our rules from Jess into a standardized rule language, such as SWRL or ruleML, in order to add them as instances of CheckActivities, FindInterdependencies and CoordinationMechanism. We expect to use SWRL, as it describes rules in OWL, which is the language used to represent the rest of the ontology. More precisely, we expect to use SWRL extension to first-order logic, because it adds the negation to standard SWRL, which is required by some of our rules, e.g. conjAct-actualStartDate (rule 7) contains this negation in its description: “... and there are NO other inner activities having ...”.

4. Web Service Implementation

We have implemented a prototype with the JessTab 1.1 [4] plug-in for Protégé 3.0 [11]. This integrates the inference engine Jess (version 6.1p7 [5] in our case) with Protégé, enabling the Jess rule-engine to work with a Protégé knowledge base. Specifically, the inference engine in JessTab can (i) access the ontology and the instances represented in Protégé, (ii) directly manipulate the ontology and instances, (iii) infer new facts from the ontology and instances, and (iv) perform all the other programming tasks permitted by Jess, such as performing computations or launching Java operations.

To create the prototype, we first implemented the coordination ontology in Protégé using the OWL plug-in. We then created a web service, which wrapped a Jess engine containing the ontology and rules. This web service provides a number of methods allowing us to register/deregister requesters and providers, along with new activity requests. It also allows us to request the whole schedule for a given resource, and logs all details of rules that are fired.

With the web service running, instances representing the activities to be coordinated were added. These activities are represented in Figure 3, in which we can see there are five AtomicActivities (namely, ‘compute-phi’/-pi-short’/-pi-long’ and ‘print-pi-short’/-long’) and two ConjunctiveActivities. These two composite activities are ‘conjAct-short’, which contains ‘compute-pi-short’ and ‘print-pi-short’, and ‘conjAct-long’, which consists of ‘compute-pi-long’ and ‘print-pi-long’. Adding these two
AtomicActivities allows rule 18 (enables-compute->print) to infer ‘compute-pi-short’ Enables ‘print-pi-short’ and ‘compute-pi-long’ Enables ‘print-pi-long’. Figure 3 represents some of the interdependencies arising amongst the activities. In particular, the instances of Compute are related by different kinds of interdependencies, that is, ‘compute-phi’ Impedes (and is impeded by) ‘compute-pi-short’. While Figure 3 should also mention that ‘compute-pi-long’ Subsumes ‘compute-pi-short’, this Subsumes does not appear on the figure because the current version of the prototype is not able to determine that. Instead, our prototype finds that ‘compute-pi-long’ Impedes (and is impeded by) ‘compute-pi-short’, which is also true, but exploiting a Subsumes would make the overall system much more efficient than avoiding an Impedes. It is with a view to recognising such a Subsumes relationship that we plan to use a more detailed definition for AtomicActivity using, for example, BPEL4WS.

We can also see that ContractualAuthority ensures that ‘requester-short’ has a higher priority than ‘requester-long’. Next, Figure 4 shows the actualStartDate and actualEndDate of the five AtomicActivities before the coordination. For example, ‘requester-long’ has requested the activity ‘compute-pi-other’ in the time slot [1;4] with the resource ‘CPU-UNIQUE’.

When a new activity request is registered by the web service, the rules detailed in Section 3 are executed, thus coordinating the activities. Figure 5 shows the execution trace, illustrating the system’s activity. Each firing of a rule in FindInterdependencies begins with the word “INTDEP” followed by the name of the fired rule and an explanation of what is done. Similarly, each fire of a rule in CoordinationMechanism begins with the word “MCA” followed by the name of the fired rule and an explanation of what is done. Rules in CheckActivities do not display anything.

Figure 6 represents the result of coordination, in which we can see that activities can be coordinated with the use of an ontology.

5. Conclusions & Future Work

This paper has investigated the flexible and explicit coordination of distributed systems working in environments which are open, dynamic and evolving. We ultimately aim at coordination managed at run-time rather than being hard-wired. For that purpose, we developed an ontology of coordination which models interdependencies among activities. Regarding this ontology, we raised two questions in the introduction. The first question is how to use a coordination ontology: we answered this by presenting an example of coordination mechanism which manages the interdependencies among activities. The second question is how to use a coordination ontology: our prototype answered this by demonstrating that our approach based on an ontology seems feasible and efficient.

The coordination mechanism and its implementation presented in the section above are a first step towards a decentralised, explicit coordination mechanism. However, we still have to resolve some issues. For instance, if some computations are not possible, what kinds of coordination rules are not computable, and are there enough rules to enable a high degree of coordination? In our implementa-
tion, we are able to solve all the implemented hard interdependencies and to profit from some soft ones. In other words, activities are coordinated but the quality of the coordination can still be improved. We have therefore empirically demonstrated that centralised dynamic coordination is achievable. The next step is to verify whether decentralized coordination can be supported by the coordination model represented in our ontology.

A number of other issues suggest themselves. First, the issue of whether and how rules should be prioritized. Priorities arise naturally in general and some principles were proposed to organize them in any rule system [1]: (i) higher priority of the source of the rule (e.g. federal law has a higher priority than state law), (ii) a more recent rule is preferred, and (iii) a rule is more specific, i.e., exceptions are stronger than the general case. In our case study, no priorities are used because we are not time-constrained. However, we indicated that some priorities should be used if we are not sure all rules will be fired, e.g. all hard interdependencies must be managed before any soft interdependency. In our model, the class OperationalRelationship already manages the kind of priority (i) above. We expect that our rule base will be consistent, so that (ii) does not apply to us. Case (iii) looks to be the most probable in our approach, but we do not expect it to occur. Finally, priorities depending on the application domain may be added to accelerate the coordination process, e.g. we may empirically see that managing a Mutually Excludes in an application for insurance companies generally avoids managing two subsequent Enables.

Another issue is the interaction of our ontology with existing standards. For example, we are able to represent our entire model of coordination in OWL Full, but can we write the rules manipulating this model in SWRL or ruleML? Specifically, can we represent all the code in the right-hand side (in particular, the content of functions and their queries) of Jess’ rules in SWRL or ruleML?

As future work, we plan to (i) use SWRL to describe the coordination rules in a standardized format instead of in Jess, (ii) replace the class AtomicActivity with a more precise one that follows the representation used by BPEL4WS, (iii) add a means of simulating the passage of time and activities competing, (iv) investigate means of decentralising the web service.

Acknowledgments The work presented in this paper is funded by the FP6 EU project Ontogrid (FP6-511513).

References