Applied Artificial Intelligence, 25:235–265, 2011

ISSN: 0883-9514 print/1087-6545 online
DOI: 10.1080/08839514.2011.553376

Taylor & Francis
Taylor & Francis Group

AN ONTOLOGY FOR COORDINATION

Ben Lithgow Smith, Valentina Tamma, and Michael Wooldridge
Department of Computer Science, The University of Liverpool, Liverpool, United Kingdom

□ *Independent agents that interact within open, distributed, and decentralized environments need to collaboratively regulate their activities in order to facilitate the harmonious and successful achievement of possibly conflicting tasks. Coordination is the process of managing these interactions by identifying and possibly resolving the interdependencies occurring between such activities. A successful coordination mechanism facilitates mutually beneficial interdependencies (e.g., by ensuring activities are not duplicated) while avoiding adverse outcomes (e.g., by preventing two processes simultaneously accessing the same non-shareable resource, potentially causing deadlock). However, for such a mechanism to work effectively within open systems, agents need to communicate and reason about activities, resources, and their properties; i.e., to commit to a shared* ontology *of coordination that defines the semantics underlying the different coordination regimes. This article describes an ontological approach to coordination in which agents dynamically manage the interdependencies that arise during their interactions. A proof-of-concept implementation in the insurance domain is described and empirically evaluated.*

INTRODUCTION

The need for coordination frequently arises in open distributed environments, where multiple independent parties need to interact for diverse reasons. For instance, regulation of access to shared resources is necessary in order to prevent deadlock or starvation scenarios, whereas different parties or *agents* might need to cooperate for the successful completion of a task whose achievement benefits all the parties involved. Furthermore, there may be an intrinsic order between the activities carried out by agents such that one cannot take place without another being executed first. In each of these scenarios, activities carried out by different actors may affect one another, thus creating interdependencies between them. Such interdependencies are not necessarily disadvantageous; they may also be beneficial. Different tasks may share a common intermediate state; thus,

The work presented in this paper was funded by the FP6 EU project Ontogrid (FP6-511513).
Address correspondence to Valentina Tamma, Department of Computer Science, The University of Liverpool, Liverpool L69 3BX, United Kingdom. E-mail: valli@CSC.liv.ac.uk

by identifying such scenarios, a duplication of effort can be avoided (thus, reducing cost or a drop in utility). Failure to exploit the interdependency does not prevent the processes from successfully completing their tasks, but it does make the successful completion of such tasks less efficient.

One of the most used definition of coordination is "the management of interdependencies amongst activities" (Malone and Crowston 1994), which has been the subject of extensive research, not least in the field of multi-agent systems (Singh 2005). Several approaches for coordination have been identified, in addition to a classification of the types of interdependencies. A simple approach for providing coordination is to use hard-wired, low-level constructs (such as semaphores or locks) to ensure that various activities do not destructively interfere with one another (Ben-Ari 1990). This method, known as *synchronization*, is useful in static environments, where the resources and activities comprising the system are well known in advance and can be taken into account at design time. However, in more open systems, in which resources can come and go and may constantly evolve, it is often impossible to anticipate every eventuality at design-time, and thus the use of synchronisation may fail. In such systems, it makes sense to enable agents to resolve any interdependencies *autonomously* (Decker and Lesser 1995), by providing them with the ability to reason about the activities they wish to perform and the consequent coordination issues that arise. To achieve this, agents will need to communicate with one another about their intentions to utilise particular resources. This communication is based on an agreed common vocabulary with explicit semantics so that all the agents can communicate in the same terms, or in other words an *ontology* of coordination. This article details such an ontology (Tamma et al. 2005) based on previous work by the multi-agents system community (Decker and Lesser 1995; Malone and Crowston 1994; Singh 1998; von Martial 1992). The semantics of the coordination mechanisms are detailed by a set of rules that can be used to manage activities and resolve the interdependencies between activities (Moyaux et al. 2006).

The feasibility of this approach was determined by producing a proof-of-concept demonstrator where ontologies and rules are wrapped in a web service acting as a centralized coordinator with which resources can register. Therefore, agents that perform activities using these resources can invoke the service that will coordinate the various requests, detecting, and resolving interdependencies appropriately. A visualization client was implemented as an ancillary service to submit and visualize the different calls to the service. The client monitors the internal state of the service and uses a Gantt chart to illustrate the various resources, activities, and interdependencies to the user.

The aim of this article is to illustrate the coordination ontology and rules, provide a brief overview of the coordination service, and describe how this system can be applied to a real-life use case taken from the domain of car insurance. An overview of the background work in coordination is

given in the Background section. The Coordination Ontology section describes how this is translated into the coordination ontology. The set of rules that implement the coordination mechanism are then presented in the Coordination Rules section before a description of the coordination service implementation and visualisation client are given in the Implementation section. The Evaluation section then demonstrates how the approach can be applied to a use case. Finally, an evaluation is provided along with some conclusions and directions for future work.

BACKGROUND

Coordination is possibly the defining problem of cooperative working, and it is essential when the activities that agents perform can interact in any way. The coordination problem is concerned with how to *manage interdependencies between the activities of agents* (Malone and Crowston 1994). Consider the following real-world examples.

- *Jerry and George want to leave a room, and so they independently walk towards the door, which can only fit one person through at a time. Jerry graciously permits George to leave first.* In this example, the activities need to be coordinated because there is a resource (the door) that both people wish to use, but which can only be used by one person at a time.
- *George intends to submit a grant proposal, but in order to do this, he needs Jerry's signature.* In this case, George's activity of sending a grant proposal depends upon Jerry's activity of signing it off—George cannot carry out his activity until Jerry's is completed. In other words, George's activity *depends* upon Jerry's.
- *Jerry obtains a soft copy of a paper from a Web page. He knows that this report will be of interest to George as well. Knowing this, Jerry pro-actively photocopies the report and gives George a copy.* In this case, the activities do not strictly need to be coordinated; since the report is freely available on a Web page, George could download and print his own copy. But, by pro-actively printing a copy, Jerry saves him some time.

These different examples illustrate the various forms that coordination can take depending on the types of interdependencies to be managed. For this reason, this section aims to survey some of the literature that identifies different types of coordination and their relationships. Indeed, it is worth noting that coordination, as defined above, encompasses the well-known (and widely studied) concept of *synchronization* (Ben-Ari 1990). Synchronization 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 synchronization

problems involve *hard-wiring* coordination regimes into program code. Thus, for example, a programmer may flag a JAVA method as synchronized, indicating that a certain access regime should be enforced whenever the method is invoked. However, in large-scale, open, dynamic systems, 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 that can reason about coordination issues dynamically, we must first identify the possible interaction relationships that may exist between the agents' activities. Hence, the goal is to derive and formally define the possible interaction relationships that may exist between activities. Some prior work on this topic exists—in their coordination theory, Malone and Crowston (1994) identify several interdependencies between processes (which they refer to as *activities*), while von Martial (1992) proposes a complementary high-level typology for coordination relationships.

Malone and Crowston (1994) propose a systematic and interdisciplinary study of coordination and a theory that attempts to model the different forms of coordination in terms of dependencies between activities:

Shared resources: this is possibly the most common type of dependency, and it usually occurs whenever multiple activities or processes share limited resources. The management of interdependencies between activities caused by the need to share a resource requires efficient *resource allocation processes* such as *scheduling*. A special case of resource allocation is the task assignment dependency, where the resource to allocate is the limited time that actors can devote to a task. Malone and Crowston suggest that the resource allocation methods used to deal with shared resources can be applied to cases of task assignment.

Producer/consumer relationships: this is another common type of dependency that can be encountered in diverse fields spanning from manufacturing to logistics to computer systems, and it encompasses all cases where the product of the execution of a process is used by another process. The producer/consumer relationship can be further specialised into the following types of dependencies:

Prerequisite constraints: In this dependency, the producer must complete its activity *before* the consumer activity can begin. *Notification, sequencing*, and *tracking* methods can detect and resolve dependecy conflicts.

Transfer: this dependency refers to the action of physically transferring something from the producer to the consumer. If what is transferred is *information* then the dependency is called *communication*. The generalized approach to manage this relationship would be to place a buffer between the consumer and producer processes and allocate space in the buffer to either of them, but not to both.

Usability: This less obvious dependency within a producer/consumer relationship refers to the need for what is produced to be *usable* by the activity that consumes it. The common method to handle this dependency is *standardization*, i.e., the creation of uniformly interchangeable output in a form that is known to the user. The usability dependency plays an important role in information systems where information is exchanged between systems. Various methods have been developed to resolve usability conflicts, such as interchange formats (Genesereth 1991) and ontologies (Studer, Benjamins, and Fensel et al. 1998).

Design for manufacturability: This type of dependency refers to the possible relationships existing between activities, such as temporal or hierarchical, and the constraints they pose on their execution. Special kinds of design for manufacturability relationships are:

Simultaneity constraints: This type of dependency refers to whether two activities or events need to occur at the same time or not. Scheduling meetings between people is an example of this dependency: all the participants need to be free at the same time for the meeting to take place.

Task/subtask: In this type of dependency some activities are all necessary in order to achieve an overarching goal. This usually happens when the goal is *decomposed* into subgoals, whose achievement allows the overall goal to be achieved. A typical example of this relationship can be seen in planning systems where goals are decomposed by a planner into a set of elementary activities.

von Martial (1992) proposes a high-level classification of coordination relationships. He suggested that relationships between activities could be classified as 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" (von Martial 1990, p. 111). In other words, positive relationships lead to an increase in the quality of the solution or utility of participants whereas negative relationships lead to a reduction in the quality of the solution or utility of the participants. Such relationships may be *requested* (one agent *explicitly* asks another for help with its activities) or *non requested* (it so happens that by working together mutliple agents can achieve a solution that is better for at least one of them, without making the other any worse off). von Martial distinguishes three types of non-requested relationships:

The action equality relationship: Jerry and George plan to perform an identical action and, by recognizing this, one of them can perform the action alone, thereby saving the other some effort.

The consequence relationship: The actions in Jerry's plan have the side-effect of achieving one of George's goals, thus relieving George of the need to explicitly achieve it.

The favour relationship: Some part of Jerry's plan has the side effect of contributing to the achievement of one of George's 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* (PGP) (Durfee 1988). The basic idea of PGP is that an agent can represent the activities it intends to perform as a plan. It then exchanges this plan of local activity with other agents in order to identify possible interactions (positive or negative). Changes to one or more plans can then be proposed in order to improve performance and the planned local activities are modified in accordance with the coordinated proposal. This work led Durfee to propose the Common Representation for Coordination Hypothesis which stated that "organizations, plans and schedules have a common representation, but differ in their degree of specificity along different descriptive dimensions." (Durfee 1993). He termed this common representation a *behavior* and included amongst the descriptive dimensions: what the behavior was intended to achieve; how it would attempt to achieve it; who was participating in the behavior; when the behavior would occur; and why the behavior has been instituted.

The ideas of PGP and some of the notions identified in Malone and Crowston were refined in Decker's subsequent work on *Generalized Partial Global Planning* (GPGP) in the TÆMS testbed (Decker and Lesser 1995). GPGP focuses on coordination while agents are scheduling their activities rather than when they are planning to meet goals. Whereas in PGP agents exchange complete schedules at a fixed level of abstraction, in GPGP agents exchange scheduling commitments to particular tasks at any level of abstraction. It utilizes domain-dependent mechanisms for detecting and predicting coordination relationships and domain independent mechanisms to manage them (by posting constraints to the local scheduler). Five techniques are used for coordinating activities:

- *Updating non-local viewpoints*: Agents have only local views of activities 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 therefore, are 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 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 other 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 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 no alternative course of action is considered.

Another body of work was performed by Singh, who proposed an event-based linear temporal logic for scheduling service calls (Singh 2000). This can be used to provide guards on events, thereby enabling events to be ordered and to be permitted or not based upon the occurence of other events. The approach can be used to enforce coordination relationships such as:

- *enables*: event *f* cannot occur unless event *e* occurs beforehand.
- *conditionally feeds*: if events *e* and *f* both occur then *e* occurs before *f*.
- *guaranteeing enables*: event *f* can only occur if event *e* has occured or will occur.
- *initiates*: event *f* occurs if and only if event *e* precedes it.
- *jointly require*: if events *e* and *f* occur in any order then event *g* must also occur (in any order).
- *compensates*: if event *e* occurs and event *f* does not then event *g* must be performed.

Singh also stated an important consideration for designing coordination mechanisms: "there is a trade-off between reducing heterogeneity and enabling complex coordination." (Singh 2005, p. 282) So the more detail of tasks that is given, the better the coordination mechanisms that can be designed, but the less widely applicable those mechanisms will be. When designing a general purpose coordination mechanism then, it is best

to focus on the most widely shared attributes of tasks. From these, a core set of coordination mechanisms can be designed, which can be extended with more domain-specific mechanisms.

In related work, WS-Coordination[1] specifies a coordination service consisting of three kinds of sub-services: 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), however, nothing is said about how the Protocol Services should perform the actual coordination.

Based on all these bodies of work, an ontology for coordination was designed, which is presented in the next section. Although ontologies for service based computing have been developed, such as OWL-S[2] and Web Services Modeling Ontology (WSMO),[3] 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 modeled as a workflow that is determined *a priori* 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*).

COORDINATION ONTOLOGY

Figure 1 provides an overview of the coordination ontology, illustrating the main concepts and relationships. 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?

The sub-sections that follow describe the ontology: the key concepts, the slots associated with these concepts, the relationships between these concepts, and axioms. This description does not present all of the components of the ontology: the aim is to provide a good overview of the ontology, rather than present all the low-level technical details.

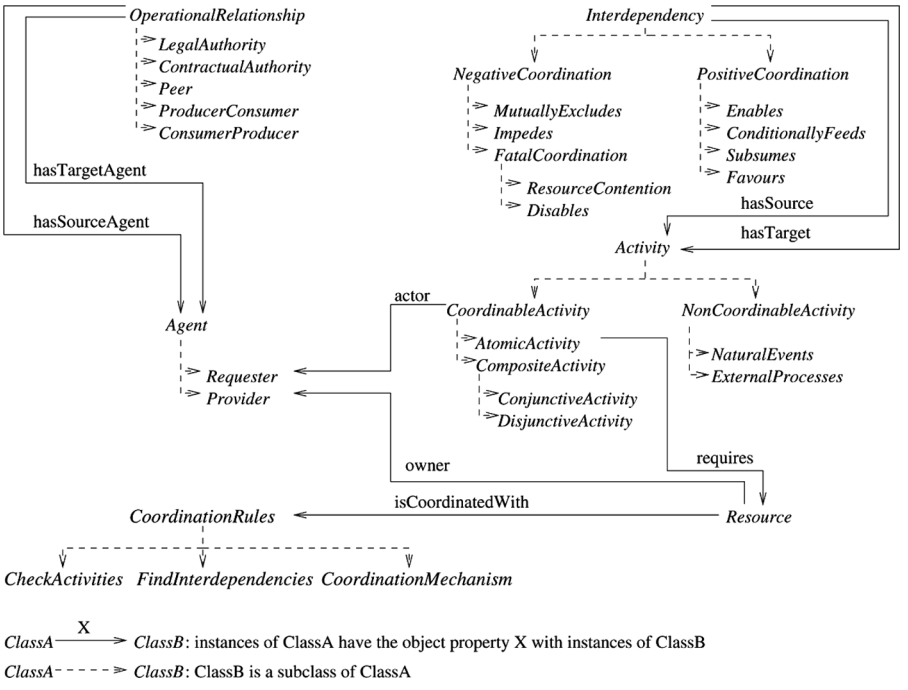

FIGURE 1 An overview of the coordination ontology.

Agents

Our starting concept is *Agent*, which 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). Agents can provide or consume a *resource*. To this end, there are two subclasses of agent, *provider*, and *requester*. As an agent may be simultaneously both a provider and a requester these subclasses are not disjoint from one another.

Resources

The *Resource* concept describes resources that may be required to expedite an activity. It 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 "working" or "broken."

- *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 artifact 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.

PROCESSES AND ACTIVITIES

The next concept is *Activity*, whose definition was influenced by the OWL-S model of processes.[4] It represents 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 activity 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 there are 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, the *NonCoordinableActivity* concept. This concept is intended 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:

- *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) irrespective of what the agents in the system do.

- *External processes*: These are processes—either physical world processes or natural processes—that 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.

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 or is scheduled to begin; 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 or is scheduled to end; null indicates that this information is not known;
- *shareable result*: a Boolean indicating whether the result of the activity can be shared with other agents;
- *status*: an enumeration type, which takes a value as follows: An activity begins by being *requested* and then becomes *scheduled* if no coordination is required or *proposed* if a change has been proposed. If the activity starts before its earliest start date or after its latest start date or if it ends after its latest end date then it is *outOfBounds*. If the results of the activity are available elsewhere then it is *superfluous*. If the activity is no longer needed for some reason then it is *redundant*. When the activity is performed it is *continuing* though it may become *suspended*. When the activity finishes it must have either *failed* or *succeeded*.

There are two direct sub-classes of coordinable activity: *AtomicActivity* and *CompositeActivity*. An atomic activity is the most basic type of activity and is indivisible into other activities. It has an additional property *requires*, which states the resource that it requires. A composite activity is one which is made up of other coordinable activities. Thus, they can be viewed as being arranged into an and/or tree hierarchy of coordinable activities (atomic or composite), with atomic activities as leaves of the tree. A slot *composedOf* contains the list of sub-activities. There are two sub-classes of composite activity:

- *ConjunctiveActivity*: a composite activity that succeeds if all of its sub-activities succeed
- *DisjunctiveActivity*: a composite activity that succeeds if any one of its sub-activities succeeds

Though these two classes may be used directly to implement coordination mechanisms, it is generally more useful to extend them by creating further subclasses with additional semantics. This is illustrated in practice in the Evaluation section.

Interdependencies Between Activities

The *Interdependency* concept is used to describe the various inter-relationships that can exist between activities. The semantics of this concept are based on the work discussed in the Background section. Thus, there are two subclasses:

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

– and the following set of slots:

- *source* and *target*: both slots are *Activities*, the idea being that these are the two activities which are interdependent.
- *type*: an enumeration, which indicates whether the relation is "soft" or "hard," 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 it will not affect whether they are successful or not.

Subclasses of *NegativeCoordination* include:

- *MutuallyExcludes*: an instance of this relationship will exist between two atomic activities iff:
  1. they both *Require* some resource *r*,
  2. the actual or scheduled usage of *r* by both activities overlaps; and
  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 scheduled, as they require access to a resource that cannot be shared. The *type* of this interdependency is therefore *hard*.
- *Impedes*: an instance of this relationship will exist between two *Atomic Activity*s iff:
  1. they both *Require* some resource *r*,
  2. the actual or scheduled usage of *r* by both activities overlaps; and
  3. *r* is shareable.

  The idea is thus that these two activities will impede one another though they will not necessarily prevent each other from succeeding. The *type* of this interdependency is *soft* as it need not necessarily be managed for the system to run effectively.

There is 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. Sub-classes of *FatalCoordination* include:

- *Disables*: one activity will disable another if the occurrence of it will definitively prevent the occurrence of the other. This is a *hard* interdependency.
- *ResourceContention*: an instance of this relationship will exist between two atomic activities 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. *ResourceContention* relationships are not required to be *hard* although, of course, they could be.

Sub-classes of *PositiveCoordination* are:

- *ConditionallyFeeds*: in such an interdependency, 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). This is a *hard* interdependency.
- *Enables*: the occurrence of activity A_1 is both necessary and sufficient for the occurrence of A_2. This is a *hard* interdependency.

- *Subsumes*: activity A_1 subsumes activity A_2 if A_1 contains all the activities of A_2. This is a *soft* interdependency.
- *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 *hard* interdependency.

Operational Relationships

In order to *resolve* a coordination relationship between two activities, it may be necessary to appeal to the *operational relationships* that exists between the agents that will carry them out. Intuitively, operational relationships exist between agents, and by understanding these relationships, it can help to resolve interdependencies. The main concept then 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 organization, 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.

COORDINATION RULES

The ontology provides a means of describing activities and the interdependencies that may exist between them. This knowledge can then be used to coordinate the various activities with one another. For this purpose, a number of rules were developed. For the sake of clarity, they are split into three groups:

- Rules to check activities;
- Rules to detect interdependencies between activities; and
- Rules to manage interdependencies between activities.

The sets of rules can be seen as building upon one another. The first set ensures that the descriptions of activities are complete and consistent. The second set then uses these consistent descriptions to identify any interdependencies that exist between activities. Finally, the third set takes the interdependencies identified and manages them accordingly.

Rules to Check Activities

These rules are used to check activities and detect any inconsistencies or omissions. Essentially they are used to capture some of the basic axiomatic properties of the ontology. This entails that whenever a new type of activity is added to the ontology it may be necessary to add some new rules to this set.

Rules to Check All Coordinable Activities

1. If an activity's latest start date is after or the same as its latest end date, then set the latest start date to be the latest end date—the expected duration.
2. If an activity's actual end date is not its actual start date + expected duration, then change the actual end date accordingly.
3. If an activity started before its earliest start date, then its status should be set to 'outOfBounds.'
4. If an activity started after its latest start date, then its status should be set to 'outOfBounds.'
5. If an activity started after its latest end date, then its status should be set to 'outOfBounds.'
6. If an activity ended after its latest end date, then its status should be set to 'outOfBounds.'
7. If an activity's earliest start date is after or the same as its latest end date, then its status should be set to 'outOfBounds.'
8. If an activity's earliest start date is after its latest start date, then its status should be set to 'outOfBounds.'

Rules to Check Composite Activities

1. If a composite activity does not have an enables or conditionally feeds interdependency with another activity then its actual start date should be that of the component activity with the earliest actual start date. (The expected duration is also modified accordingly.)
2. The actual end date of a composite activity should be that of the component activity with the latest actual end date. (The expected duration is also modified accordingly.) This rule assumes a pessimistic view for disjunctive activities, i.e., they always take the longest time possible.

Rules to Check Component Activities

1. If a component activity is part of a composite activity that enables or conditionally feeds interdependency with another activity then the actual start date of the component activity should not be earlier than the actual start date of its composite activity.
2. If a composite activity has status 'failed,' 'succeeded,' or is 'redundant,' then all sub-activities should have the status 'redundant.'
3. If the earliest start date of a component activity is before that of the composite activity then it is set to the latter.
4. If the latest end date of a component activity is after that of the composite activity then it is set to the latter.
5. If the latest start date of a component activity is after the latest end date of the composite activity then it is set to the latter.
6. If the latest start date of a component activity is before the earliest start date of the composite activity then its status is set to 'failed'.
7. If the expected duration of a component activity is greater than the difference between the earliest start date and latest end date of the composite activity then its status is set to 'failed.'

Rules to Check Conjunctive Activities

1. If all component activities of a conjunctive activity have status 'succeeded' then set the status of the conjunctive activity to 'succeeded.'
2. If any one of the component activities of a conjunctive activity has status 'failed' then set the status of the conjunctive activity to 'failed.'

Rules to Check Disjunctive Activities

1. If any one of the component activities of a disjunctive activity has status 'succeeded' then set the status of the disjunctive activity to 'succeeded.'
2. If all component activities of a disjunctive activity have status 'failed' then set the status of the disjunctive activity to 'failed.'

Rules to Detect Interdependencies Between Activities

These rules examine activities and infer new instances of interdependencies from them. Similar to the previous rules, this entails that whenever a new type of activity is added to the ontology new rules may need to be added to this set. If a new type of interdependency is added to the ontology then new detection rules will certainly be required.

The rules themselves are divided into two categories: those that detect positive interdependencies and those that detect negative interdependencies. Negative interdependencies tend to be more general, and as such, the rules to find them are readily applicable to a wide range of domains. Positive interdependencies on the other hand tend to be more domain specific, relying on particular properties of activities; hence, specific rules have to be written to find such interdependencies. The Evaluation section demonstrates how rules can be created tailored to a particular domain. The following rules are currently implemented to detect instances of the negative interdependencies *impedes* and *mutually excludes*:

1. If the end of a time slot for an activity overlaps the beginning of the time slot for another activity, and both activities require access to the same non-shareable resource, then assert an interdependency stating that the two activities are mutually exclusive. This rule is illustrated in Figure 2.
2. If the time slot for an activity is included in the time slot for another activity, and both activities require access to the same non-shareable resource, then assert that the two activities are mutually exclusive.
3. If the end of a time slot for an activity overlaps the beginning of the time slot for another activity, and both activities require access to the same shareable resource, then assert an interdependency stating that the two activities impede one another.
4. If the time slot for an activity is included in the time slot for another activity, and both activities require access to the same shareable resource, then assert that the two activities impede one another.

Rules to Manage Interdependencies Between Activities

These rules describe the coordination regime itself. Thus, different sets of rules can be used to provide different regimes, depending upon requirements.

if
(AtomicActivity A_1 (requires $resource_r$, actualStartDate SD_1, actualEndDate ED_1))
(AtomicActivity A_2 (requires $resource_r$, actualStartDate SD_2, actualEndDate ED_2))
($resource_r$ (shareable false))
($SD_2 > SD_1$)
($ED_2 > ED_1$)
($ED_1 > SD_2$)
then
(MutuallyExcludes (hasSource A_1, hasTarget A_2, hasDuration (ED_2-SD_1)))

FIGURE 2 Example rule to find an interdependency.

Each of the rules acts on an interdependency by modifying a coordinable activity accordingly. This modification consists of changing either the actual start date, the actual end date or the status of the activity. In the case of the start or end date being modified, the knowledge base is first queried to find an suitable new slot. Once an interdependency has been managed, it is removed along with all other unmanaged interdependencies involving the modified activity. This ensures that the knowledge base is left in a consistent state.

The coordination rules themselves are based upon operational relationships first and foremost. So, for example, if two agents related by a legal authority relationship request to carry out conflicting activities then the activity of the agent with the lower precedence is modified. A similar rule exists for when there is a contractual authority relationship. If the two agents are peers, or the same agent requests two conflicting activities, then the shortest activity is moved. This is one example of a coordination regime, although it could be readily substituted for another.

Additionally, it was determined that it would be appropriate to distinguish between hard and soft interdependencies so that hard interdependencies, which determine the successful execution of the system, are always handled before soft interdependencies, which only affect efficiency. For this reason, each rule is effectively replicated with the only difference occuring in the type of interdependency encountered. It is then possible to specify that the rules dealing with hard interdependencies have priority over those dealing with soft interdependencies.

The following rules have been implemented to provide a working example of a coordination regime:

1. If two activities are mutually exclusive, and they were requested by different agents—one of which has a legal authority over the other—then move the activity of the agent with lower precedence.
2. If two activities are mutually exclusive, and they were requested by different agents, one of which has a contractual authority over the other, then move the activity of the agent with lower precedence.
3. If two activities are mutually exclusive, and they were both requested by the same agent, or they were requested by different agents—neither of which has authority over the other (i.e., they are peers or no relationship between the two has been explicitly stated)—then move the activity with the smallest expected duration. If they are both of equal expected duration then move the activity that was requested last.
4. If two activities impede one another and they were requested by different agents, one of which has a legal authority over the other, then move the activity of the agent with lower precedence.[5]

5. If two activities impede one another, and they were requested by different agents—one of which has a contractual authority over the other—then move the activity of the agent with lower precedence.
6. If two activities impede one another, and they were both requested by the same agent or they were requested by different agents—neither of which has authority over the other (i.e., they are peers or no relationship between the two has been explicitly stated)—then move the activity with the smallest expected duration. If they are both of equal expected duration then move the activity that was requested last.
7. If an activity enables another activity then modify the latter activity so that it occurs after the activity that enables it.
8. If an activity conditionally feeds another activity then modify the latter activity so that it occurs after the activity that conditionally feeds it.
9. If an activity subsumes another activity then set the status of the subsumed activity to 'superfluous.'
10. If an activity that subsumes another activity succeeds then set the status of the subsumed activity to 'succeeded.' This only occurs if the subsumed activity still has the status 'superfluous' as it may have become redundant or be part of a composite activity that is redundant, failed, or succeeded.
11. If an activity that subsumes another activity fails then set the status of the subsumed activity to 'requested,' i.e., attempt to reschedule the subsumed activity. Again, this only occurs if the subsumed activity still has the status 'superfluous.'
12. If an activity that subsumes another activity becomes redundant then set the status of the subsumed activity to 'requested,' i.e., attempt to reschedule the subsumed activity. Once more, this only occurs if the subsumed activity still has the status 'superfluous.'

IMPLEMENTATION

The coordination ontology was implemented in the Web Ontology Language (OWL)[6] using Protégé 3.0,[7] and all of the rules were implemented in Jess (Friedman-Hill 2003). The JessTab[8] plug-in for Protégé was used to enable the ontology to be loaded into a Jess rule engine as a Protégé knowledge base. This effectively encapsulates the coordination engine (ontology, instances, rules, and rule engine) into a Java object.

A Web Services Resource Framework (WS-RF)[9] service was then implemented in Globus Toolkit 4 (GT4).[10] This service acts as a wrapper for the coordination engine and provides methods the following methods:

- *register resource*: used to register a new resource.
- *deregister resource*: used to deregister a resource.

- *request activity*: used to request a new activity.
- *withdraw activity*: used to withdraw an activity request should it no longer be required.
- *set status*: used by the requester of an activity to set the activity's status.
- *check entire schedule*: used to retrieve the entire list of activities for a particular resource.

Additionally, since Globus Toolkit 4 implements the WS-Notification specification[11] it was also possible to implement a publish-subscribe notification mechanism allowing agents to subscribe to receive messages detailing activity changes or the deregistration of resources. This asynchronous form of communication is essential for updating resource providers and requesters of changes which may impact upon them. The service is illustrated in Figure 3.

A graphical client application was also developed to allow for intuitive user interaction with the service when performing testing and evaluation. The client provides access to all of the service's API methods and automatically subscribes to receive all notification messages sent by the service. Using these messages, the client is able to build up a representation of the internal state of the service. The user can then access this information in the form of a Gantt chart representing either the entire list of resources and activities known to the service, or the list of activities for a specific resource. With this information on screen, the user is able to select an activity and view the interdependencies associated with it.

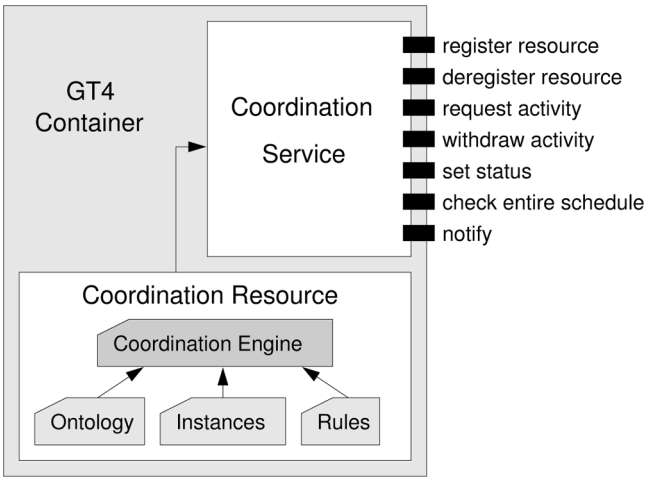

FIGURE 3 Architecture of the coordination service.

USE CASE

This section presents a sample scenario, taken from the domain of car insurance fraud, to which a centralized coordination mechanism could be applied to successfully coordinate a number of activities.

The scenario involves a number of insurance companies who wish to collaborate to discover whether the claims they receive are fraudulent. As such, a virtual organization (VO) is established and member insurance companies make their databases available to other members (though with many limitations). When one of the members then wishes to assess a claim it performs checks against a number of known fraud models. For example, the Berliner fraud model involves stealing a car and then crashing it into an insured car that is already damaged and claiming the damage from the insurance company of the stolen car. Other fraud models include the stolen cars model, the Saarland model and the Autobumser model. To detect whether one of these models is present, the insurance company will send a number of queries to the other insurance companies in the VO and then aggregate and analyze the results. Generally, the models should be checked in a specified order, and if one of the models is detected then the rest need not be checked.

The Ontology

To implement this scenario, the coordination ontology was extended as illustrated in Figure 4. Working from the bottom up, two new types of resource are included in the ontology:

- *CPU*: This represents a CPU that will be used to perform some processing task. The property *shareable* is set to true as multiple processing tasks may be performed simultaneously.
- *InsuranceCompanyBO*: This represents a database (back-office) of an insurance company. The property *shareable* is set to false as the back-office operations are intensive and the insurance companies wish to limit the number of operations that can be performed.

There are then two new sub-classes of AtomicActivity which use these resources (though an intermediate class *CarFraudActivity* is introduced for clarity):

- *Query*: These represent the queries that are performed on the insurance company databases, and as such the *requires* property must be an instance of InsuranceCompanyBO. They also have an additional slot *content* that details the content of the query.

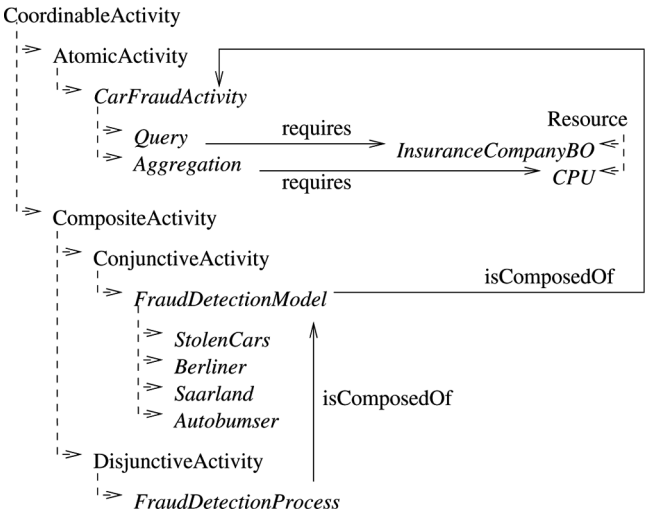

FIGURE 4 The extended ontology for the car fraud use case.

- *QueryAggregation*: These represent the aggregations of queries that are performed by insurance companies. As such the *requires* property must be an instance of CPU.

These activities are then used to compose *FraudDetectionModel* activities, which are a composite activity representing a fraud-detection model; as such, these activities have subclasses for each of the detection models (StolenCars, Berliner, Saarland and Autobumser). FraudDetectionModel is itself a sub-class of ConjunctiveActivity, as it is necessary for all of the queries and query aggregations within a particular model to complete successfully in order for that activity to complete successfully.

Finally, FraudDetectionModel activities are used to compose *FraudDetectionProcess* activities, which are also composite activities they, and represent the entire process undertaken by an insurance company checking for fraud. It is a sub-class of 'DisjunctiveActivity,' since if any of the component fraud detection model activities succeeds then the whole fraud detection process succeeds.

The Rules

With the new activities defined it was necessary to consider whether any new rules are also needed. Such rules may be required in any of the three categories and so the following process was observed.

Firstly, it may be necessary to define new rules to check the consistency and completeness of newly defined activities. Often, however, this will not

be the case, since new activities will extend existing activities and so the existing rules will also apply to these new activities.

Secondly, it is necessary to examine any interdependencies that may involve the new activities to determine whether any new rules are required for detecting them. As stated in the Rules section to detect interdependencies between activities, the rules to detect negative interdependencies are more generally applicable than those used to detect positive interdependencies, so it is unlikely that new rules will be required here, unless new types of negative interdependencies have also been defined in the ontology. It is more likely there will be positive interdependencies that are already classified in the ontology but that rely on the particular properties of domain specific activities. In these cases, specialized rules must be written to detect such interdependencies. Of course, new types of positive interdependencies may be defined as well, in which case rules will be needed to detect these too.

Finally, it is necessary to determine whether any new rules are required for managing interdependencies. Generally though, this will not be the case unless a new coordination regime is required.

Following this process, it was found that no new rules were necessary for checking activities or for managing interdependencies. Additionally, the following interdependencies would be detected by the existing detection rules:

- If two Query activities require the same InsuranceCompanyBO at the same time then they *mutually exclude* one another, since the InsuranceCompanyBO is non-shareable.
- If two QueryAggregation activities require the same CPU at the same time then they *impede* one another, since the CPU is shareable.

However, several new rules were required for detecting interdependencies:

- If a Query activity and a QueryAggregation activity are part of the same FraudDetectionModel then the Query *enables* the QueryAggregation.
- If two Query activities require the same InsuranceCompanyBO and have the same content then the activity with the earliest actual end date *subsumes* the other.
- If a StolenCars activity is part of the same FraudDetectionProcess as a Berliner or Saarland or Autobumser activity then the StolenCars activity *enables* the other.
- If a Berliner activity is part of the same FraudDetectionProcess as a Saarland or Autobumser activity then the Berliner activity *enables* the other.
- If a Saarland activity is part of the same FraudDetectionProcess as an Autobumser activity then the StolenCars activity *enables* the Autobumser activity.

EVALUATION

In initial testing of the system, the coordination service was found to detect and manage all of the expected interdependencies between activities. Given this, a test harness was developed based upon the client described in the Implementation section. This harness allows the user to set a number of variables from which it generates a series of activities to submit to the coordination service. As the test harness subscribes to all of the notifications sent by the coordination service, it has a complete view of the status of the coordination engine at all times. From this, it is able to determine if a newly submitted activity should have any interdependencies with an existing activity. It then checks for each new activity whether any interdependencies are detected and whether they are managed successfully.

The following factors were varied in the evaluation:

- The number of resource providers;
- The number of resource requesters;
- The number of resources;
- The number of activities to submit; and
- The number of interdependencies that should exist between submitted activities.

The following results were measured:

- The response time of the service;
- The number of interdependencies detected/resolved;
- The level of communication between the service and its users, i.e., the number of notifications sent in response to a service call. Typically, such notifications will be sent when activities are moved as a result of an interdependency being detected and resolved; and
- The number of activities that cannot be scheduled within the bounds of their earliest and latest start/end dates.

Number of Resources vs. Response Time

The first experiment performed was intended to identify how the system performed with a varying numbers of resources. A series of runs were performed during which a constant number of activities were submitted to the service, while the number of providers, requesters, and interdependencies were also kept constant.

The only variable was the number of resources being managed by the coordination service. The time taken for the coordination service to respond to requests for new activities was recorded. The results are detailed in Figure 5.

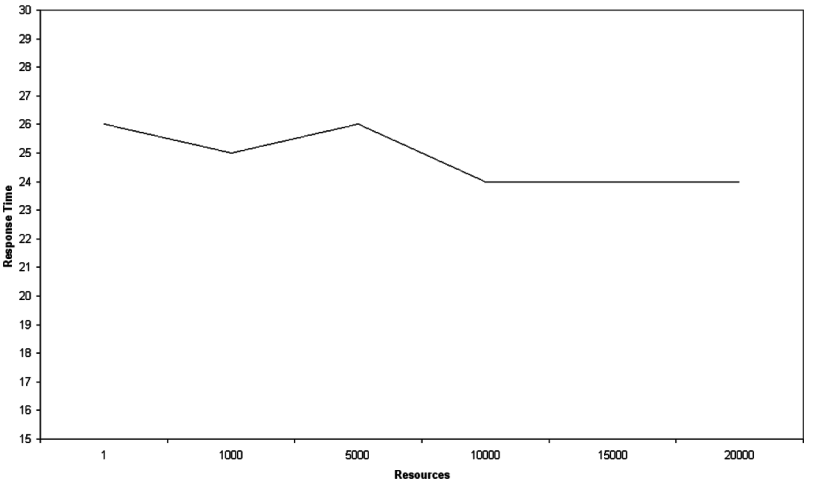

FIGURE 5 Number of resources vs. response time.

As can be seen from these results, the number of resources being managed by the coordination service had no discernible effect on the time taken to respond to service calls to add new activities. Only a difference of 2 ms (around 7.5%) existed between the response times of the service with 1 resource and that with 20,000 resources. Furthermore, the service managing 20,000 resources responded more rapidly than that managing a single resource.

The explanation for this is that for the service managing a single resource, all activities were specified as using that one resource, whereas for the service managing 20,000 resources, the activities were distributed over all of these resources. Hence, when a new activity is added to the service managing one resource, it takes longer to check that activity against the list of other activities already using that resource.

Number of Activities vs. Response Time

The next experiment performed was intended to identify how the system performed with a varying numbers of activities. A series of runs were performed during which a constant number of resources were managed by the service, while the number of providers, requesters, and interdependencies were also kept constant. For this experiment, the number of interdependencies was set to zero.

The only variable was the number of activities to add to the coordination service. The time taken for the coordination service to respond to requests for new activities was recorded. The results are detailed in Figure 6.

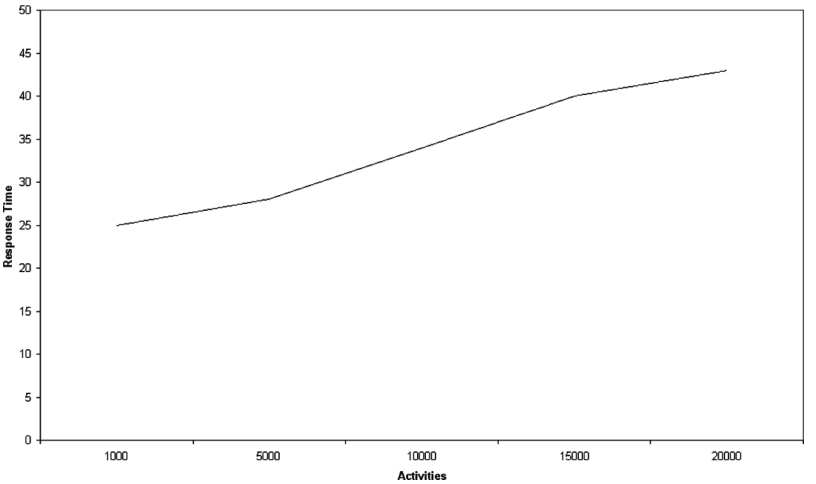

FIGURE 6 Number of activities vs. response time.

As can be seen from these results, the greater the number of activities being managed by the coordination service, the longer the service took to respond to service calls to add new activities. This relationship was found to be linear, i.e., as the number of activities added was increased, the response time increased proportionately.

This result was as expected, and it has a similar explanation as the result found in the previous experiment, i.e., with a larger number of activities there are more activities per resource, and hence, when a new activity is added to the service managing a resource, it takes longer to check that activity against the list of other activities already using that resource.

Number of Interdependencies vs. Response Time

Another experiment was carried out to identify how the system performed with a varying numbers of interdependencies. A series of runs were performed during which a constant number of resources were managed by the service, and a constant number of activities were requested using these resources. Additionally, the number of providers and requesters were also kept constant. The only variable was the number of interdependencies that should be detected.

The time taken for the coordination service to respond to requests for new activities was recorded. This was further divided into the time taken for the service to respond to activity requests when an interdependency was detected, and the time taken to respond when no interdependencies were

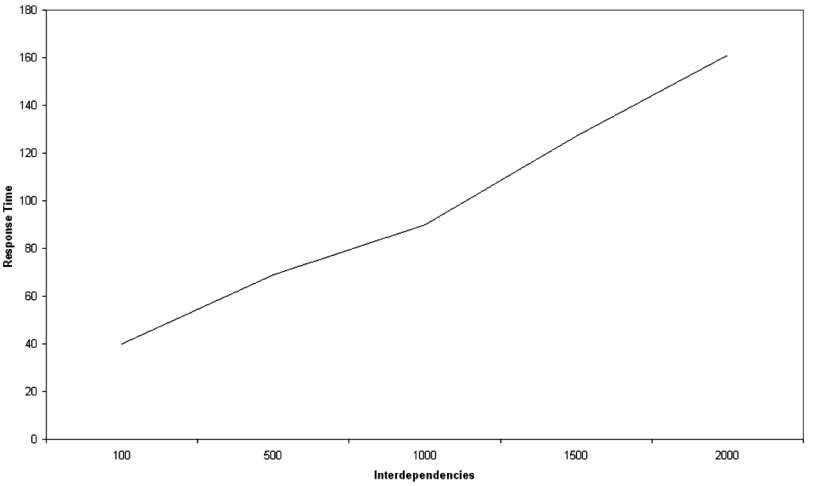

FIGURE 7 Number of interdependencies vs. response time.

detected. Furthermore, the number of notifications sent by the service, and the number of interdependencies detected and resolved were also measured. Finally, the number of activities that were moved as a result of an interdependency such that the start and end dates were now outside the prescribed limits (i.e., that were out of bounds) was also measured. The results are detailed below:

As can be seen from Figure 7, the greater the number of interdependencies between activities, the longer it takes for the service to respond to new activity requests. What is more, this relationship appears to be linear. This is to be expected as a call to add a new activity with an interdependency will take longer than one without since it takes time to manage the interdependency and move any activities appropriately.

However, as illustrated in Figures 8 and 9, the time taken to respond to activity requests that do not involve an interdependency increases at a much slower rate than the time taken to respond to activity requests with an interdependency. The explanation for this is that as the number of interdependencies increases, and hence, the proportion of interdependencies to activities increases, the greater the number of activities that need to be moved around and accommodated when managing the interdependencies. Similarly, this results in a greater proportion of activities that are moved outside of the start and end date limits, as illustrated in in Figure 10.

The most significant result of this experiment was that in each run the expected number of interdependencies was detected and resolved. Hence, the coordination service consistently identified and resolved all of the different types of interdependency with a 100% success rate.

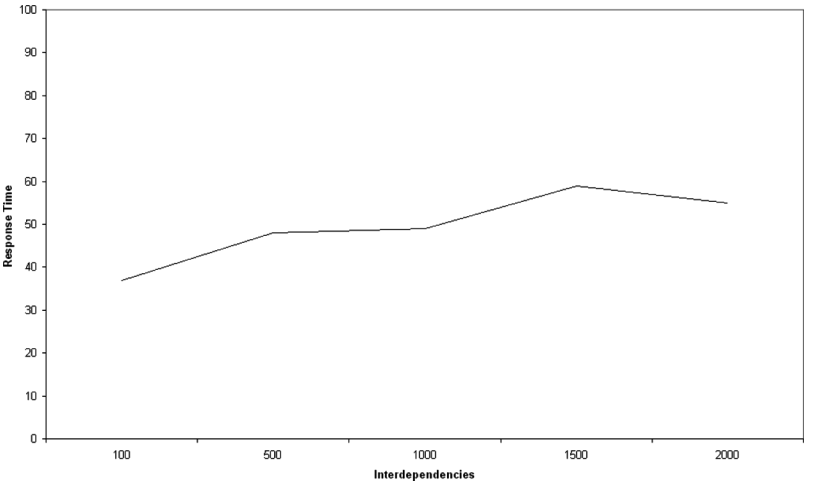

FIGURE 8 Response time for activities without interdependencies.

Discussion

The experiments demonstrate that the coordination service successfully detects and resolves all of the interdependencies. Furthermore, it is largely unaffected by the number of resources that it has to manage, while the response times increase linearly with the number of activities and interdependencies. Also, the proportion of activities that are moved outside of their limits is fairly low until the number of interdependencies approaches the number of activities. Of course, this assumes that activities are randomly

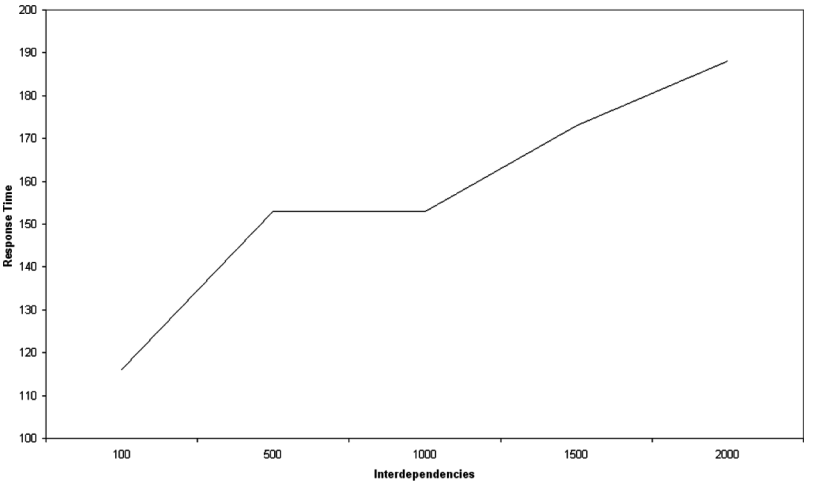

FIGURE 9 Response time for activities with interdependencies.

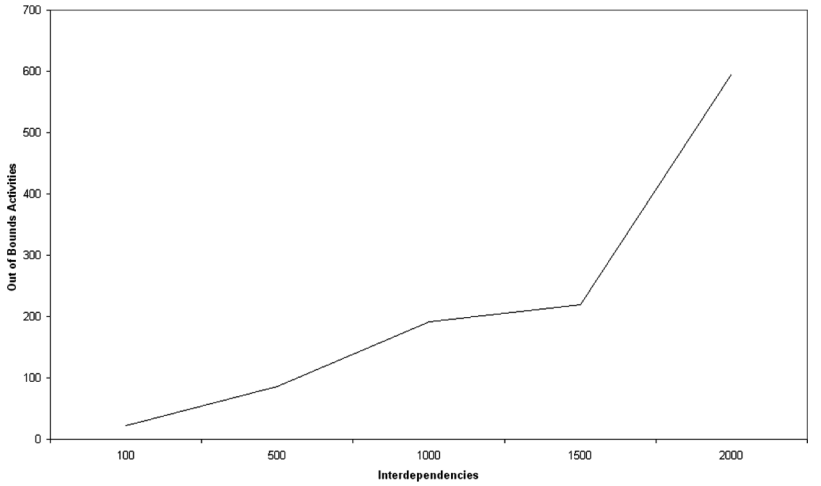

FIGURE 10 Number of interdependencies vs. number of 'outOfBounds' activities.

added to each resource, as they were in these experiments, and that the flexibility of activities is about 10 times the duration of the average activity (i.e., for an activity of duration 10 time units the flexibility will be 100 time units, or 45 each side of the start and end date). With greater flexibility, this proportion will decrease further to a minimum of 0 when no bounds are set for activities.

Several other improvements could be made to increase the efficiency of the coordination mechanism. For example, when an interdependency is detected and an activity is to be moved, the function to find a new slot for that activity currently performs a linear sort and search of all activities using the same resource (within a specified time range). This could be improved by implementing a binary search or interpolation search so as to improve the response time of the system. A number of other rules and functions could also be re-factored for greater efficiency should an industrial strength implementation of the system be required.

CONCLUSIONS & FUTURE WORK

The aim of this work is to provide coordination at run time rather than being hard-wired at design time. The current system demonstrates that the approach developed is a viable means of detecting and resolving interdependencies. In the simplest case, the approach can be used in place of a queue-based or prioritized scheduler. When the system becomes more complex, however, and there is a need to dynamically recognize and resolve interdependencies between activities, then traditional queue-based

approaches will fail, whereas the approach developed herein will still manage the resources effectively.

The main point of future is to apply the approach to a decentralized environment. To this end, it would be beneficial to implement the rules in a language such as the Semantic Web Rule Language (SWRL),[12] which would enable them to be encapsulated in the ontology thereby enabling the coordination mechanism to be more portable and exchangeable. However, SWRL currently has a number of limitations that prevent many of the rules being directly translated; for example, it only supports the conjunction of atoms, there is no support for negation, and there are no explicit quantifiers but instead implicit universal quantification for all variables. Rule engines for SWRL also suffer from limitations. For example, Bossam[13] has no built-in support for math functions and string handling; it has the facility to retract facts (when they are no longer true); and it has no support for the boolean datatype.

A related point of future work is the implementation of a number of alternative coordination regimes. These could be collected as libraries so that coordination regimes could be substituted for one another dependent on the environment and requirements. A useful experiment would then be to determine how easily these different regimes could be swapped.

Another point of future work consists of examining the definition of *AtomicActivity* alongside related representations such as those used by BPEL4WS, to see whether it can be made more precise. The key point here though will be that made by Singh, i.e., the more detailed the description of tasks becomes, the better the coordination mechanisms that can be designed but the less widely applicable those mechanisms will be. The mechanism so far developed has been demonstrated to be widely and readily applicable, and it would be highly desirable to maintain this level of applicability.

NOTES

1. http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os/wstx-wscoor-1.2-spec-os.html
2. http://www.w3.org/Submission/OWL-S/
3. http://www.wsmo.org/
4. http://www.w3.org/Submission/OWL-S/
5. Note that it is possible to simply allow both activities to proceed as scheduled, but with their durations increased by a particular factor. This would be an example of an alternative coordination regime.
6. http://www.w3.org/TR/owl-guide/
7. http://protege.stanford.edu/
8. http://www.ida.liu.se/~her/JessTab/
9. http://www.globus.org/wsrf/
10. http://www.globus.org/toolkit/
11. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn
12. http://www.w3.org/Submission/SWRL/
13. http://bossam.wordpress.com/about-bossam/

REFERENCES

Ben-Ari, M. 1990. *Principles of concurrent and distributed programming*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Decker, K., and V. R. Lesser. 1995. Designing a family of coordination algorithms. In *Proceedings of 1st International Conference on MultiAgent Systems (ICMAS-95)*. San Francisco (CA, USA). 73–80. Menlo Park, CA, USA: AAAI Press.

Durfee, E. 1993. Organizations, plans, and schedules: An interdisciplinary perspective on coordinating ai systems. *Journal of Intelligent Systems, Special Issue on the Social Context of Intelligent Systems* 3(2–4): 2–4.

Durfee, E. H. 1988. *Coordination of distributed problem solvers*. Boston, MA: Kluwer Academic Publishers.

Friedman-Hill, E. 2003. *Jess in action: Java rule-based systems*. Greenwich, CT, USA: Manning Publications Co.

Genesereth, M. R. 1991. Knowledge interchange format. In *Principles of knowledge representation and reasoning, KR'91, Proceedings of the second conference*, eds. J. Allen, R. Fikes, and E. Sandewell, 599–600. San Francisco, CA, USA: Morgan Kaufmann Publisher.

Malone, T. W., and K. Crowston. 1994. The interdisciplinary study of coordination. *ACM Computing Surveys* 26 (1): 87–119.

Moyaux, T., B. Lithgow-Smith, S. Paurobally, V. Tamma, and M. Wooldridge. 2006. Towards service-oriented ontology-based coordination. In *Proceedings of 4th International Conference on Web Services (ICWS 2006)*: 265–274, Chicago, IL, USA: IEEE Computer Society.

Singh, M. 2005. Formal specification and enactment. In *Service-oriented computing: Semantics, processes, agents*, eds. M. Singh and M. Huhns, 281–303. Chichester, UK: Wiley.

Singh, M. P. 1998. A customizable coordination service for autonomous agents. In *Intelligent Agents IV (LNAI Volume 1365)*, eds. M. P. Singh, A. Rao, and M. J. Wooldridge, 93–106. Berlin, Germany: Springer-Verlag.

Singh, M. P. 2000. Synthesizing coordination requirements for heterogeneous autonomous agents. *Autonomous Agents and Multi-Agent Systems* 3:107–132.

Studer, R., V. Benjamins, and D. Fensel. 1998. Knowledge engineering, principles and methods. *Data and Knowledge Engineering* 25 (1–2): 161–197.

Tamma, V., C. van Aart, T. Moyaux, S. Paurobally, B. Lithgow-Smith, and M. Wooldridge. 2005. An ontological framework for dynamic coordination. In *Proc. of 4th Int. Semantic Web Conf. (ISWC 2005)*, Lecture Notes in Computer Science, Galway (Ireland), 638–652. Berlin: Springer.

von Martial, F. 1990. *Interactions among autonomous planning agents*. Amsterdam, The Netherlands: Elsevier.

von Martial, F. 1992. *Coordinating plans of autonomous agents*. New York: Springer-Verlag.