Knowledge-Based Coordination with a Reliable Semantic Subscription Mechanism

Martin Murth and eva Kühn
Vienna University of Technology, Institute of Computer Languages
Space Based Computing Group
Argentinierstraße 8, 1040 Vienna, Austria
{mm,eva}@complang.tuwien.ac.at

ABSTRACT
Semantic technologies promise to solve challenging problems of current enterprise information systems, e.g., the integration of heterogeneous information clients and the evaluation of complex data dependencies. As many of these problems also apply to coordination applications, recent research initiatives have proposed to integrate coordination models with semantic technology. In this paper, we present such an integrated coordination model which combines logic-based reasoning with a reliable semantic subscription mechanism. We present a formal definition of the model's behavioral semantics and investigate the added value of using semantic technologies. Finally, we draw first conclusions about the practical applicability of the proposed approach based on performance benchmarks of a prototype implementation.

Keywords
Knowledge-based coordination, coordination model, logic-based reasoning, reliable subscription mechanism.

1. INTRODUCTION
In the past years, semantic technologies have reached a certain level of maturity and have been adopted by various tools and platforms. Recent initiatives in the research field of the semantic web such as TripCom [28] and TSC [7] have proposed to build infrastructures for sharing semantic data by employing concepts of space based coordination systems [16]. Fensel et al. [6] argue that the employment of the publish-and-read paradigm allows for improved scalability of the infrastructure and better decoupling of client-to-client communication. It has also been investigated in several works (cf. [24]) how space based coordination systems can be extended with semantic web technology. In this paper, we build upon these works and describe a concrete, knowledge-based coordination model that integrates a logic-based reasoning engine with a reliable semantic subscription mechanism. We depict the conceptual interaction model and formalize the model's behavioral semantics. Furthermore, we outline how the coordination model benefits from semantic technologies and evaluate its practical applicability based on performance benchmarks of a prototype implementation.

SAC'09, March 8-12, 2009, Honolulu, Hawaii, U.S.A.

Coordination applications are frequently confronted with heterogeneous IT environments. Logic-based reasoning systems, as a core constituent of semantic technologies, promise to provide a useful means for reducing complexity of managing heterogeneity [3]. They help in coping with incomplete and inconsistent coordination data and ease the integration of different terminologies. Furthermore, employing logic-based reasoning on coordination data allows detecting complex dependencies between the coordinated clients, which would be hard to detect with classic coordination systems. We illustrate this by the following example:

The coordination of agents of an open agent system relies on the data published to a shared data space. In the example, we consider three properties of an agent:

- **Name**: the name of the agent
- **Owner**: either enterprise "internal" or "external"
- **Forwards work orders to**: those agents that it can delegate work orders to

Table 1 shows the current contents of the shared data space. The question marks denote information that is not (yet) available in the system.

Table 1. Shared coordination data in an agent system

Name	Owner	Forwards work orders to
A	?	B, C
B	internal	C
C	?	D
D	external	?

As a general policy, we define that work orders need to be forwarded to an internal agent, which verifies the work order and then forwards it to an external agent. We now assume that agent A only wants to start producing work orders as soon as it can be sure that it can forward them to an internal agent which itself forwards them to an external agent. It therefore wants to be notified by the coordination system about the existence of the forwarding path "A → internal agent → external agent".

Even with space based coordination systems that offer more comprehensive interfaces to coordination data than Linda-based tuplespaces [9] (e.g. SQL-based, cf. [10][17]), it is often difficult to describe such conditions. Furthermore, in most coordination systems, processing the available data would lead to the following interpretation: Agent A forwards work orders to the internal agent B, and agent B forwards them to agent C. Since we do not know whether agent C is internal or external, this path is not the one we are looking for. Agent A also forwards work units to agent C, but again, we do not know whether agent C is internal or not and, thus, this is also not the path we are looking for. Consequently,

the system would not notify client A about the existence of a suitable forwarding path.

However, with logic-based reasoning, we can derive some more facts about the coordination scenario. The above example describes all possible settings for which the known facts are true. These settings can be divided into two distinct groups: those in which C is an internal agent and those in which it is an external agent. In the first group of settings, agent A can forward work orders to the internal agent C which forwards them to the external agent D. This already represents a path that we are looking for. In the second group of settings, agent A can forward its work orders to the internal agent B which then forwards them to the external agent C. Consequently, we already know that there is a suitable forwarding path, even if it is not yet decidable which path it will be. For agent A, this information may already be sufficient to start producing work orders.

Generally, the employment of logic-based reasoning allows for the description and processing of data dependencies for which conventional algorithmic solutions would typically be complex or not intuitive.

1.1 Related Work

An approach that combines similar concepts is taken by RDF-based publish/subscribe systems (GToPSS [26], OPS [29]). These systems extend the subscription mechanism of traditional publish/subscribe systems with semantic matching capabilities and provide a useful improvement of metadata and content-based subscription. However, they still implement a purely store-and-forward [13] based interaction mechanism. In contrast, our approach allows reacting to situations that can be derived from the entire (shared) knowledge about the coordination scenario.

Semantic tuplespaces [24] (sTuples [14], TSC [7], Semantic Web Spaces [23], TripCom [28]) represent a coordination approach that also builds upon a shared knowledge base. The knowledge is usually stored as RDF triples [19] in the tuplespace and most implementations offer either a Linda-based or a publish/subscribe based interface to access the stored data.

Semantic tuplespaces with a Linda-based interface (sTuples, TSC, Semantic Web Spaces, TripCom) allow for retrieving data using an extended template matching mechanism. Retrieval operations can be invoked in a blocking and a non-blocking mode and are used to request both explicit and implicit (i.e. inferred) data from the knowledge base. However, a core characteristic of Linda-based systems is that they do not make any guaranties about which, when, or if at all clients are notified about matching data. While this allows for several optimizations of distributed implementations [5], in industrial applications, guaranteed subscriber notification is often a core requirement.

Taking this into account, some semantic tuplespaces also implement publish/subscribe based interfaces (TSC, TripCom) that guarantee the delivery of messages according to a certain delivery policy. These systems verify at each publication operation, whether the published data satisfies some semantically defined matching conditions. Conceptually, this resembles the functionality of RDF-based publish/subscribe systems, but technically, the systems are targeted at different types of applications. While semantic tuplespaces try to optimize evaluation of big knowledge bases, RDF-based publish/subscribe system are designed for high message throughput.

The semantic coordination model presented in this paper combines the two interface types. General queries can be registered at a semantic data space and subscribers are reliably notified about all results of these queries. In addition to this, we provide a formal definition of the system's interface and also consider the semantics of a delete operation. Both aspects have not been addressed in detail in other works.

2. KNOWLEDGE BASED COORDINATION

The proposed model shall allow for designing, implementing, and managing complex and heterogeneous coordination applications by leveraging the reasoning capabilities of logic-based reasoning systems. Software clients (the *coordinated entities*) publish coordination-relevant data at the coordination system and subscribe to particular coordination events. In contrast to conventional publish/subscribe systems, the published data is not directly forwarded to the subscribers but is, in a first step, added to the system's knowledge base. A reasoning engine then tries to infer relevant coordination information from the new state of the knowledge base and delivers the found results to the subscribers. The reasoning process also includes case-based reasoning, which allows finding coordination dependencies like the one described in Section 1, and the validation of constraints on the coordinating data.

Subscriptions are formulated as concept definitions (the *coordination law*). Concept definitions define under which conditions "something" can be considered an instance of the respective concept. E.g., the concept *LazyAgent* can be defined as (i) an agent that (ii) forwards work orders to an internal agent which (iii) itself forwards work orders to an external agent. *Agent* and *Work Order* might themselves be concept definitions and *forwardsTo* could define a relation between two agents. Whenever the system can guarantee that the defined conditions hold for a certain part of the knowledge base, the subscriber is notified about it and is provided with the according data.

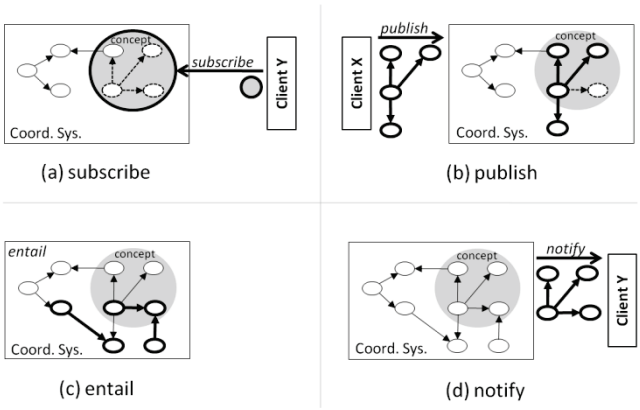

Figure 1. Interactions with the coordination system

Figure 1 illustrates the four main interaction steps:

(a) A software client Y subscribes to the coordination system to receive notifications about a certain knowledge domain. It therefore provides a concept definition that precisely defines the knowledge it is interested in.

(b) A software client X publishes new knowledge at the coordination system. The provided knowledge is added to the system's knowledge base (explicit knowledge).

(c) The coordination system employs a reasoning engine to infer new knowledge from the current state of the knowledge base (implicit knowledge).
(d) The coordination system evaluates whether there is knowledge available that matches a subscribed concept definition. Subscribers are notified about those parts of the knowledge base that match the concept definition and which they have not already been notified about before.

A software client can also delete data from the knowledge base of the coordination system. When a delete operation is executed, the knowledge base is re-evaluated and knowledge that can no longer be inferred from the explicit knowledge is removed. When particular knowledge can be inferred again after some subsequent publication operation, also the subscribers are notified about the newly inferred knowledge again.

The rules that are employed to infer new knowledge are themselves stored in the knowledge base. Consequently, software systems can change these rules during system runtime by simply adding and removing rules to and from the knowledge base.

3. TOWARDS A FORMAL MODEL

In this section, we provide a formal definition of the proposed system's behaviour by combining concepts from description logics and temporal propositional logic. In the following, we will briefly introduce the concepts employed. A more detailed discussion of formal frameworks for the specification and validation of concurrent systems can be found in [8].

Description logics (DL) [3] are mostly defined as subsets (and sometimes extensions) of first order logic (FOL) [2] and constitute the theoretical foundation for several ontology languages applied in the semantic web area. They are used to describe knowledge in the form of concepts and roles and instances thereof. The clearly defined semantics of DL allow for deriving implicit consequences of explicitly represented knowledge and for validating whether hypotheses may hold in the described model. In this section, we will use the following constructs of DL:

- **Definitions**: Concept and role definitions are used to define the terminology of a domain. E.g., *Agent* would be a concept, *forwardsTo* would be a role that defines a relation between two agents.
- **Assertions**: Assertions are used to define extensional knowledge about the domain. E.g., *Agent(A)* asserts agent *A* being an instance of the concept *Agent*. *A* is also referred to as *individual.*
- **Knowledge base**: A knowledge base Σ consists of a set of definitions and a set of assertions describing a certain knowledge domain.
- **Entailment of concept assertions**: A knowledge base Σ entails a concept assertion $C(a)$ (we write $\Sigma \vDash C(a)$), if every model of Σ satisfies $C(a)$. E.g., $\Sigma \vDash IntAgent(B)$ states that the knowledge base entails agent *B* being an internal agent.

Temporal propositional logic (TPL) [15] allows for defining propositions whose truth and falsity may depend on time. TPL is often used to specify the behaviour of concurrent processes in distributed systems. The behaviour of the system is described as a *trace*, i.e. as a sequence of global system states $\sigma=s_0, s_1, s_2, s_3$ ..., and some conditions that must hold for these states. An atomic predicate *P* is true for a subtrace $\sigma_{|i} = s_i, s_{i+1}, s_{i+2}$, ..., if and only if it is true for the first state of the subtrace. A *specification* of a system is defined as a set of traces. A system satisfies a specification, if it only exhibits traces contained in this set of traces [8].

Besides the common logical operators, we will use the following temporal operators for temporal formulas:

□ "*always*": $\Box\varphi$ is true for trace σ iff for all $i \geq 0$, φ is true for the trace $\sigma_{|i}$.

◇ "*eventually*": $\Diamond\varphi$ is true for trace σ iff there exists an $i \geq 0$, such that φ is true for the trace $\sigma_{|i}$.

○ "*next*": $\bigcirc\varphi$ is true for trace σ iff φ is true for the trace $\sigma_{|1}$.

$\mathcal{U}$ "*until*": $\varphi\ \mathcal{U}\ \psi$ is true for trace σ iff there exists a $j \geq 0$, such that for all $0 \leq i < j$, φ is true for the trace $\sigma_{|i}$ and ψ is true for the trace $\sigma_{|j}$.

We now specify the behaviour of the system by defining (i) the state changes for input and output operations and (ii) conditions to hold for valid traces (following the approach described in [22]).

Our system exhibits five interface operations: four input operations *pub, del, sub, unsub*, and one output operation *notify*. These operations take parameters from different domains: The letter *K* represents knowledge in the form of a concept definition, a role definition, a concept assertion, or a role assertion. *C* represents a valid concept definition and *a* refers to an individual described in the knowledge base. The state of the system is defined by the following variables:

- A common *knowledge base* Σ containing all currently available explicit knowledge about the coordination scenario.
- A set S_x for every coordination entity *X* of *active subscriptions*, i.e. concept definitions which the entity has subscribed to and not unsubscribed from yet.

The state changes caused by every interface operation are defined in Table 2.

Table 2. Changes of the state variables caused by interface operations

$pub(K)$	$\Sigma' = \Sigma \cup \{K\}$
$del(K)$	$\Sigma' = \Sigma \setminus \{K\}$
$sub(X, C)$	$S_x' = S_x \cup \{C\}$
$unsub(X, C)$	$S_x' = S_x \setminus \{C\}$
$notify(X, C, a)$	-

When an entity publishes knowledge *K*, then this knowledge is added to the knowledge base Σ of the coordination system. The delete operation removes the knowledge from the knowledge base. When an entity *X* subscribes to be notified about changes of a concept, then the concept definition *C* is added to the component's active subscriptions, and it is removed again, when the component unsubscribes from it. The notify operation represents the notification of a component *X* about an individual *a* satisfying the concept *C*.

We specify the behaviour of the system by a safety and a liveness condition [1]. The *safety condition* guarantees that nothing "bad" will happen, i.e. it specifies under which conditions operations may be raised. The *liveness condition* requires that something

"good" will eventually happen, i.e. it specifies a condition and a system state that has to be reached if this condition is fulfilled.

(Safety) **(Def. 1)**

$$\square[notify(X,C,a) \Longrightarrow C \in S_x \\ \wedge \Sigma \models C(a) \\ \wedge \bigcirc(\neg notify(X,C,a)\ \mathcal{U}\ \Sigma \not\models C(a))]$$

(Liveness) **(Def. 2)**

$$\square\left[\square(C \in S_X) \Longrightarrow \left[\Sigma \not\models C(a) \wedge \bigcirc\left(\Sigma \models C(a)\right) \right.\right. \\ \left.\left. \Longrightarrow \bigcirc\Diamond notify(X,C,a)\right]\right]$$

The safety condition (Def. 1) states that whenever a notification is delivered, then

- the receiver of the notification must have an active subscription for the respective concept,
- the knowledge base Σ must entail the delivered individual being an instance of the subscribed concept, and
- the same notification must not be delivered twice, unless the knowledge base has changed in such a way that the concept assertion could not be entailed for some time and was then entailed again.

The latter condition implicitly specifies how the system has to handle delete operations. In the previous section, we defined that subscribers are notified about the *addition* of knowledge. We therefore specify that an entity is notified about a newly entailed assertion, even if it was already possible to entail the same assertion at an earlier state of the knowledge base.

A system that only produces traces that satisfy the safety condition already guarantees that it is always in a consistent state. However, there is no guarantee that it does anything at all. It therefore also has to satisfy the liveness condition.

The liveness condition (Def. 2) states that whenever an entity *X* is subscribed to a concept definition *C* and never unsubscribes from it, then

- if there is a system state that does not entail a particular individual *a* being an instance of the concept *C*, and
- which is followed by a state that does entail this concept assertion,
- then (at this next state) a notification must eventually be sent to the subscriber.

Thus, the liveness condition guarantees that all subscribers are notified about newly entailed individuals of the subscribed concept.

Having defined the semantics of the interface operations, we can now derive several consequences for a system implementation:

First, the specification does not prescribe *when* a notification has to be delivered. The liveness condition defines that any finite processing delay is acceptable. Moreover, it only affects subscriptions that remain registered forever. If a subscription would be unregistered at some time, the system does not need to notify the accordant entity about any events at all. While this may seem problematic at the first glance, the specification is precise enough to unambiguously define the behaviour of a system implementation. Since an implementation cannot know whether a subscriber will eventually unsubscribe, it has to prepare for the "worst case" (i.e., the subscription is never unregistered) and start to deliver notifications at some time in order to stay compliant with the specification.

Second, the current specification does not prescribe any ordering of notifications. Components could receive notifications in an order different from the order of the operations they resulted from. However, the specification prescribes to deliver notifications as long as they are entailed by the knowledge base. In general, this would require the system to notify the subscriber immediately after the entailment of new knowledge, since it cannot be sure that an assertion can still be entailed at the next system state. In monotonic knowledge bases, however, it is guaranteed that the addition of new knowledge does not invalidate previously entailed knowledge. Thus, coordination systems that employ monotonic knowledge bases will only need to deliver notifications right before any deletion operations are executed.

4. EVALUATION

In this section, we discuss conceptual advantages and disadvantages of the presented coordination model (Section 4.1) and draw first conclusions about its practical applicability based on performance benchmarks of our prototype implementation (Section 4.2).

4.1 Discussion

Expressiveness of the coordination language: The precisely defined semantics and the expressiveness of DL probably represent the main advantages of the proposed coordination model. The use of logic-based reasoning allows detecting and reacting to complex dependencies between the coordinated systems which are not explicitly modelled in the scenario but which can be inferred from the provided data. The dependencies in the example of Section 1 can be easily described with the following simple concept definition:

$$LazyAgent \equiv \exists fwdsTo.(IntAgent \sqcap \exists fwdsTo.ExtAgent)$$

The concept *LazyAgent* is defined to have a forwarding path to "something" that is an *IntAgent* and has a forwarding path to an *ExtAgent*. In contrast, describing the same dependency with a query on a database or an event stream would typically require nested subqueries, special treatment of null values, constraint evaluation and several case differentiations. Further examples for dependencies (concepts) that cannot or not intuitively be expressed with conventional mechanisms are *concept disjointness*, *value restrictions*, *complement concepts*, *transitive properties*, and the definition of *equality* and *subsumption* of concepts. Nonetheless, there are also aspects of coordination for which there are no special constructs available in most DL-based reasoning systems. Time and time-based relations, for instance, usually have to be modelled explicitly.

Reliable notification delivery can also be considered to add to the expressiveness of the subscription mechanism. Without this guarantee, clients would need to implement their own protocol on top of the provided coordination primitives to ensure that they receive all notifications (like, e.g., numbering patterns in Linda; cf. [6]).

Interoperability and extensibility: Equality and subsumption relations can be employed to define precise and consistent mappings between two terminologies [4]. Since these mappings are also part of the knowledge base, clients can provide their own mappings by simply publishing them to the coordination system. The reasoning engine then tries to infer new knowledge (i.e., knowledge represented in the new terminology). Clients can

define subscriptions using their own terminology and will also receive notifications in their own terminology. When new knowledge is published, the reasoning process automatically translates it to all provided terminologies. Thus, the system serves multiple terminologies at the same time. The use of multiple mappings is only limited by the increased reasoning effort that is added with each new mapping.

Adaptations and extensions of coordination applications can be realized in two steps. First, the new terminology is published to the coordination system. Both the old and the new terminology are run in parallel and the client systems can successively adopt the new terminology. Second, as soon as all clients are using the new terminology, the old terminology is deleted from the knowledge base and the reasoning process removes all knowledge that is represented in this old terminology. All adaptations can be done dynamically at system runtime.

Performance: Generally, there is a trade-off between the expressiveness of the description language and the complexity and processing effort of reasoning. Although the worst case runtime complexity of many reasoning algorithms is exponential, it has been shown that they can efficiently solve many real-world reasoning problems (cf. [11]). However, a coordination system not only executes a single evaluation of a knowledge base but rather needs to continuously modify, reason about, and query the knowledge base. The benchmarks in Section 4.2 will show for which kinds of coordination applications our prototype implementation already exhibits good performance and for which ones further optimizations will be required.

Logic-based reasoning also allows for optimizations of the subscription mechanism by employing *satisfiability checking*, a core reasoning task of reasoning engines that verifies whether it is generally possible to find instances of a particular concept[1]. If the concept of a new subscription is not satisfiable with respect to other concepts and assertions in the knowledge base, then there will be no matching individuals until concepts or assertions are modified. For monotonic knowledge bases, instance checking for the according subscriptions can thus be suspended until the next delete operation.

Data consistency: In description logics, a knowledge base is considered to be *consistent* if it only contains satisfiable concepts and assertions. As a first approach to improving quality of coordination data, satisfiability checking could be employed to detect or even prevent the publication of data that would lead to an inconsistent knowledge base. E.g., asserting *IntAgent*(C) and *ExtAgent*(C) could be detected as inconsistency since the both concepts are defined to be disjunctive.

Furthermore, *subsumption checking*, a reasoning task that verifies whether one concept will always represent a subset of another, can be employed to find redundant data and concept definitions.

An issue that will require further investigation is whether current reasoning engines can do satisfiability and subsumption checking efficiently enough to allow for integration with the interface operations of a coordination system without significantly reducing the system's responsiveness.

Integration with other systems: The proposed interaction mechanism allows for a simple integration with conventional publish/subscribe and event based systems. Messaging middleware [13] could be used to reliably exchange data with other information systems and event stream processors [18] could be employed to find time-based relations between notification, publication and deletion events.

[1] A concept C is said to be *satisfiable*, if there exists a model such that the set of individuals of C is nonempty.

4.2 SENS Prototype

The prototypical implementation of the coordination system SENS (Semantic Event Notification Service) [21] employs OWLIM 2.9 [25] as its reasoning and query engine and implements the interface defined in Section 3. The prototype is implemented in Java and can currently be accessed in-memory or via an RMI adapter. Coordination data is represented in RDF [19] and the terminology is defined with the ontology language OWL [20].

In the following, we briefly introduce the system's API primitives (Listing 1) and present performance benchmarks of the prototype. For the measurements, we employed the LUBM benchmark [11], a widely used framework for benchmarking semantic data repositories. The framework generates test data about universities, students, professors, courses, etc. of different sizes ranging from LUBM(1) ≈ 10^5 to LUBM(10) ≈ 10^6 explicit statements in our test scenario. It also defines an ontology that allows for entailing about the same amount of additional implicit statements. Furthermore, the framework defines 14 queries for the evaluation of different properties of the querying capabilities of semantic repositories. In the test scenario, we translated these queries to concept definitions and registered them at SENS. E.g., a query that returns all students that are registered to a particular course is translated to a subscription that notifies the subscriber each time a new instance of the respective concept is entailed.

Listing 1. SENS API (Java)

```
public interface SENS {

    // publishes knowledge in the form of RDF triples to the knowledge base
    void publish(Graph graph);

    // subscribes for all individuals being instances of the concept referenced
    // by the provided RDF resource
    SubscriptionID subscribe(Subscriber s, Resource concept);

    // subscribes for all individuals being instances of the concept referenced
    // by the provided RDF resource and matching the filter pattern
    SubscriptionID subscribe(Subscriber s, Resource concept,
                             GraphPattern pattern);

    // removes a subscription
    void unsubscribe(Subscriber s, SubscriptionID id);

    // deletes all triples that match the given statement pattern
    void delete(StatementPattern pattern);

}
```

publish: adds knowledge in form of a graph data structure to the system's knowledge base. Concept and role definitions as well as concept and role assertions are added in the form of RDF statements.

The prototype always keeps the order of incoming requests and groups requests of the same type. Subscriptions are evaluated each time a group of requests has been processed. This way, the prototype adheres to the defined API semantics and at the same time allows for high throughput of publication operations. Figure 2 shows the publication performance for concept and role assertions for different sizes of the knowledge base. In the test scenario, all 14 concept definitions were registered at SENS.

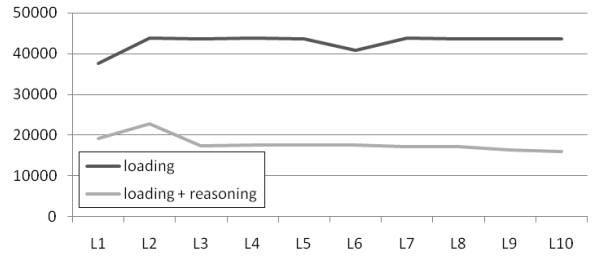

Figure 2. Publication performance (triples/sec) for concept and role assertions for different sizes of the knowledge base[2]

The upper graph shows the publication rate for a knowledge base without inference of implicit knowledge. The lower graph shows how the publication rate is affected by the reasoning process. Although performance is reduced by 45-55%, an almost constant publication rate of about 20 000 triples per second still allows for efficient addition of new knowledge.

Figure 3 shows the time required for publishing new concept definitions at SENS for the knowledge base sizes LUBM (1,5,10).

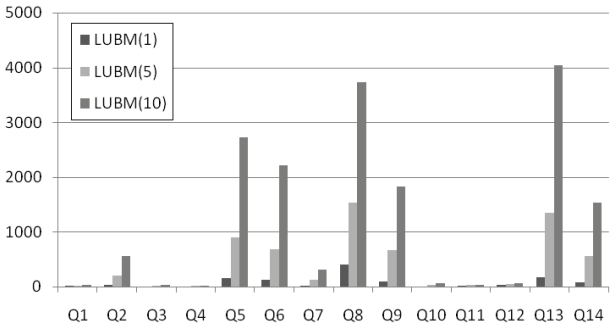

Figure 3. Concept publication times (ms) for LUBM queries

The current implementation reasons about the entire knowledge base right after the publication of a new concept. While this increases the publication time, it significantly reduces the time required for subsequent notification processing. Generally, the publication times mainly depend on the complexity of the new concept and on how many additional statements are entailed. Publication times of up to 4 seconds for LUBM(10) are acceptable, if adaptations of the terminology and coordination laws are only required for maintenance and advancements of a coordination application. In contrast, coordination applications with dynamically changing terminologies or coordination laws will require essentially lower concept publication times. With the current version of the prototype, such publication times can only be achieved for knowledge bases with up to ~10^5 statements.

subscribe: registers a subscriber to be notified about newly entailed instances of a concept that is referenced by an RDF resource. The subscribe operation takes an optional parameter of the type GraphPattern which can be used to define properties of an individual that are included in a notification. As these properties are considered mandatory, the graph pattern can also be used as an additional notification filter. Although SENS does not notify subscribers about filtered notifications, we formally interpret these notifications as delivered but unhandled by the client. This way, we ensure that the semantics of the system are not affected by the filter mechanism.

Figure 4 presents the average time required to notify subscribers about new instances of a concept when new statements are published. This includes reasoning about the knowledge base and checking whether an instance has already been entailed and delivered to the subscribers before.

Even at a knowledge base size of more than 3 million statements, the notification about new instances takes less than 80ms for all subscriptions except of Q6, Q9, and Q14. The higher processing times of these subscriptions result from the high number of instances that are recognized to be instances of the given concept (Q6: 99566, Q9[3]: 99566, Q14: 75547 statements).

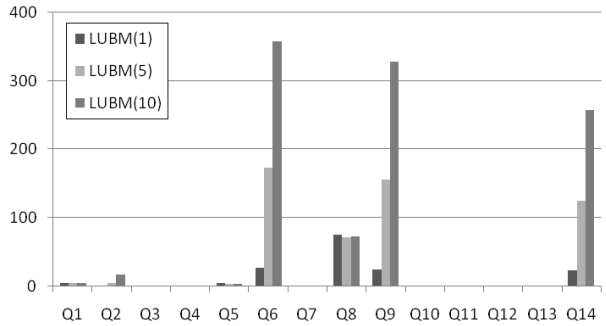

Figure 4. Notification times (ms) for LUBM queries

unsubscribe: removes an active subscription.

The unsubscribe operation does not involve any reasoning or query tasks.

delete: removes all triples from the system's knowledge base that match the given statement pattern.

Deletion times range from 0.9 seconds for LUBM(1) to 12.8 seconds for LUBM(10) when all 14 concept definitions were registered at SENS.

The reasoning engine used in the prototype requires re-reasoning about the entire knowledge base after every removal operation. The benchmark shows that the deletion operation currently exhibits the bottleneck of the system. Although SENS improves the average removal times for many application scenarios by grouping deletion operations, worst case deletion times of up to 13 seconds for LUBM(10) will only allow for the implementation of responsive coordination applications, if comparably few deletions requests are issued to the system. However, current research in the field of reasoning algorithms already addresses this problem. E.g., in [12], an algorithm is proposed that allows for performance improvements for dynamic knowledge bases of "up to three orders of magnitude". We plan to investigate these algorithms and to integrate this technology in the next version of SENS.

[2] All benchmarks were run on an Intel Pentium IV HT 3GHz, 3 GB Ram, Windows Vista PC.

[3] Note that actually only 2540 individuals match this subscription. For detecting exactly the same set of individuals as specified in LUBM Q9, we employ a graph pattern that filters the raised notifications.

5. CONCLUSION

In this paper, we proposed a coordination model based on a subscription mechanism for semantic repositories that reliably notifies subscribers about semantically defined events. We described the core interaction mechanisms of the coordination model, presented a formal description of its behavioural semantics, and discussed how the coordination system can leverage semantic technology with respect to expressivity, interoperability, extensibility, and data consistency.

The presented performance benchmarks showed already good results for the implementation of coordination applications with steadily growing, monotonic knowledge bases. However, they also pointed out the need for optimizations for the processing of dynamic knowledge bases.

In future work, we are going to address this problem and to further investigate the use of incremental reasoning algorithms. Besides this, we also plan to extend the formal definition of the model with ordering and reliability requirements in order to provide the formal basis for additional optimizations of the coordination system.

6. ACKNOWLEDGEMETNS

This work was partly supported by the FP6 project TripCom (IST-4-027324-STP).

7. REFERENCES

[1] Alpern, B., Schneider, F.B. Defining liveness. *Information Processing Letters*, 1985.

[2] Andrews, P. D. *An introduction to mathematical logic and type theory: to truth through proof.* Kluwer Academic Publishers, 2002.

[3] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., Patel-Schneider, P. F. *The description logic handbook: theory, implementation, applications*. Cambridge University Press, Cambridge, UK, 2003.

[4] Catarci, T. And Lenzerini, M. Representing and using interschema knowledge in cooperative information systems. *Journal of Intelligent and Cooperative Information Systems*, 2(4):375-398, 1993.

[5] Feng, M.D., Gao, Y. Q., and Yuen, C.K. Distributed Linda Tuplespace Algorithms and Implementations. In Proc. of the 3rd Joint Int'l Conf. on Vector and Parallel Processing, 1994.

[6] Fensel, D., Kühn, e., Leymann, F., and Tolksdorf, R. Queues Are Spaces - Yet Still Both Are Not The Same? Technical Report, DERI, Univ. Innsbruck, October 2007.

[7] Fensel, D., Krummenacher, R., Shafiq, O., Kühn, e., Riemer, J., Ding, Y., and Draxler, B. TSC - Triple Space Computing, In Special issue on ICT research in Austria, Journal of Electronics & Information Technology (e&i Elektrotechnik & Informationstechnik), January-February, 2007.

[8] Fiege, L., Mühl, G., F.C Gärtner. A modular approach to build structured event-based systems. In *Proc. of 2002 ACM Symposium on Applied Computing* (SAC'02). ACM, 2002.

[9] Gelernter, D. Generative Communication in Linda, In *ACM Transactions in Programming Languages and Systems (TOPLAS)*, 7(1), 80-112, 1985.

[10] Gigaspaces. http://www.gigaspaces.com: August 2008.

[11] Guo,Y., Pan, Z., Heflin, J. LUBM: A Benchmark for OWL Knowledge Base Systems. In *Journal of Web Semantics 3(2)*, 2005, pp158-182.

[12] Halaschek-Wiener, C., Parsia, B., Sirin, E.: Description Logic Reasoning with Syntactic Updates. *OTM Conferences (1) 2006*: 722-737.

[13] Hohpe, G, Woolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley Professional, 2003.

[14] Khushraj, D., Lassila, O. and Finin, T.W. sTuples: Semantic Tuple Spaces. In *1st Ann. Int'l Conf. on Mobile and Ubiquitous Systems*, August 2004.

[15] Kröger, F. *Temporal Logic of Programs*. Springer, 1987.

[16] Kühn, e. *Virtual shared memory for distributed architecture*. Nova Science Publishers, 2001.

[17] Kühn, e., Mordinyi, R., Schreiber, C. An Extensible Space-based Coordination Approach for Modeling Complex Patterns in Large Systems. In: *Proc. of 3rd Int'l Symp. on Leveraging Applications of Formal Methods (ISOLA)*, 2008.

[18] Luckham, D. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[19] Manola, F. and Miller, E. *RDF Primer W3C-Recommend*., 2004. Available at: http://www.w3.org/TR/rdf-primer/

[20] McGuinness, D.L. and van Harmelen, F. OWL Web Ontology Language, W3C Recommendation**,** 2004. Available at: http://www.w3.org/TR/owl-features/

[21] Murth, M. and Kühn, e. Knowledge-driven coordination using a semantic event notification service. In *Proc. of 1st Int'l workshop on emergent semantics and cooperation in open systems (ESTEEM)*. Rome, Italy, 2008.

[22] Mühl, G., Fiege, L., Pietzuch, P. R. Distributed event-based systems, Springer-Verlag, 2006.

[23] Nixon, L.J.B., Paslaru Bontas Simperl, E., Antonenko, O., and Tolksdorf, R. Towards Semantic Tuplespace Computing: The Semantic Web Spaces System. In *22nd Ann. ACM Symposium on Applied Computing*, March 2007.

[24] Nixon, L. J. B., Simperl, E., Krummenacher, R., and Martin-Recuerda, F.: *Tuplespace-based computing for the Semantic Web: a survey of the state-of-the-art.* The Knowledge Engineering Review, Volume 23, Issue 02, 2008.

[25] Ontotext. *OWLIM Fact Sheet*, available at: http://www.ontotext.com/owlim/OWLIMFactSheet.pdf

[26] Petrovic, M., Liu, H., and Jacobsen, H. G-ToPSS: fast filtering of graph-based metadata. In *Procc of the 14th Intl. Conf. on World Wide Web (WWW '05)*, ACM Press, 2005.

[27] RDFCube. Database grid grid – RDFCube. Available at: http://projects.gtrc.aist.go.jp/dbwiki/pukiwiki.php?RDFCub,

[28] Simperl, E., Krummenacher, R., and Nixon, L. A Coordination Model for Triplespace Computing. In *Proc. of: 9th Int'l Conf. on Coordination Models and Languages,* 2007.

[29] Wang, J., Jin, B., and Li, J. An Ontology-Based Publish/Subscribe System. In *Proceedings of the 5th ACM/IFIP/USENIX Intl. Conf. on Middleware,* 2004.