Design of a Scalable Reasoning Engine for Distributed, Real-time and Embedded Systems

James Edmondson[1] and Aniruddha Gokhale[1]

Dept. of EECS, Vanderbilt University, Nashville, TN 37212, USA
Email: {james.r.edmondson,a.gokhale}@vanderbilt.edu

Abstract. Effective and efficient knowledge dissemination and reasoning in distributed, real-time, and embedded (DRE) systems remains a hard problem due to the need for tight time constraints on evaluation of rules and scalability in dissemination of knowledge events. Limitations in satisfying the tight timing properties stem from the fact that most knowledge reasoning engines continue to be developed in managed languages like Java and Lisp, which incur performance overhead in their interpreters due to wasted precious clock cycles on managed features like garbage collection and indirection. Limitations in scalable dissemination stem from the presence of ontologies and blocking network communications involving connected reasoning agents. This paper addresses the existing problems with timeliness and scalability in knowledge reasoning and dissemination by presenting a C++-based knowledge reasoning solution that operates over a distributed and anonymous publish/subscribe transport mechanism provided by the OMG's Data Distribution Service (DDS). Experimental results evaluating the performance of the C++-based reasoning solution illustrate microsecond-level evaluation latencies, while the use of the DDS publish/subscribe transport illustrates significant scalability in dissemination of knowledge events while also tolerating joining and leaving of system entities.
keywords: knowledge reasoning and dissemination; mission-critical systems; fault tolerance; scalability

1 Introduction

Distributed, real-time and embedded systems (DRE) are characterized by scarce resources and high demands on what is available. More often than not, DRE systems are mission-critical systems where decisions on what to disseminate to other entities in the network must be done within hard deadlines (often in microseconds). Additionally, a deployed DRE system may feature components or agents that are more important than other entities participating in the network or at least have tighter deadlines. Systems that feature such configurable priorities, deadlines, and resource requirements per entity are often called quality-of-service (QoS)-enabled systems.

There have been recent efforts to incorporate knowledge and reasoning into QoS-enabled systems [1, 2], but these efforts are not aimed at mission-critical scenarios, they use proprietary and minimal reasoning, and are implemented in managed languages like Java. These systems are limited in their support to scale to thousands of systems, cannot meet microsecond latency requirements, and are often developed simply as proof-of-concept prototypes or theoretical ideas. Even highly optimized solutions [3] built on C++ engines like RACERPRO operate in dozens of millisecond ranges.

Based on our extensive experience testing such DRE systems and the performance bottlenecks observed in meeting the knowledge reasoning demands for mission-critical QoS-enabled DRE systems, we surmise these limitations to stem from the following reasons: (1) contemporary knowledge reasoning systems are often developed using managed languages in which garbage collection, just-in-time compilation, and other language features tend to cause significantly high latencies during local reasoning operations – generally in the dozens to hundreds of milliseconds when inserting new knowledge into the system (or evaluating new rules); and (2) the dissemination of knowledge between knowledge reasoning peers in a network continues to rely on transports that do not offer quality-of-service differentiation (*e.g.*, UDP and TCP), require blocking communication, and do not scale well to thousands of reasoning peers. Additionally, the current knowledge reasoning engines that are available do not appear to have options for fault tolerance due both to node failures and incorrect knowledge.

To address these issues, we have developed the Knowledge and Reasoning Language Engine (KaRL Engine), which is an open-source knowledge reasoning and dissemination framework for DRE systems. The design and implementation of KaRL focuses on delivering (1) microsecond latency for knowledge rule evaluations, (2) low latency in knowledge dissemination across a network, and (3) QoS mechanisms for fault tolerance, high priority reasoning peers, and knowledge conflicts.

The rest of this paper is organized as follows. In Section 2.1, we present a short motivating scenario for a situation where reasoning services in a DRE system would be appreciated, and then we present our solution in Section 2. We evaluate our approach in Section 3, and review related literature in Section 4. We then wrap up the paper with concluding remarks in Section 5.

2 A Framework for QoS-enabled Reasoning Services

This section describes KaRL, which is our solution to provide a real-time and scalable, distributed knowledge reasoning and dissemination capability. Before we describe the details of our solution we bring about the key requirements for such a capability in Section 2.1.

2.1 Eliciting Requirements for Scalable and Real-time Knowledge Dissemination

We use an example case study to elicit requirements for scalable and real-time knowledge dissemination for DRE systems. Our case study shown in Figure 1 comprises at least ten components that have been deployed into a network (dozens could be deployed as failover components, in case the primary component fails) and each of these components has dependencies between each other that must be reasoned out before the component is able to fully load. This is not an uncommon problem in DRE systems. One component may require services already be started (like GPS systems, schedulers, etc.) before it may execute its own logic which depend on those services being available in the network.

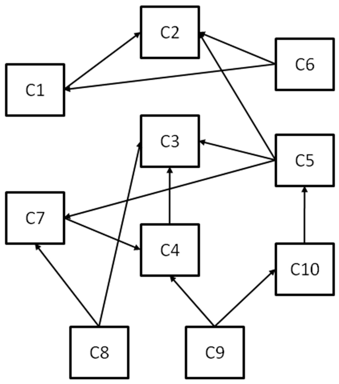

Fig. 1. Deployment plan of ten components with dependencies on each other

Each component in our case study has an init method that checks its deployment dependencies and a run method that is executed after all dependencies have been met. The run method will contain an arbitrary but important user program, and the init method may contain user logic in between checking for our component's dependencies being met and signalling that our component is ready to service others.

The scenario requirements can consequently be summarized into the following:

1. Mission-critical services depend on this ten-component deployment, and timing is crucial. Microsecond deadlines exist per component, and the entire deployment should take milliseconds, including local reasoning and dissemination across the network.
2. The user needs to be able to insert logic in between the reasoning service validating that the component's requirements have been met, and informing other interested components that this component's services are now available (after the user startup logic has been executed).
3. The solution should scale and be configurable to add new components, especially failover components that can replace faulty mission-critical components in the network. Some components are more mission-critical than oth-

ers, and quality-of-service mechanisms should be available for fine-grained reasoning and dissemination of knowledge support.

Our solution to meet these requirements can be broken into three main vectors of attack. In Section 2.2, we describe a highly optimized reasoning language and associated reasoning engine mechanisms that we developed to provide an intuitive abstraction to capture and mutate knowledge for distributed real-time and embedded systems (Challenge 1 and 2). In Section 2.3, we describe our techniques for further optimization and scalability with a high performance, QoS-enabled anonymous publish/subscribe transport (Challenge 1 and 3). Finally, in Section 2.4, we describe the quality-of-service mechanisms like knowledge quality and domains that address some key reasoning issues introduced in Challenge 3.

2.2 Language and Reasoning Engine Mechanisms

Despite the stringent timing requirements of Challenge 1, we were also interested in usability of the reasoning middleware. The Knowledge and Reasoning Language (KaRL) looks and feels like a modern programming language and allows complex conditionals and mutations to be constructed with multi-modal knowledge variables and constants. Support is provided for programming in both a rule-based (e.g., *condition => resulting knowledge*) and declarative rule programming (e.g., *condition && resulting knowledge*), and KaRL allows for both local variables (knowledge that will not be disseminated but can be referenced and mutated by the reasoning agent) and global variables (which will be dessiminated to interested reasoning agents).

To evaluate KaRL, we provide several engine mechanisms exposed to C++ programmers via an object-oriented library of function calls. Several mechanisms are provided that allow for accessing or mutating individual knowledge variables, interpretating KaRL logics, and providing fine-grained debugging capabilities of global knowledge state from a C++ executable or library. Each mechanism is atomically executed, *i.e.*, no outside entities may change the local knowledge state maintained by the KaRL reasoning engine while the user program is calling these mechanisms, and the library is completely thread-safe. We focus our discussions on the two chief mechanisms that are required to meet Challenge 2: *evaluate* and *wait*.

The evaluate and wait mechanisms both compile the provided KaRL user logic into an optimized format and then evaluate the resulting expression tree. The key difference between the evaluate and wait mechanisms in the KaRL engine is that an evaluate call evaluates the expression tree only once. In contrast to evaluate, the wait mechanism will evaluate the expression tree until the result of the expression tree is non-zero (true). To save precious microseconds on future evaluations, the expression tree is cached and future evaluations or waits on the KaRL user logic will not be compiled again. At the end of an evaluate or wait call, the knowledge state mutations are aggregated into a knowledge event that is ready to then be disseminated to all interested peer entities.

In a system with periodic nature that needs to run a KaRL logic repeatedly for an extended period of time, the wait mechanism reduces computation overhead by limiting the re-evaluation to only when changes have been made to the global knowledge state. If the same logic is executed with an evaluate mechanism nested within a for loop, the logic will be evaluated constantly, even if no change has been made. Consequently, wait should always be preferred in a periodic system that synchronously waits until certain predicates are met. Evaluate should be used when the results of the logic are not critical to the software entity or there is other work that can be done while waiting for conditions or global state to change.

We mention these distinctions because our motivating scenario requires both mechanisms. The wait mechanism is inserted into the top of of the component's init function, e.g., component C5 requires components C2, C3, and C7 to be ready so it calls *engine.wait ("C2.ready && C3.ready && C7.ready")*. The user is then allowed to insert their own programming logic for setting up the component services and subsequently calls the evaluate mechanism with a global variable mutation to indicate that the service is ready, e.g., for component C5, *engine.evaluate ("C5.ready = 1")*.

After an evaluate or wait call has been made, a tuple of form $S(K, V)$ is created to represent the changes to the local state. K is a vector of length n of all changed knowledge variables and V is a vector of length n of the corresponding values, where $K_i = V_i$. With the local state changes aggregated into a tuple form, we have the first major piece of the knowledge event.

2.3 Dissemination of Knowledge in the KaRL engine

Once a knowledge tuple S has been formed via the KaRL language and reasoning engine mechanisms outlined in Section 2.2, the event is almost ready to be passed to interested reasoning agents. In this section, we discuss the mechanisms and additional information required to globally order the knowledge events as well as demonstrate how the anonymous publish/subscribe paradigm is used to disseminate knowledge to entities within a knowledge domain (a partition of the networking entities that might be interested in knowledge from this entity).

The tuple S which represents changes to an entity's local state is now augmented with a Lamport clock time t [4] and a quality q, which is essentially the priority of the entity to write knowledge to this variable and is explained in more detail in Section 2.4. The Lamport time mechanism requires each entity to keep a local counter for each variable (referred to as V_t) in the knowledge base which represents the time stamp at which the last update occurred to the knowledge variable. Additionally, an entity Lamport clock c is maintained which is incremented only if the state is changed by a KaRL user logic via a KaRL engine mechanism call on the local entity (in which case $c = c+1$) or a knowledge event arrives which has a more recent Lamport clock time t, in which case $c = t$. To clarify this process, we outline the equation for updating the entity clock (c) in Table 1.

Table 1. Equation for updating an entity's Lamport clock c

$$c = \begin{cases} c+1 & \text{if E(S,t,q) is a local event} \\ max(c,t) & \text{if E(S,t,q) is a remote event} \end{cases}$$

Before a knowledge event of form $E(S,t,q)$ is constructed, the entity clock is updated according to the equation in Table 1 and then we set $t = c$. After this step is completed, $E(S,t,q)$ is formed and the knowledge event is finally ready to be disseminated across a knowledge domain d. The domain can be set via the KaRL engine mechanisms to identify partitions of the global knowledge space that the software entity is interested in.

To facilitate the scalable delivery of knowledge events to entities in the domain, we use the OMG Data Distribution Service (DDS) [5]. Specifically, we provide a configurable transport for both OpenSplice Community Edition, an open source implementation of the DDS standard, and RTI's NDDS, a closed source option.

The anonymous publish/subscribe paradigm used by DDS fits our solution concept in the following ways. First, DDS allows for us to be host and entity agnostic because it relies on an anynomous paradigm, which enables a programming model that intrisically supports fault tolerance (Challenge 3) because we are no longer tied to host/port information and are instead interested in components publishing knowledge relevant to our interests. If a sensor or actuator in a DRE system is tracking a globally recognized knowledge variable, the value of the variable is essentially more important than where it came from.

Second, with the dynamic nature of DRE systems (*e.g.*, failing and joining entities), any solution that requires predetermined URI information for connecting ontologies and variables is detrimental to a real-time mission critical system. Third, many of the features of DDS can be directly mapped to our application domain of distributed knowledge and reasoning, *e.g.*, DDS domains, for the most part, directly map to our concept of knowledge domains. Last, these transports have a history of scaling to thousands of components and data dissemination latency in microseconds [6, 5]. Consequently, in order to meet the scalability requirements in Challenge 3, we need to add very little overhead via local reasoning services (preferably low microseconds to nanoseconds) to the low latency of the DDS transport mechanisms (which are also in the microseconds on local area networks).

2.4 Quality-of-Service Mechanisms in KaRL

The knowledge event $E(S,t,q)$ has a parameter q, the knowledge event quality, which indicates the utility of the updates from this software entity, according to the KaRL logic developer. When forming the q of the knowledge event E, the KaRL engine aggregates the qualities of all variables in the knowledge event according to the following equation: $q = max(q, S.V_i)$. By allowing developers to set quality per knowledge variable or entity on each entity, a KaRL logic developer can exert fine-grained control over the entity's knowledge quality on a per-variable basis. This same mechanism also allows a developer to safely deploy redundant components from our motivating scenario, the system will still function properly, and preference may be given to a component running on the best hardware or one which is running on top of the sensor that knowledge in the network is based on (e.g. the component is running on a GPS sensor).

Quality always overrides the Lamport clock value to dictate global ordering (i.e. t is a tiebreaker for q - not the other way around). This caveat means that regardless of how fast a low quality sensor, actuator, component, etc. is publishing its knowledge events, a high quality sensor's data will always be preferred. We do provide an optional timeout quality-of-service to allow for low quality knowledge events to eventually overwrite high quality knowledge, if new updates by high quality knowledge publishers haven't been received after some user-provided timeout interval. Wherever possible, we map our quality-of-service parameters to the DDS transport under the hood without requiring the user to understand DDS quality-of-service concepts.

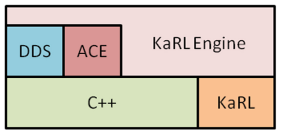

Fig. 2. KaRL Architecture

The resulting solution architecture is shown in Figure 2. The Adaptive Communication Environment (ACE, project site at http://www.dre.vanderbilt.edu/ACE) shown in the diagram is used by the KaRL engine whenever portable operating-system specific mechanisms are required (e.g. transient threads for receiving knowledge updates while user application logic is being ran in the main thread of execution).

3 Evaluating the KaRL Engine

This section empirically evaluates the KaRL reasoning engine to determine if it is able to support the knowledge reasoning and dissemination requirements of DRE systems, which include microsecond-level latencies for evaluating knowledge events, and scalability to thousands of system entities. To that end, Section 3.1 investigates the KaRL engine's throughput in interpreting logic. Section 3.2 investigates the dissemination latency when using Open Splice DDS as a transport.

Unless specified otherwise, all experiments were conducted on five IBM blades with dual core Intel Xeon processors at 2.8 GHZ each and 1 GB of RAM running

Fedora Core 10 Linux. The code was compiled with g++ with level 3 optimization, and each test featured a real-time class to elevate OS scheduling priority to minimize jitter during the test runs. Each test scenario was executed 20 times to form an average, and then the tests were repeated 10 times. The code for the test is available via the KaRL project site.

3.1 Knowledge Reasoning Latency

In this series of tests we create KaRL logics that test the KaRL engine's ability to process local knowledge without dissemination to test whether or not the engine can support the timing properties (microsecond latencies) of DRE systems. These tests isolate the latency of the KaRL engine from the network dissemination latency of the KaRL engine and DDS transport, which are tested in the next section. The results help to identify the strengths of KaRL to meet the performance requirements.

Although the performance of the KaRL engine in evaluating one knowledge rule is a useful metric, we were also interested in timing information for how efficient KaRL is in handling larger user logics. To test each of these scenarios, we constructed four total logics. The first is a simple reinforcement (*++.var1*) that is evaluated inside of a C++ *for loop* 1,000,000 times. The second is a predicate guarded reinforcement (*predicate => ++.var1* that is evaluated inside of a C++ *for loop* 1,000,000 times. These logics were then expanded into KaRL logics that performed 10,000 simple reinforcements and predicate guarded reinforcements that were evaluated in a C++ *for loop* 100 times. The results of these experiments are shown in Table 2.

Table 2. Reasoning Engine Latency

Reinforcement	997 ns
Chained Reinforcement	502 ns
Guarded Reinforcement	1,000 ns
Chained Guard Reinforcement	597 ns

The results indicate that not only can our solution perform reasoning services within microsecond deadlines, but the KaRL engine can perform reasoning services on larger user logics in nanoseconds of time. The specific optimizations performed here that result in larger user logics performing faster is removing the function call and compiled expression tree lookup that is performed with every call to *engine.wait* or *engine.evaluate* as described in Section 2.2.

3.2 Dissemination Latencies in KaRL

In this series of tests we create a number of KaRL logics that generate knowledge events and disseminate them across a local area network to interested reasoning peers using OpenSplice DDS as the transport. We break down the latencies experienced as the logic is compiled, evaluated, and disseminated. Our aim is to investigate how close KaRL is to the native dissemination protocol, which is an indicator of how much overhead, if any, it imposes on the dissemination and other latencies. To gauge these latencies, we construct tag-along logics shown in Table 3.

Table 3. Latency test setups

Process 1	Process 2
P0 == P1 => ++P0	P0 != P1 => P1 = P0

The two tag-along processes modify global knowledge variables and produce knowledge events for dissemination. The first process changes $P0$ whenever $P0 == P1$. The second process changes $P1$ whenever $P1 \, ! = P0$. Combined together with the *engine.wait* mechanism, these KaRL logics create a continuous system that we can stop after a certain number of logic evaluations. The logics are evaluated 5k, 25k, 50k, and 100k times and we include the network latencies obtained via a ping utility to guage the latency added by the KaRL engine and DDS dissemination transport.

Table 4. Average Latency Results

	5k	25k	50k	100k	500k
Ping	114 us	114 us	114 us	114 us	114 us
Dissem	650 us	440 us	437 us	315 us	317 us
Compile	55 us	55 us	55 us	55 us	55 us
Eval	3 us	2 us	3 us	3 us	3 us

The compilation time penalty is only paid once: the first time `wait` or `evaluate` mechanism is called on the KaRL user logic. After the KaRL user logics are compiled with the wait or evaluation mechanisms, the expression is cached, and compilation time is reduced to the time it takes to lookup the KaRL logic in an STL map (part of the C++ standard library), which is a matter of nanoseconds on modern processors. Thus, after compilation time is reduced to lookup time, total evaluation latency is linear to the number of operations performed in the compiled expression tree.

Each dissemination latency average includes KaRL logic evaluation latency because in the dissemination latency we are interested in the time it takes for a user-provided KaRL logic to be evaluated, and the changes propogated across the network and received by interested entities. Though we compute dissemination latency with roundtrip times, we present the one shot time in Table 4 by dividing the average roundtrip time by 2, since the roundtrip is the result of a knowledge event going to the subscriber and then returning for 2 total events for a single latency calculation.

These experiments were repeated twenty times and the average latencies could deviate by as many as 70 us. Our lowest average latency for 100k and 500k messages hovered at 240 us and our highest was 380 us. We also saw jitter of 30+ us in the ping tests. We believe the majority of this jitter is being caused by operating system context switching due to I/O operations required for dissemination. The last important note to make about DDS and the dissemination latency is that this latency scales well to thousands of reasoning peers because of its reliance on broadcast and multicast paradigms where available and flexible asynchronous patterns underneath that even handle single-point communications efficiently on multi-core systems [6, 5].

4 Related Work

In this section we review existing work in reasoning languages and the dissemination of knowledge between reasoning peers. Most existing knowledge and reasoning engines require ontologies and do not support location transparency between participating entries. For example, DRAGO [7] specifically uses TCP and HTTP for connection-oriented communications between each reasoning entity. The SOMEWHERE [8] project is built on P2PIS [9], which creates a network of peer and variable mappings. DDL [10, 11] is similar to the other technologies but also provides data aggregation and the specification of incoming and outgoing peer/variable mappings. All of these technologies require users to build mappings of variables from local variables to other peers by their URIs, do not temporally order events, and do not distinguish between important and unimportant events. In contrast, our solution does not require such peer and variable mappings, it temporally orders events, is built for real-time systems, and allows for priorities.

Partition-based Reasonings, such as the High Performance Knowledge Base (HPKB) [12], allow for variable mappings between logical partitions. The concept of logical partitions is similar to our solution concept of knowledge domains, which separate the dissemination space and reduce message complexity. The consequence finding algorithm employed in the HPKB is, however, very specific to modal (*i.e.*, boolean) logic and will result in conflicts when multiple sensors or actuators are modifying the same variable with different values – an issue that occurs with multiple sensors informing listeners of target tracking or temperature. Our solution to these issues involves quality and multi-modal logic, which is described in Section 2.

Policy management services, such as KAoS [13, 14], are built to work with semantic web languages based on the Web Ontology Language (OWL) [15]. KAoS essentially provides a policy reasoning engine which can be configured to work with the semantic variable mappings provided by the OWL standard. Recently, this type of policy management service was integrated into the Quality-of-service Enabled Dissemination (QED) project [1, 2], which also uses a publish/subscribe paradigm. However, the QED infrastructure was designed to disseminate user-provided data events and not knowledge and reasoning information and does not globally order events.

Several content-based tools for publish/subscribe ontologies have been created including OPS [16] and other Semantic Message Oriented Middleware [17, 18]. All of these tools feature a content-based querying system that forces subscribers to subscribe to all topics, which results in more messages than should be necessary. Additionally, these systems are typically built to sustain latencies of 1-2 seconds for performing complex queries, parsing complex types like graphs, or other types of semantic matching operations on content. These technologies also do not try to enforce temporal consistency (ordering) of events or event priorities between sensors, actuators, or software entities.

In contrast to these content-based pub/sub ontologies, KaRL allows for querying content through a *wait* mechanism within a specified domain to reduce message complexity and work within time constraints (microseconds) for real-time systems. Consequently, we use matching on content after subjects (topics) have been matched. Additionally, we temporally order events and allow for knowledge quality, to specify importance and priority, which these technologies do not support.

5 Conclusions

In this paper we presented KaRL which was designed to provide knowledge and reasoning services for distributed, real-time and embedded systems. Included in these technologies is a flexible, multi-modal grammar for creating knowledge events. It also includes an infrastructure for globally ordering and setting priorities - called qualities in the KaRL engine - for knowledge events based on modified knowledge. KaRL uses anonymous publish/subcribe paradigms and technologies to facilitate knowledge dissemination in a data-centric manner that scales well. Other transports like UDP and TCP can be supported by extending well-defined C++ interfaces and involves overriding a total of two functions. Empirical validation of KaRL indicates it supports both the timing and scalability requirements for knowledge reasoning and dissemination in DRE systems.

The KaRL engine has recently served as the foundation of an automated testing and deployment suite called KATS, which is being used for distributed instrumentation and testing of smart phones and services. Ongoing work in KaRL includes support for other transports, data types and ontologies. KaRL and its associated tool suite are available in open source from madara.googlecode.com.

References

1. Loyall, J.P., Gillen, M., Paulos, A., Bunch, L., Carvalho, M., Edmondson, J., Varshneya, P., Schmidt, D.C., III, A.M.: Dynamic policy-driven quality of service in service-oriented systems. In: In Proceedings of the 13th International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 10. (2010)

2. Loyall, J., Carvalho, M., Schmidt, D., Gillen, M., III, A.M., Bunch, L., Edmondson, J., Corman, D.: Qos enabled dissemination of managed information objects in a publish-subscribe-query information broker. In: SPIE Defense Transformation and Net-Centric Systems. (2009)
3. Kaplunova, A., Möller, R., Wandelt, S., Wessel, M.: Towards scalable instance retrieval over ontologies. In: Proceedings of the 4th international conference on Knowledge science, engineering and management. KSEM'10, Berlin, Heidelberg, Springer-Verlag (2010) 436–448
4. Lamport, L.: Ti clocks, and the ordering of events in a distributed system. Commun. ACM **21** (July 1978) 558–565
5. Pardo-Castellote, G.: OMG data-distribution service: architectural overview. In: 23rd International Conference on Distributed Computing Systems Workshops, 2003. Volume 0. (May 2003) 200–206
6. Xiong, M., Parsons, J., Edmondson, J., Nguyen, H., Schmidt, D.: Evaluating technologies for tactical information management in net-centric systems. In: Proceedings of the Defense Transformation and Net-Centric Systems conference. (2007)
7. Serafini, L., Tamilin, A.: Drago: Distributed reasoning architecture for the semantic web. In: Extended Semantic Web Conference. (2005) 361–376
8. Adjiman, P., Chatalic, P., Goasdou, F., c. Rousset, M., Simon, L.: Distributed reasoning in a peer-to-peer setting. In: In de Mantaras. (2004) 945–946
9. Abdallah, N., Goasdoué, F.: Non-conservative extension of a peer in a p2p inference system. AI Communications **22** (December 2009) 211–233
10. Serafini, L., Borgida, A., Tamilin, A.: Aspects of distributed and modular ontology reasoning. In: International Joint Conferences on Artificial Intelligence. (2005) 570–575
11. Borgida, A., Serafini, L.: Distributed description logics: Assimilating information from peer sources (2003)
12. Amir, E., Mcilraith, S.: Partition-based logical reasoning for first-order and propositional theories. Artificial Intelligence **162** (2000) 49–88
13. Uszok, A., Bradshaw, J.M., Johnson, M., Jeffers, R., Tate, A., Dalton, J., Aitken, S.: Kaos policy management for semantic web services. IEEE Intelligent Systems **19** (July 2004) 32–41
14. Tonti, G., Bradshaw, J.M., Jeffers, R., Montanari, R., Suri, N., Uszok, A.: Semantic web languages for policy representation and reasoning: A comparison of kaos, rei, and ponder. In: International Semantic Web Conference. (2003) 419–437
15. Horrocks, I., Patel-Schneider, P.F., Harmelen, F.V.: From shiq and rdf to owl: The making of a web ontology language. Journal of Web Semantics **1** (2003) 2003
16. Wang, J., Jin, B., Li, J.: An ontology-based publish/subscribe system. In Jacobsen, H.A., ed.: Middleware 2004. Volume 3231 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2004) 232–253
17. Petrovic, M., Burcea, I., Jacobsen, H.A.: S-topss: Semantic toronto publish/subscribe system. In: IN: PROC. OF CONF. ON VERY LARGE DATA BASES. (2003) 1101–1104
18. Lien, Y.C.N., Wu, W.J.: A lexical database filter for efficient semantic publish/subscribe message oriented middleware. In: Proceedings of the 2010 Second International Conference on Computer Engineering and Applications - Volume 02. ICCEA '10, Washington, DC, USA, IEEE Computer Society (2010) 154–157