International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:1, No:4, 2007

DJess – A Knowledge-Sharing Middleware to Deploy Distributed Inference Systems

Federico Cabitza, Bernardo Dal Seno

Abstract**—** In this paper DJess is presented, a novel distributed production system that provides an infrastructure for factual and procedural knowledge sharing. DJess is a Java package that provides programmers with a lightweight middleware by which inference systems implemented in Jess and running on different nodes of a network can communicate. Communication and coordination among inference systems (agents) is achieved through the ability of each agent to transparently and asynchronously reason on inferred knowledge (facts) that might be collected and asserted by other agents on the basis of inference code (rules) that might be either local or transmitted by any node to any other node.

Keywords**—** Knowledge-Based Systems, Expert Systems, Distributed Inference Systems, Parallel Production Systems, Ambient Intelligence, Mobile Agents.

I. Introduction

DJess stands for "Distributed Jess" and Jess [1] for "Java Expert System Shell", a Rule Engine and scripting environment written in Java™ at Sandia National Laboratories since the mid 90's. Jess, as a rule engine tightly integrated with the Java language, allows programmers to build Java applications that have the capacity to reason on "factual" knowledge expressed in terms of symbolic expressions by means of "procedural" knowledge supplied in the form of declarative rules. DJess has been developed in order to add to the Jess inference functionalities a set of mechanisms that allow to transparently create and manage an integrated collection of *Inference Systems* (IS) that communicate according to a *generative communication* model [2]. In a generative communication framework, agents communicate through a shared and distributed memory so that no explicit communicative message is necessarily exchanged. At this extent, DJess communication model can be considered akin to the tuple space model embodied by the Linda language [3] and its several specific dialects. As a middleware[1], DJess offers both spatial and temporal decoupling by allowing processes to be directly unaware of each other's identities and to run their control flows asynchronously and even within non-overlapping lifetimes; it enables information sharing and communication among distributed agents through hidden remote object invocations (i.e., RMI); furthermore, DJess provides a common programming abstraction across a distributed system by turning a collection of devices (and applications running on them) into a single integrated computational environment; such a system within the DJess framework is called a *Web of Inference Systems* (WoIS). We use the quite general term "inference system" to indicate computational systems (e.g., agents, devices[2]) that are able to reason about what they perceive, and hence able to create and manipulate knowledge about the environment and themselves. A WoIS is then a network of communicating, independent (time and space decoupled) inference systems that are integrated by the capability of inferring on knowledge that is transparently shared among the network's members.

In the next section we first introduce the concept of *distributed inference system* along with the main characteristics of the DJess architecture; this will be further outlined in the following section along with some implementation details; the last section contains a brief report on both current and future work and the main motivations of the project.

II. DJess-based Distributed Inference Systems

In this section we first give some vocabulary on rule-based systems with a particular reference to the Jess lexicon; then an outline of the DJess architecture is given.

A. Jess-based Inference Systems

Jess is a Java program written by Ernest Friedman-Hill to deploy fast and flexible *Rule-Based Systems* (RBS) [4]. RBSs are programs whose control flow can be said both event-driven and data-driven in that they perform some action only if some condition is true and they are able to produce (infer) conclusions from a set of premises[3]. In fact, *rules* are but *if-then* computational constructs expressing recommendation of action if some condition occurs and RBSs are programs that select and execute available rules according to their current context; this makes RBSs particularly suitable for implementing reactive architectures that must be modular and easily extensible. Like other systems implemented by similar rule-based languages (e.g., CLIPS, OPS5), each Jess inference system is composed of three main components: a *rule set*, a *working memory* and a *rule engine*. The rule set is the collection of all the rules that can be executed by the IS according to the current content of the working memory. The Working Memory (WM) is the storage of the knowledge

F. Cabitza and B. Dal Seno are with the Laboratory of Models & Architectures for Coordination at Università degli Studi di Milano Bicocca, Milano, Italy (e-mail: cabitza@disco.unimib.it; dalseno@disco.unimib.it)

[1]For middleware we adopt the widely accepted definition that conceives of it as any software that mediates between application programs and a network in order to make them interconnected and hence able to communicate.

[2]We can refer to these systems by several equivalent terms, such as devices or agents if we like to stress respectively either their user-interactional and computational capabilities or their intelligent and autonomous behavior in complex real settings; likewise we can refer to ISs by the term nodes, if we refer to their spatial decoupling in a topological network, or production systems (or rule-based systems) if we refer to the particular kind of reasoning model they reify. Since agents are often described as entities able to perform inferential (i.e., deductive) symbolic reasonings, we consider the terms Inference Systems and Agents as synonyms.

[3]Accordingly, from this point on, we use rule-based systems, production systems, and inference systems as synonyms.

nuggets the rule engine operates on; these nuggets represent the factual knowledge of the RBS and are hence called *facts*: facts are like records with certain value fields called *slots*; the fields structure is specified by a template that defines types and default values of the slots. The rule engine — also called *inference engine* — is a program (interpreter) that performs iteratively the so-called *Match-Resolve-Act* (MRA) cycle over the rule set and the working memory. In a MRA cycle, at any given time, the rules of the rule set are first selected and activated by matching them against the WM elements (Match phase); then a rule among the activations is chosen for immediate execution according to some selection strategy (Resolve phase) and finally executed (Act or firing phase). In the Act phase, facts can be modified or deleted, and new facts can be added to the working memory as well.

In choosing the scripting environment to base our distributed inference system on, the possibility to deploy our solution across different platforms and to rely on a well documented architecture played a decisive role. Therefore, we chose Jess since its source code is available, it is written entirely in Java and is widely used within a professional programmers' community.

B. DJess: Distributing an Inference System

Distributing an inference system means both to distribute in space the inference engines and to parallelize in time the inference process made on the same facts. Distribution of an inference system can be useful for two main reasons: making the inference process faster and getting physical separation. The former allows better performances in terms of throughput and response time; the latter supports robustness, resource sharing, agents cooperation, and pervasiveness of computation. As a matter of fact any design must come to compromises and no optimal solution for every application domain is given. In designing the distribution of our Jess-based inference systems, we had to face design and implementation alternatives between performance and consistency guarantee. The application domain that has mainly influenced the DJess design is the *Ambient Intelligence* (AmI) environment [5]. An AmI environment can be defined as a tightly integrated collection of computational devices that is able to aptly assist the user whenever and wherever she needs it. Such a collection is inherently distributed and heterogeneous since the computational systems that compose it can be either mobile and embedded in any kind of everyday object or embedded and hidden in the background infrastructure surrounding the user [6]; besides ubiquity and transparency, such systems must also exhibit some intelligent behavior in terms of context awareness and ability to recognize and react aptly to user actions by reasoning on knowledge that was previously acquired or possibly learned from past experience. For such a domain, we think that the deployment of distributed inference systems is likely predictable; in such systems, coordination would rely on a very high level (semantic) interoperability provided by a suitable communication middleware.

Since our proposed solution — the Web of Inference Systems — has been conceived for a set of application domains where response time is less critical, its design and implementation trade often performance for consistency where this is suitable for the AmI domain. That notwithstanding, the WoIS provides an abstraction level and a sound mechanism to relieve the programmer of having to build a communications framework in every domain where a knowledge-based and inference-based technology is feasible. A representation of a WoIS is given in Fig. 1: a WoIS is defined as a set of registered ISs, called *members* of the WoIS; a set of *shared facts* (constituting the *Shared Working Memory*, SWM); and the WoIS *manager*, which can be either a regular IS with managing functions or a dedicated service provider.

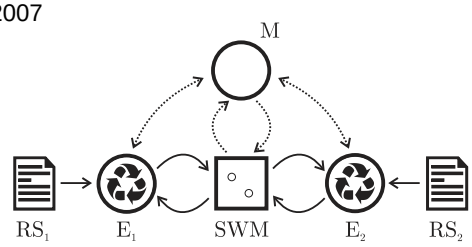

Fig. 1. *Web of Inference Systems*. WoIS members are inference systems constituted by an inference engine (E) and a rule set (RS) each. SWM is the *shared working memory* and it is where all the shared facts are stored and accessed by any member of the WoIS. A system, M, plays the role of web *manager*.

The manager offers two services: the first one is the registration and the locating of ISs as they join or leave the WoIS; the second one is the incremental backup of the SWM so as to manage temporary disconnections of the members and to partially reconstruct the distributed inference history. The use of a centralized WoIS manager has the disadvantage of creating possible bottlenecks in the system, but it is obviously the simplest solution for a name server. We have assumed that in a typical AmI environment computation be mainly driven by user activity and that communication be based on broadband LANs, so that the delays induced by a centralized model are presumably not appreciable.

As depicted in Fig. 1, the SWM is the set of all the shared facts. Each agent can match and execute its rules on shared facts as if they were local; in effect there is no physical common memory in DJess: every IS has a copy of each shared fact in its own local WM and all ISs' engines run independently of each other. For these two reasons we say that the DJess WoIS is a distributed asynchronous inference system [7] where no centralized data repository is kept and facts are transparently shared among the nodes of the network.

Generally a distributed system is said transparent when it is able to ensure that "a collection of independent computers appear to its users as a single coherent system" [8]. This acceptation of transparency has been one of the most important requirement in designing DJess and it regards both the Java programmer and the inference system designer, i.e. who writes the rule set and conceives the facts structure according to the application domain. For the former, we speak of transparency especially for the synchronization mechanisms involved in knowledge sharing. For the latter, transparency has been guaranteed in the inter-IS communication, since asserting facts

is a sort of implicit communication between the asserting IS and the other ISs, whose rules might match the asserted fact. This ensures a sort of backward compatibility, as almost no modification to the RBS code is required when porting it from a monolithic system to a distributed one.

Other two important properties of distributed systems have been taken into account in designing our architecture: reliability and, at some extent, performance. With regard to the former we have conceived both security mechanisms related to the identification/registering of the members of the system and fault tolerance mechanisms, at least in terms of managing sudden disconnections of members and misalignment of the distributed working memories; in regard to the latter property — performance — we have designed our system so that the most time-demanding phase in the inference execution (the Match phase [9], [10]) is left local[4].

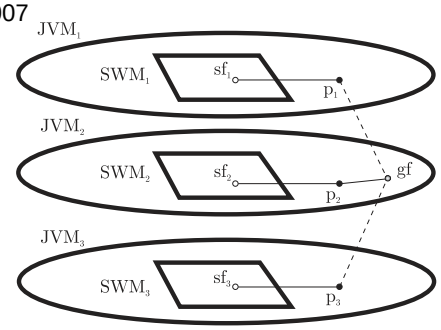

Fig. 2. A graphical representation of the synchronization mechanism in DJess is given. JVM_i are Java Virtual Machines; SWM_i are local copies of the shared working memory; sf_i are shadow facts, representing the same shared fact; p_i are their corresponding proxy objects; and gf is the ghost fact used for the shared fact synchronization.

III. The DJess Architecture and Implementation

DJess is mainly an extension of Jess; its architecture is designed with the goal of keeping modifications of the Jess code as few as possible, so as to cut down on development time and bugs, and to avoid duplicate work being done on the same functionalities. DJess, like Jess, is both a scripting language and a rule engine, that is a program that involves several Java classes and their methods (API).

As a scripting language DJess is a tool for the rule-based application designer. From her point of view, DJess totally adopts the Jess syntax but also provides a `load-rule` function and a different default resolve strategy. The `load-rule` function enables to load a rule into a remote rule set; this makes DJess a (weak) mobile code environment [11] since it allows interpretable code to be transparently transferred among different inference systems. The default resolve strategy is partially nondeterministic in order to lower the overhead due to synchronization between ISs. About sharing, programmers can decide whether a fact will be private or shared at the assertion time; the default policy is that any asserted fact is transparently shared.

As an inference engine, DJess is a tool mainly for the Java programmer. Since a Java program can easily embed a Jess or DJess rule-base system, the development of a bridge between the inference application and the platform or device functionalities is up to the Java programmer. The only difference between Jess and DJess is that, in the latter, methods for creating, joining, leaving, and destroying a WoIS are available.

In order to synchronize the shared memory, DJess exploits a powerful Jess feature: *shadow facts*. This feature allows to use Java beans (objects whose attributes are accessible through set and get methods) as if they were elements of the working memory; when an object is used in such a way, Jess creates a fact — the shadow fact — that is dynamically linked with that object: every modification made on the bean is mirrored on the "shadowed" fact and vice versa. For every fact shared across the WoIS, a shadow fact, and hence a shadowed java bean, is present in every node (see Fig. 2). The Java bean is called *proxy*; all the proxies corresponding to the same shared fact are linked together by means of a Java remote object that we call *ghost fact*. The ghost fact has been introduced for two purposes: storing the state of the shared fact, i.e. the slot values; and providing a single point of synchronization for the ISs. For simplicity each ghost fact is stored in the Java virtual machine of the asserting IS.

Conflicts and unwanted interferences may occur when different ISs access the same shared fact by firing interfering rules at the same time. Two rules are said to interfere if there is a dependency of some sort between them [12]. A read-write dependency (or a true data dependency) occurs between two engines if one of them fires a rule that writes (i.e., modifies or deletes) a fact read (contained in the if-part) by the other and a write-write dependency (or an output dependency) occurs if both of them write the same fact [13].

In DJess, to prevent inconsistency due to interfering, a mutual exclusion mechanism is implemented through locks associated with ghost facts and acquired in the transition from the Resolve to the Act phase. Rule firings are treated as single indivisible units (consistency through serializability[5]): the effects of two parallel rule firings are the same of a serial sequence of those firings; moreover changes made by a rule firing cannot trigger another rule firing until the former is finished. In fact rule firings in DJess are akin to transactions, and the system used to control concurrency is a frequent solution in transaction systems, the *two-phase lock* protocol[6] [14], yet without rollback. Rule firing execution is divided in three steps: locks acquisition, statements execution, and locks release. In the first step, a lock is acquired for every fact matched in the activation and for every fact that has a binding (i.e., a variable referencing it); if the acquisition of a lock fails, all locks are released and the firing is postponed, without any rollback since there is nothing to undo. The second step is the actual rule firing; actions are then performed

[4]Moreover, in so doing, the number of messages exchanged for the synchronization of the shared memory is minimized.

[5]DJess addresses only the consistency problems due to concurrency; those possibly arising from knowledge bases integration are left to IS designers.

[6]In a two-phase lock scheme no further lock can be acquired after having released any lock.

according to the rule statements and if an `assert` is executed, the new fact is created in a locked state[7]. In the last step, all locks are released and if a rule has been blocked by any of these locks, it is eventually fired.

Although shared facts are replicated in each IS's working memory, the coordination achieved through the ghost fact permits to conceive of the SWM as a distinct and unique entity. Ghost facts do not require to modify Jess's implementation of the Rete algorithm; no modifications are needed for the `modify` primitive either, due to the use of shadow facts. In Jess (hence in DJess), whenever `modify` is invoked on a shadow fact to change some slot values, the corresponding set methods of the shadowed Java bean are called. So in DJess, when a set method of a proxy is called in response to a `modify`, no proxy internal variable is updated; the method just calls the corresponding ghost fact set method. In so doing, the ghost fact updates its attributes and notifies all its other proxies of the modification by calling their set methods; consequently the modification is shadowed in all ISs' working memories through the native Jess mechanism.

Asserting and retracting shared facts involves some modifications of Jess mechanisms. Every time the engine of an IS tries to create a new template in the shared module in response to a `deftemplate` statement, the template is parsed and the Java code for two new classes, namely the proxy and the ghost fact classes, is generated, so that these classes have a bean property for each slot in the original template. The code is then compiled and sent along with the template to all the other WoIS members, that from then on are ready to assert shared facts using the new template. When the engine of an IS uses a shared template and attempts to assert a fact, it instantiates the associated ghost fact class and initializes this new object with the slot values of the fact; it then notifies all engines in the WoIS, sending them a reference to the ghost fact. Each engine creates a new proxy bound to the ghost fact and uses it to assert a shadow fact in its working memory. Likewise, when an engine retracts a shared fact, it notifies all the WoIS members' engines and they remove the corresponding shadow fact from their memory; finally also the ghost fact is removed from the system.

IV. Future Work and Conclusions

In this paper we have presented DJess, a scripting environment that enables programmers to write powerful rule-based inference systems. Such systems are able, even running on different machines, both to share elements of their working memory, so as to infer on remote facts as if they were local, and to exchange procedural knowledge (rules) so as to reach a better semantic interoperability on the base of common factual representations. Full integration with Java, transparency, and the adoption of a pure generative communication model make DJess quite different from DCLIPS, an infrastructure for inferential code mobility used in multi-robot systems [15] that to our knowledge is the closest work in approach and aims. Since the set of facts that constitutes any single working memory can represent the internal state of the IS as well as its symbolic description of the current situation, we intend to deploy such a system in an Ambient Intelligence domain, whose typical requirements and constraints drove our implementation choices. In fact, we think DJess can facilitate the design and writing of lightweight rule-based applications running on top of pervasive devices by providing off-the-shelf inference capabilities and a simple communication middleware through which context-aware systems can transparently share both representations of the current context and code to manage this knowledge properly.

Our future work will then address the refinement of the rule-based inference approach as a general methodology for AmI systems, and will address the further development and optimization of the security, resilience, and synchronization mechanisms involved in memory sharing and consistency guaranteeing.

At the current moment, DJess is in an alpha state and the programming interfaces have been fully developed[8]; we have realized some demonstrators to probe the reliability and the performance of the overall architecture. The beta release will be based on Jess 6.1 version, the Jess latest stable version to date.

[7]This is to prevent another engine from firing a rule on a new fact before the asserting rule execution has been completed.

[8]For a bird's eye view of the implementation, a provisional documentation is given at `http://www.mac.disco.unimib.it/djess`.

References

[1] Jess, the Java Expert System Shell, http://herzberg.ca.sandia.gov/jess/
[2] D. Gelernter, "Generative communication in Linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, 1985.
[3] N. Carriero and D. Gelernter, "Linda in context," *Communications of the ACM*, vol. 32, no. 4, pp. 444–458, 1989.
[4] E. Friedman-Hill, *Jess in Action – Java Rule-based Systems*. Manning Publications Co., 2003.
[5] M. Lindwer, D. Marculescu, T. Basten, R. Zimmermann, R. Marculescu, S. Jung, and E. Cantatore, "Ambient intelligence visions and achievements: Linking abstract ideas to real-world concepts." in *Proceedings of the conference on Design, Automation and Test in Europe (DATE '03)*, 2003, pp. 10–15.
[6] M. Weiser, "Some computer science issues in ubiquitous computing," *Commun. ACM*, vol. 36, no. 7, pp. 75–84, 1993.
[7] T. Ishida, "Parallel, distributed and multi-agent production systems – A research foundation for distributed artificial intelligence," in *Proceedings of the First International Conference on Multi-Agent Systems*, V. Lesser, Ed. San Francisco, CA: MIT Press, 1995, pp. 416–422.
[8] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.
[9] T. Ishida, "Parallel rule firing in production systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 3, no. 1, pp. 11–17, 1991.
[10] S. Kuo, "A parallel asynchronous message-driven production system," Ph.D. dissertation, University of Southern California, 1991.
[11] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding code mobility," *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342–361, May 1998.
[12] A. Acharya, "Eliminating redundant barrier synchronizations in rule-based programs," in *Proceedings of the 10th international conference on Supercomputing*. ACM Press, 1996, pp. 325–332.
[13] S. Tata, G. Canals, and C. Godart, "Specifying interactions in cooperative applications," in *In Eleventh International Conference on Software Engineering and Knowledge Engineering*, Kaiserslautern, Germany, June 1999.
[14] J. Gray and A. Reuter, *Transactions Processing: Techniques and Concepts*, M. Kaufmann, Ed., San Mateo, CA, USA, 1994.
[15] F. Amigoni, M. Somalvico, and D. Zanisi, "A theoretical framework for the conception of agency," *International Journal of Intelligent Systems*, vol. 14, no. 5, pp. 449–474, May 1999.