A Quantitative Assessment of the JADEL Programming Language

Federico Bergenti[1], Eleonora Iotti[2(✉)], Stefania Monica[1], and Agostino Poggi[2]

[1] Dipartimento di Scienze Matematiche, Fisiche e Informatiche,
Università degli Studi di Parma, 43124 Parma, Italy
{federico.bergenti,stefania.monica}@unipr.it
[2] Dipartimento di Ingegneria e Architettura,
Università degli Studi di Parma, 43124 Parma, Italy
eleonora.iotti@studenti.unipr.it, agostino.poggi@unipr.it

Abstract. This paper reports a quantitative assessment of JADEL, an agent-oriented programming language designed to implement JADE agents and multi-agent systems. The assessment is structured in two parts. The first part is intended to evaluate the effectiveness of JADEL for the concrete implementation of agent-based algorithms expressed using a pseudocode. The second part examines the functionality of the language regarding concurrency and message passing by comparing the implementation in JADEL of a set of benchmark algorithms with the corresponding implementations in Scala. The metrics introduced for the two parts of the assessment are meant to evaluate the expressiveness and ease of use of JADEL, and reported results are encouraging.

Keywords: Agent Oriented Programming · JADEL · JADE

1 Introduction

Agent-Oriented Programming (*AOP*) is the programming paradigm first introduced by Shoham in [40]. AOP identifies as core abstractions the autonomous and proactive entities known as (*software*) *agents*, and, over the years, several languages and tools have been developed to coherently support AOP and to provide advanced features for the development of agents and multi-agent systems. *Agent programming languages* is a class of programming languages, which includes AOP languages, that has gained significant relevance in the literature. The interest in agent programming languages dates back to the introduction of agent technologies and, since then, it has grown rapidly. As a matter of fact, agent programming languages turned out to be especially convenient to model and develop complex multi-agent systems, in contrast with traditional (lower-level) languages, that are often considered not suitable to effectively implement *Agent-Oriented Software Engineering* (*AOSE*) [13]. Nowadays, agent programming languages are widely recognized as important tools in the development of agent technologies and they represent an important topic of research

J. van den Herik et al. (Eds.): ICAART 2017, LNAI 10839, pp. 157–178, 2018.
https://doi.org/10.1007/978-3-319-93581-2_9

(see, e.g., [1,18]). Each agent programming language is usually based on a specific agent model, which is often formally defined, and aims at providing dedicated constructs to adopt the specific agent model at a high level of abstraction. Simplicity and ease of use are characteristics which made the success of agent programming languages among developers. In fact, thanks to such characteristics, agent programming languages allow developers to reduce the complexity of their work and to expedite the creation of agents and multi-agent systems.

Besides agent programming languages, other tools have been provided over the years to support the effective construction of agents and multi-agent systems. Agent platforms are examples of such tools which offer language-agnostic approaches to the development of agents and multi-agent systems. One of the most popular agent platforms is the *Java Agent DEvelopment framework* (*JADE*, jade.tilab.com) [2,3], which is a middleware that offers several APIs and graphical tools to support the development of distributed multi-agent systems. JADE can be considered a consolidated tool, and it is widely used both for industrial and academic purposes [30]. In particular, it has been used for many relevant research projects (see, e.g., [8,9,11,35], just to mention some recent projects of the authors), and it has been in daily use for service provision and management in Telecom Italia for more than six years, serving millions of customers in one of the largest broadband networks in Europe [10]. Moreover, JADE is considered a valid enabler for the use of agent technology in various application domains, such as agent-based social networks modeling [12] and localization [6,32,33]. As a notable feature, JADE supports the development of agents and multi-agent systems that are compliant with the specifications of *IEEE Foundation for Intelligent and Physical Agents* (*FIPA*, www.fipa.org), with a particular focus on *FIPA interaction protocols* [25].

Beside such considerations, JADE also owes its success to its pure Java approach to agent technologies. As a matter of fact, when JADE was conceived and developed, in the early 2000's, its main design decisions were based on the technologies that were most popular and promising at the time. Developers wanted to use Java, and the common opinion was that such a technology would have been able to change many aspects of software development processes. In such a context, a pure Java approach seemed to be a perfect choice for a software framework that aimed at becoming a solid and reliable tool. Nowadays, such a design choice is less appealing to developers of agents and multi-agent systems. Our experience in using agent technologies and teaching it to graduate students shows a slow regression of the appreciation on JADE for its intimate link with Java. In fact, Java does not natively support agent-oriented technologies and methodologies, and this is perceived as a limitation and a source of errors. Moreover, JADE is constantly expanding and its continuous growth—in terms of features, and available APIs—increases the complexity of the framework. As a result, there is a high number of implementation details that a developer must handle to build a non-trivial multi-agent system. In order to address such problems, we are working on automated tools to help the analysis and verification

of JADE agents and multi-agent systems on the basis of a formal operational semantics that we are developing [5,16].

In a plan of simplification and renovation of the experience of using JADE, the introduction of a specific AOP language seems an appropriate option. The *JADE Language* (*JADEL*) is an AOP language based on JADE which is meant to simplify the work of JADE users, at least, in some aspects of the development process. JADEL provides abstractions and constructs which focus on relevant agent-oriented features of JADE, and it aims at simplifying the adoption of such features at a high level of abstraction. A first description of JADEL can be found in [7], and more recent developments are discussed in [17], where an overview of current syntax and semantics is presented. Then, in [4,14], the JADEL support for FIPA interaction protocol, which was not included in the first versions of the language, is presented. Finally, [15] is a first attempt at assessing the features of the language by discussing the use of JADEL for the implementation of a non-trivial agent-based algorithm. The objective of this last exercise is to illustrate the steps that a programmer needs to follow from pseudocode to implementation, and to analyze the effort spent in doing such a task when JADEL is adopted. Due to the distributed nature of JADE, the algorithm chosen as case study is a well-known procedure for solving distributed constraint satisfaction problems, the *Asynchronous BackTracking* (*ABT*) algorithm. In [15], the source code written with JADEL of the ABT algorithm is compared with the original pseudocode from [44,45], and some considerations on the effectiveness of the use of JADEL for this task are presented. In this paper, the problem of translating a known pseudocode into a working program, and of evaluating the effectiveness of JADEL in such a task, is further investigated. The main steps of the translation from the original pseudocode to JADEL source code are recalled, and relevant metrics for the evaluation of programming languages are applied to this case. Moreover, in order to investigate the features of JADEL in terms of support to concurrency and message passing, accepted benchmark algorithms found in the Savina benchmark suite [29] are implemented in JADEL and compared against known implementations in Scala.

The paper is organized as follows. Section 2 briefly reports on some of the most popular agent programming languages to overview related work. Section 3 shows the JADEL programs used to support the proposed quantitative evaluation. In particular, it shows the implementation of the ABT pseudocode and of selected programs of the Savina benchmark suite. Section 4 uses the presented JADEL source codes to discuss a quantitative evaluation of JADEL. Finally, a brief recapitulation of major presented results concludes the paper.

2 Related Work

The obvious collocation of JADEL is in the wide scope of agent programming languages, but JADEL is also a *Domain-Specific Language* (*DSL*) and it should be treated as such. The wide range of technologies and tools involved in the development of a DSL is brightly discussed in [31,34], where DSLs are clearly

marked as important tools to support *model-driven development*. Such works also provide in-depth analyses of the motivations that may lead developers to decide in favor of a new DSL, which is a difficult decision because of the inherent costs of DSLs in terms of implementation and maintenance. Nevertheless, the use of a DSL for a specific application domain leads to important benefits. As a matter of fact, the syntax of a DSL is tailored on the specific domain that it describes, with the aid of user-friendly notations that are simpler than the respective general-purpose ones. This facilitates code understanding, and it allows many repetitive and tedious activities to be automated. Moreover, DSLs are meant to be easily integrated with a host language, which is typically a general-purpose programming language, and this fact ensures the applicability and reusability of domain-specific code in real-world scenarios, where the interoperability with existing code is essential. All such benefits justify the design of JADEL as a DSL for AOP with the intent to increase the adoption of JADE in model-driven development. Notably, the approach of designing an new agent programming language as a DSL for agent-oriented programming has been adopted by other languages mentioned below.

The features of agent programming languages may differ significantly, concerning, e.g., the selected agent mental attitudes (if any), the integration with an agent platform (if any), the underlying programming paradigm, and the underlying implementation language. In order to compare the characteristics of different agent programming languages and to provide a clear overview of the state of the art, it is worth recalling accepted classifications of relevant agent programming languages that have already been proposed. [1] classifies agent programming languages on the basis of the use of mental attitudes. According to such a classification, agent programming languages can be divided into: AOP languages, *Belief Desire Intentions* (*BDI*) languages, hybrid languages, which combine the two previous classes, and other languages, which fall outside previous classes. It is worth noting that such a classification recognizes that BDI languages follow the AOP paradigm, but it reserves special attention to them for their notable relevance in the literature. [18] proposes a different classification, where languages are divided into declarative, imperative, and hybrid. Declarative languages are the most common because they focus on automatic reasoning, which is theme closely related to agent technologies. Some relevant imperative languages have also been proposed, and most of them were obtained by adding agent-oriented constructs to existing procedural programming languages. In the rest of this section, a list of the most popular agent programming languages and their features is given, in chronological order.

Shoham introduced the AOP paradigm in [39] together with his appreciated AGENT-0 language [38]. One of the direct descendant of AGENT-0 is the language called *PLAnning Communicating Agents* (*PLACA*). It extends the capabilities of AGENT-0 by providing improved syntax and new mental categories. Just like AGENT-0, PLACA has experimental nature and it was not meant for practical use. Another important, yet experimental, language is Concurrent MetateM [24], which is an agent programming language based on

temporal logics. Another important example of classic agent programming languages is AgentSpeak(L), whose syntax and semantics were formalized by Rao in [36]. The proposed formalization of AgentSpeak(L) is based on the BDI agent model. Other agent programming languages based on the BDI model are *An Abstract Agent Programming Language* (*3APL*) [27], which includes features of both imperative and logic programming languages, and the *JACK Agent Language* (*JAL*), which is built on top of JACK platform [41], an environment to develop multi-agent systems in which agents are based on the BDI paradigm. Another software framework that implements a BDI-based reasoning engine is Jadex [19]. It combines declarative and imperative approaches by using an XML specification language to define beliefs, goals and plans, and by using Java as procedural language to implement plans. Then, *A Computational Language for Autonomous Intelligent and Mobile Agents* (*CLAIM*) [23] is an agent language that supports agent mobility. While, the *Semantic web-Enabled Agent Language* (*SEA_L*) [20–22] is a DSL to model and develop multi-agent systems in the scope of the Semantic Web. Finally, SARL [37] is one of the latest entries in the plethora of agent programming languages. It is a general-purpose imperative language with an intuitive syntax, and it can be considered platform-agnostic, even if it is commonly used with the dedicated agent platform called Janus.

3 Implementations of Selected Algorithms in JADEL

JADEL supports four main agent-oriented abstractions, namely, *agents*, *behaviours*, *communication ontologies*, and *roles* in interaction protocols. Actually, agents in JADEL use ontologies and behaviours, and they take roles in interaction protocols.

A JADEL agent can be defined by using the keyword `agent` followed by its name. It has a life cycle that consists in a start-up phase followed by an execution phase, and it is eventually terminated by a take-down phase. The declaration of an agent is allowed to extend the declaration of another agent, with the usual semantics of inheritance, and two event handlers are provided to support initialization and take-down phases, namely, the `on-create` and the `on-destroy` handlers. During agent initialization, a sequence of tasks, called behaviours following accepted JADE nomenclature, can be added to an internal list by means of the `activate-behaviour` expression. After the start-up phase, actions specified in such behaviours are performed by the agent. New tasks can be added dynamically during the life cycle of the agent, and tasks that are no longer needed can be removed.

Behaviours for JADEL can be of two types, namely `cyclic` or `oneshot`. Cyclic behaviours represent actions that remain in the behaviour list of an agent after their execution. This means that the action of a cyclic behaviour can be used one or more times during the life cycle of the agent. A one-shot behaviour, instead, contains an action which terminates immediately and it is removed from the list of the agent after just one execution. The action of a behaviour can be an auto-triggered action, i.e., it starts immediately after its behaviour is chosen

by the agent, or it can be triggered by an event, e.g., the reception of a message. Message reception is handled by means of a specific construct of JADEL, the `on-when-do` construct, which also provides a control over the type of message the agent intends to receive.

Ontologies and interaction protocols are used in agent communication. In particular, ontologies provide formal means to support the semantics of the adopted agent communication language. Ontologies represent one of the most tedious and error prone tasks in the development of multi-agent systems with JADE, and JADE users tend to agree that the large amount of implementation details and repetitive idioms involved in ontology classes shift the focus on technical parts rather than on the semantics of involved ontology elements. For this reason, JADEL provides a special lightweight syntax for ontologies and it permits the automation of many repetitive tasks. As a matter of fact, a JADEL ontology is defined as a set of concepts, predicates, propositions and actions. Such terms compose a sort of dictionary, which is usually organized in a hierarchical structure. Agents sharing such a dictionary can interact by using common terms as content of their messages.

Besides ontologies, JADEL support structured communication by means of a specific constructs to allow agents taking roles in FIPA interaction protocols. Roles are particular behaviours, which are composed of a set of predefined event handlers. Each of such handlers covers a different step of the interaction protocol, by filtering messages through their performatives, as expected from FIPA specifications.

3.1 Implementation of the ABT Algorithm

The *Asynchronous BackTracking* (*ABT*) [43] algorithm is a well known algorithm to solve *Distributed Constraint Satisfaction Problems* (*DCSPs*) [43]. DCSPs are distributed variants of constraint satisfaction problems and, as such, a DCSP consists in a finite set of variables and a finite set of constraints over such variables. As in [43], we denote variables as $x_1, x_2, \ldots, x_n$. Each variable x_i takes values in a domain, called D_i. Constraints subsets of $D_1 \times \cdots \times D_n$, and a DCSP is *solved* if and only if a value is assigned to each variable, and each assignment satisfies all constraints. In a DCSP constraints and variables are distributed among agents. Such agents manage a number of variables and they know the constraints over managed variables. Commonly, each agent is associated with just one variable, and it finds an *assignment* of its variable, i.e., a pair (x_i, d) where $d \in D_i$, that satisfies involved constraints. The interactions in the multi-agent system allows each agent to obtain the assignments of other agents, and to check if constraints are really satisfied. Informally, a DCSP is solved if each agent finds a local solution that is consistent with the local solutions of other agents. In [44], a survey of the main algorithms for solving DCSPs is given. In particular, pseudocode and examples are shown for the ABT, the asynchronous weak-commitment search, the distributed breakout, and the distributed consistency algorithms.

The ABT algorithm solves DCSPs under three assumptions: each agent owns exactly one variable, all constraints are in the form of binary predicates, and each agent knows only the constraints that involve its variable. Because it is not necessarily true that all agents in a multi-agent system know each others, they can communicate only if there is a connection between the sender and the receiver of a message. For each agent, the agents who are directly connected with it are called its *neighbors*. In ABT, each agent maintains an *agent view*, which is the agent local view of its neighbors assignments. Communication is addressed by using two types of messages, *OK* and *NoGood*, which work as tools to exchange knowledge on assignments and constraints. More precisely, OK messages are used to communicate the current value of the sender agent variable, and NoGood messages provide the recipient with a new constraint. Agents are associated with a priority order, which can be, e.g., the alphabetical order of their names (or variables). OK messages flow from top to bottom of the priority list of agents, and NoGood messages, instead, go up from lowest priority agents to highest ones. Core of the algorithm is the *check agent view* procedure, which controls if the current known assignments are consistent with the agent value. If not, procedure *backtrack* is used to send NoGood constraints to neighbors. The rest of the algorithm is given in terms of event handling constructs which react at other agents messages.

The ABT algorithm was originally described using a pseudocode [44]. For the sake of brevity, the pseudocode is not reproduced here. The proposed implementation in JADEL follows precisely the original pseudocode. The presentation of the JADEL source code is structured into the presentation of the ontology, of the agents, of support procedures and of event handlers, as follows.

Ontology. An important entity in JADEL is the ontology. From ABT pseudocode, messages are divided into different categories, but there is no specification or definition of an ontology. JADEL takes advantages from a light syntax for defining communication means which describes how agents could interoperate in a given application. The ontology for ABT algorithm includes propositions, concepts, and predicates, as shown in the JADEL code below.

```
ontology ABTOntology {
    concept Assignment(aid index, integer value)
    predicate OK(Assignment assignment)
    predicate NoGood(many Assignment assignmentList)
    proposition NoSolution
    proposition Neighbor
    predicate Solution(many Assignment assignmentList)
}
```

The assignment is a central concept in ABT algorithm. Its implementation consists in the definition of an ontology term which is composed of an agent identifier, i.e., x_i, and the value of its variable, i.e. d_i, called `index` and `value`, respectively. The two predicates used in the main part of the algorithm, namely, the `OK` and the `NoGood` predicates, are defined on the basis of the definition of the

`Assigment`. In fact, an OK message is the current assignment of the agent, while the NoGood message is a sequence of forbidden assignments. Also a predicate `Solution` is defined, which is used to communicate to other agents the solution of the problem, when found. `NoSolution` and `Neighbor` are simply propositions, that agents can exchange to indicate the algorithm termination with no solutions, and the neighbor request, respectively.

Agents. ABT pseudocode describes event handlers and main procedures, but it does not illustrate how agents should be written. In JADEL, an agent must be defined. Such an agent is called `ABTAgent`. It consists of some properties, among which there are the agent view and the set of neighbors. The initialization of an `ABTAgent` is done by filling the set of neighbors with the identifiers of connected agents, and by setting the priority of the each agent. Moreover, `ABTAgent` provides two important methods, namely, `checkConstraints` and `assignVariable`. The first checks if all constraints are satisfied by current assignments in agent view, while the second selects a value which is consistent with agent view and assigns it to the variable owned by the agent. Both methods return `true` if the operation was successful and `false` if it is was not.

Procedures. The core procedure of the ABT algorithm is the *check agent view* procedure, which controls if the current value $my\_value \in D_i$ of the agent x_i is *consistent* with its agent view. A value $d \in D_i$ is called consistent with the agent view if for each value in agent view, all constraints that involve such value and d are satisfied. If this is not the case, the agent has to search for another value. At the end, if none of the values in D_i satisfies the constraints, another procedure is called, namely, the *backtrack* procedure. Otherwise, an OK message is sent to the agent neighbors, which contains the new assignment. In the JADEL implementation of the ABT algorithm, the *check agent view* procedure becomes a one-shot behaviour. In fact, its action has to be performed only once, when the behaviour activates, as follows.

```
oneshot behaviour CheckAgentView for ABTAgent {
```

The keyword `for` denotes which agents are allowed to activate such a behaviour. In this case, such agents are instances of the `ABTAgent` class. Inside the behaviour, methods and public fields of the agent can be called by using the field `theAgent`, which is implicitly initialized with an instance of the agent specified. If no agent is specified with the `for` keyword, `theAgent` refers to a generic agent. The `CheckAgentView` behaviour does not need to wait for messages, or events, so the keyword `do` is used, as follows.

```
do {
    if(!theAgent.checkConstraints()) {
        if(!theAgent.assignVariable()) {
            activate behaviour Backtrack(theAgent)
        } else {
            activate behaviour SendOK(theAgent)
        }
    }
}
```

The procedure *backtrack* is meant to locally correct inconsistencies. First, a new NoGood constraint has to be generated. Generating a NoGood is done by checking all assignments that are present into the agent agent view. If one of these is removed, and then the agent succeeds in choosing a new value for its variable, it means that such an assignment is wrong. Hence, that assignment is added to the NoGood constraint. After this phase, the new generated NoGood can be empty or not. If no assignment appears within that new constraint, then there is no solution for the DCSP. Otherwise, a NoGood message has to be sent to the lowest priority agent, and then its assignment has to be removed from agent view. Then, a final check of the agent view is done. JADEL implementation of such a procedure is another one-shot behaviour, whose code follows precisely the original pseudocode of the algorithm.

```
oneshot behaviour Backtrack
 for ABTAgent {
  do {
        var V = new HashMap<AID, Integer>(theAgent.agentview)
        var sortedVariablesList = V.keySet.sort
        V.remove(theAgent.AID)

        for(v : sortedVariablesList) {
            var removed = V.remove(v)
            if(theAgent.assignVariable(V)) V.put(v, removed)
        }

        if(V.isEmpty){
            activate behaviour SendNoSolution(theAgent)
        } else {
            activate behaviour SendNoGood(theAgent, V)

            theAgent.agentview.remove(V.keySet.max)

            activate behaviour CheckAgentView(theAgent)
        }
    }
}
```

Event Handlers. Others procedures specified in the original ABT pseudocode concern the reception of messages. When the agent receives an OK message, it has to update its agent view with that new information, then it must check if the new assignment is consistent with others in agent view. The reception of a message requires a cyclic behaviour, which waits cyclically for an event and checks if such an event is a message.

```
cyclic behaviour ReceiveOK for ABTAgent {
```

To ensure that such a message is the correct one, namely, an OK message, some conditions have to be specified. JADEL provides the construct `on-when-do` to handle this situation. The clause `on` identifies the type of event and eventually gives to it a name. If the event is a message, the clause `when` contains an expression that filters incoming messages, as follows.

```
on message msg
when {
    ontology is ABTOntology and
    performative is INFORM and
    content is OK
}
```

Conditions in `when` clause can be connected by logical connectives `and`, `or`, and they can be preceded by a `not`. They refer to the fields of the message, namely, `ontology`, `performative`, and `content`. Fields that are not relevant can be omitted, and multiple choices can be specified. For example a behaviour can accept `REQUEST` or `QUERY_IF` messages with `performative is REQUEST or performative is QUERY_IF`. The clause `do` is mandatory and contains the code of the action.

```
do {
    extract receivedOK as OK

    val a = receivedOK.assignment

    theAgent.agentview.replace(a.index, a.value)

    activate behaviour CheckAgentView(theAgent)
}
```

The content of the message is obtained by means of the JADEL expression `extract- as`, which manages all the needed implementation details and gives a name and a type to the content. Once the content of type `OK` of the message is obtained, its assignment is used to revise the agent view. Then, the behaviour `CheckAgentView` is activated.

Finally, the pseudocode of the procedure that manages the reception of a NoGood message is a cyclic behaviour for `ABTAgent`.

```
cyclic behaviour ReceiveNoGood for ABTAgent {
```

Checking if the event is a message, and then, if the message is actually a NoGood message, is done similarly to the OK reception, by using the clauses `on` and `when`, as shown in the following code.

```
on message msg
when {
    ontology is ABTOntology and
    performative is INFORM and
    content is NoGood
}
```

Inside the `do` body, the message content is extracted as a NoGood and it is recorded as a new constraint. We assume that the agent holds a set of constraints within the field `constraint` which is accessed by the agent instance `theAgent`.

```
do {
    extract receivedNoGood as NoGood

    val newConstraints = receivedNoGood.assignmentList

    theAgent.constraints.putAll(newConstraints)
```

Then, if some constraints involve an agent which is not in the agent neighborhood, a request is sent to such an agent, in order to create a new link.

```
for (x : newConstraints.keySet) {
    if (!theAgent.neighbors.contains(x)) {
        activate behaviour SendRequest(theAgent, x)

        theAgent.neighbors.add(x)
    }
}
```

Finally, the agent view must be checked, and if the previous value of the agent variable x_i remains unchanged, an OK message is sent.

```
var oldValue = theAgent.agentview.get(theAgent.AID)

activate behaviour CheckAgentView(theAgent)

if (oldValue == theAgent.agentview.get(theAgent.AID)) {
    activate behaviour SendOK(theAgent)
}
```

3.2 Implementation of Savina Benchmarks

Other evaluations on JADEL are made by comparing it using the Savina benchmarks [29]. Savina is a benchmark suite to test actor libraries performances, and the source code of the thirty Savina benchmarks can be

found at github.com/shamsimam/savina. For each benchmark, Savina provides an implementation by using the actor features of Akka [42], Functional-Java (www.functionaljava.org/), GPars (www.gpars.org/), Habanero-Java library [28], Jetlang (github.com/jetlang), Jumi (jumi.fi/actors.html), Lift (liftweb.net/api/26/api/#net.liftweb.actor.LiftActor), Scala [26], and Scalaz (github.com/scalaz). The thirty benchmarks that Savina provides are divided into *classic micro-benchmarks*, *concurrency benchmarks* and *parallelism benchmarks*.

Micro-benchmarks are simple benchmarks which test specific features of an actor library. For example, the classic `PingPong` benchmark measures the message passing overhead, while the `Counting` benchmark tests message delivery overhead. Concurrent benchmarks focus on classic concurrency problems, such as the dining philosophers, and they represent more realistic tests than micro-benchmarks. Finally, parallelism benchmarks exploit pipeline parallelism, phased computations, divide-and-conquer style parallelism, master-worker parallelism, and graph and tree navigation. In [29], the scope and the characteristics of each benchmark are discussed, and some experimental results are shown. It is worth noting that Savina is a suite that helps testing actor-oriented solutions, and it does not consider agent-oriented features. Nevertheless, Savina benchmarks are also suitable to analyze some features of agent programming languages, such as concurrency and message passing. For this reason, in this paper we take a few benchmarks from those proposed by Savina, and re-implemented them in JADEL. Savina does not yet contains inter-languages comparisons. As a matter of fact, sources are written in Java and Scala, and all benchmarks shows almost the same code: the differences among them are due to the various actor implementations. Additional language comparisons could be useful to evaluate the elegance, the readability and the simplicity of a given solution, beside its performances. Only a few Savina micro benchmarks are considered here, namely, the `PingPong`, `ThreadRing`, `Counting`, `Big`, `Chameneos` benchmarks.

It is worth noting that Savina benchmarks are thought for actor-based systems, and thus are heavily based on message passing. JADEL ontologies help in managing such task effectively. In the JADEL source code below, the simple ontology used for implementing the `PingPong` example is shown. The commented parts are the identifiers of the message objects which Savina implementation defines and uses for messages.

```
ontology PingPongOntology {
    proposition Start // PingPongConfig.StartMessage
    proposition Ping  // PingPongConfig.SendPingMessage
    proposition Pong  // PingPongConfig.SendPongMessage
    proposition Stop  // StopMessage
}
```

The `PingPong` classic example consists in the definition of two agents which uses such an ontology, exchanging N `Ping` and `Pong` messages, alternatively. In the following listing, the source code of the ping agent, i.e., the initiator agent, is shown.

```
agent PingAgent uses ontology PingPongOntology {
    var AID pongAgent

    on create {
        pongAgent = newAID(arguments.get(0) as String)

        activate behaviour WaitForStartOrPong(this,
                PingPongConfig.N)
    }
}
```

Then, the pong agent, i.e., the responder agent has the following source code in JADEL.

```
agent PongAgent uses ontology PingPongOntology {
    var AID pingAgent

    on create {
        pingAgent = newAID(arguments.get(0) as String)

        activate behaviour WaitForPingOrStop(this, 0)

        activate behaviour SendInformMsg(this, #[pingAgent],
                new Start)
    }
}
```

Similarly, `ThreadRing` agents are defined. In this benchmark, N agents exchange R `Ping` messages, and they are limited to communicate only with the next agent in the ring. As for the `PingPong` benchmark, an ontology which follows precisely the Savina structure of message object is defined. In this example, message content are predicates rather than propositions, because they need to carry information between involved agents.

```
ontology ThreadRingOntology {
    predicate Ping(integer left) // ThreadRingConfig.PingMessage
    predicate Data(aid next)     // ThreadRingConfig.DataMessage
    predicate Exit(integer left) // ExitMessage
}
```

The JADEL source code for agent definition is listed below.

```
agent ThreadRingAgent extends JadelBaseAgent
      uses ontology ThreadRingOnto {
    var int id
    var AID na

    on create {
        na = newAID(arguments.get(0) as String)
        id = arguments.get(1) as Integer

        activate behaviour WaitForMsg(this)

        if (id == ThreadRingConfig.N - 1) {
            activate behaviour SendInformMsg(this, #[na],
                    new Ping(ThreadRingConfig.R))
        }
    }
}
```

As an example of `cyclic` behaviour, the following code shows the reception of an increment message in the `Counting` example. In this example, a `Producer` agent sends N increment messages to a `Counter` one, which counts the number of arrived messages. When the counter agent received all messages, it must inform the other agent of the resulting value of its count. As we can see, the behaviour `WaitForMsg` is a `cyclic` behaviour, as in ABT event handlers, because it must wait for a message and repeat its action each time a message arrives. For this scope, the construct `on-when-do` is used, as follows.

```
cyclic behaviour WaitForMsg for Counter {
    on message msg
    when {
        content is Increment
    } do {
        theAgent.count = theAgent.count + 1

        if (theAgent.count >= CountingConfig.N) {
            activate behaviour SendInformMsg(theAgent,
                    #[theAgent.producerAgent],
                    new ResultingValue(theAgent.count))

            activate behaviour Delete(theAgent)
        }
    }
}
```

Other micro benchmarks are implemented in the same fashion, with

1. A definition for each different kind of agent involved, which activates needed behaviours in the start up phase of its life cycle by means of the `on-create` handler;

2. The definition of a number of `cyclic` behaviour whose purpose is to intercept messages and process the correct ones; and
3. The definition of an ontology which terms are equivalent to the Savina ones.

Hence, the methodology used in [15] for implementing the ABT algorithm is the common way to creating agents and multi-agent systems with JADEL, whether the example is a very simple one (e.g., the `PingPong`), or a more complex algorithm as ABT.

4 Experimental Results

Methods to evaluate DSLs can be found in, e.g., [20], which focuses on multi-agent systems. Other surveys, such as [31,34], highlight the main advantages of the use of DSLs.

The comparison between the ABT pseudocode and its JADEL implementation is done by defining some metrics, which help us to get an idea of JADEL advantages and disadvantages. Then, we compare JADEL code with an equivalent JADE code, measuring the amount of code written, and the percentage of agent-oriented features of such a code. Nevertheless, comparing a pseudocode with an actual implementation is a difficult task, due to the informal nature of the pseudocode, and the implicit technical details it hides. Moreover, pseudocodes from different authors may look different, depending on their syntax choices and their purposes. As far as we know, there are not standard methods for evaluating the closeness of a source code to a pseudocode, and its actual effectiveness in expressing the described algorithm. Hence, we limit our evaluation to the use case of JADEL shown in this paper: the ABT example presented in previous section.

The first consideration that is made in evaluating the JADEL implementation of ABT is that ABT pseudocode is presented by means of procedures and event handlers, with the aid of the keywords `when` and `if`. As a second consideration, the notation used inside the ABT pseudocode is the same of the DCSP formalization. As a matter of fact, there are *agentview* and *neighbors* sets, and assignments are denoted as (x_i, d_i), where x_i is the variable associated with the i-th agent, and $d_i \in D_i$. A message is identified according to its type and its content, i.e., $(OK, (x_i, d_i))$ for an OK message, or $(nogood, (x_i, V))$ for a NoGood. Such characteristics of ABT pseudocode allow us to talk about *similarity* between it and the JADEL source code. In fact, in JADEL, both procedures and event handlers are represented as behaviours of the agent. In particular, procedures are one-shot behaviours that define an auto-triggering actions, while event handlers are cyclic behaviours, each of them waits for the given event and then performs its action. Hence, we can associate each behaviour with a procedure or an event handler, and analyze each of them separately. Moreover, calls to procedures in ABT pseudocode translate into the activation of the corresponding behaviour in JADEL. Also, the sending of a message is done by activating a specific JADEL behaviour. Hence, we associate each *send* instruction in ABT pseudocode to that activation. The DCSP notation is used also in JADEL,

by means of the two maps, *theAgent.agentview* and *theAgent.neighbors*, and by defining some ontology terms. As a matter of fact, terms *OK* and *NoGood* are predicates in a JADEL ontology, and they contain an assignment, and a list of assignments, respectively. Each assignment consists in a *index* and a *value*, i.e., x_i and d_i, respectively. The domain D_i of a variable is defined once in the start-up phase of the agent and it is never modified during the execution of its actions. We associate ABT pseudocode notations with the respective JADEL notation described above. Finally, the reception of a message is done by using the construct `on-when-do`, which is the corresponding of ABT pseudocode construct **when** *received*(...) **do**.

We will say, in the following, that a line of ABT pseudocode *corresponds* to a line (or, a set of lines) of JADEL implementation, if it falls in one of the previous cases. Then, for each line of ABT pseudocode, we measure the number of the corresponding *Lines Of Code* (*LOC*) of JADEL implementation. The absolute value of the difference between ABT lines and corresponding JADEL LOC is used as a first, rough, distance. For example, in the reception of an *OK* message, the first line of the pseudocode corresponds to the `on-when-do` constructs to capture the correct event, and filter other messages that are not complied with the expected structure, as follows.

```
on message msg
when {
    ontology is ABTOnto and
    performative is INFORM and
    content is OK
}
```

Moreover, the `extract-as` expression is used to obtain the message content.

```
extract receivedOK as OK
```

Hence, we can conclude that in this case there are six LOCs instead of one line of the pseudocode. Thus, the distance is of five LOCs. Such a distance gives us an idea of the amount of code which is necessary to translate pseudocode into JADEL, in case of ABT example. A summary is shown in [15] where a count of nested blocks also presented. ABT pseudocode and JADEL implementation do not differ significantly in terms of nested blocks, and JADEL code often requires one more level (the `do` block), but its structure is usually very similar to ABT pseudocode.

The count of nested blocks makes more sense when JADEL code is compared to the equivalent JADE one. Such an equivalent implementation is obtained directly from the available JADEL compiler [7], which translates JADEL code into Java and uses JADE APIs. In fact, JADEL entities translate into classes which can extend JADE `Agent`, `CyclicBehaviour`, `OneShotBehaviour`, and `Ontology` base classes, while JADEL event handlers translate into the correct methods of JADE APIs, in order to obtain the desired result. JADE code is automatically generated from the JADEL one, and this means that the final code may introduce some redundancy or overhead. For this reason, we also write a

JADE code that implements ABT algorithm directly. Nevertheless, this alternative implementation is as complex as JADE generated code, because of some implementation details that JADE requires.

A comparison between JADEL and JADE implementation is made in terms of amount of code, i.e., by counting the number of non-comment and non-blank LOCs of each entity, namely, the `ABTAgent`, the `ABTOntology`, and all the behaviours. Results are shown in [15]. In order to emphasize the advantage in using JADEL instead of JADE, the percentage of lines which contains agent-oriented features over the total number of LOCs is also shown in [15]. We define as agent-oriented features each reference to the agent world. For example, keywords `agent`, `behaviour`, `ontology` are agent-oriented features, but are also special expressions such as `activate-behaviour`. In JADE, agent-oriented features are simply the calls to the API. [15] shows that the JADEL implementation is far lighter than the JADE one, and that it is more dense in terms of agent-oriented features. Such measures can be viewed as an indication of simplicity of JADEL code with respect to JADE.

The comparison between the chosen Savina benchmarks and their JADEL implementation is done by using the metrics of LOCs, with some restrictions. As in the ABT case, JADEL code is also compared with an equivalent JADE code, in terms of amount of code written. The main problem here is the different structure of a JADEL implementation, developed by using the JADEL approach, and the structure of a benchmark in Savina. Savina and JADEL projects are analyzed in terms of file written, utilities, and base classes, and only relevant parts of the benchmarks are evaluated (for example, configuration files are not counted). For a deeper evaluation, JADEL ontologies and Savina message objects are treated separately.

Savina benchmarks are structured as follows. There is a Java-written file of configurations, where parameters are initialized and managed (e.g. the number of pings N for the `PingPong` example), and objects for message passing are implemented. There is also a Scala source code that contains the implementations of actors, a class which implements the benchmark, i.e., a class that manages the iteration and the cleanup phases of each test, and an entry point for the benchmark. Similarly, JADEL benchmarks are structured as follows. The configuration file is the same as Savina. There is a Java file that implements the benchmark and the entry point. The most important, there is a JADEL file which contains the agents that are used in such a benchmark, their behaviours and an ontology for agent communications. It is worth noting that ontology predicates, concepts and propositions completely substitute that objects in the configuration file which are used in Savina for message passing. So, the JADEL implementation uses only a part of such a file, for getting parameter values, and does not take advantage of message objects.

Then, all benchmarks share a common base of methods and utilities. In Savina, for each considered actor framework, an actor base class is implemented.

Table 1. Number of LOCs, for Scala, JADEL and JADE implementation of selected examples from Savina benchmark suite.

	JADEL	Scala	JADE (generated source)
Base	55	68	197
PingPong	62	73	295
ThreadRing	46	53	239
Counting	72	40	308
Big	90	76	407
Chameneos	121	112	574
Philosopher	108	64	503

In JADEL, a base agent with two behaviours is implemented. Finally, Scala, JADEL and JADE LOCs are calculated using the following rules:

1. Blank lines or comments are not counted;
2. Prints for debugging are not counted;
3. Regarding Savina benchmarks, actors implementations are counted and also the definition and implementation of `Message` objects;
4. Regarding JADEL, the agent, behaviour and ontology implementations are counted; and
5. Regarding JADE, all generated files are counted.

Each measurement of the Savina suite is done by considering the Scala actor implementation of the benchmark. In Table 1, LOCs of some examples are shown, namely, the `PingPong`, `ThreadRing`, `Counting`, `Big`, `Chameneos` and `Philosopher` benchmarks. Table 2 emphasize the fact that JADEL syntax for ontologies is very light and the number of LOCs for ontologies remains little for each example, as opposed to Savina message objects or JADE ontologies.

Table 2. Number of LOCs, for Scala, JADEL and JADE implementation of ontologies and messages.

	JADEL ontology	Scala message objects	JADE ontology
Base	4	0	37
PingPong	6	29	57
ThreadRing	5	30	94
Counting	5	3	62
Big	5	21	47
Chameneos	7	32	141
Philosopher	7	6	97

5 Conclusions

In this paper, a quantitative evaluation of the agent-oriented programming language JADEL was presented. First, AOP paradigm and agent programming languages were briefly recalled, and JADEL was shortly presented. Then, a direct translation of the ABT pseudocode was presented together with a lighter presentation of the implementation of selected Savina benchmarks in JADEL. Such implementations were finally used quantitatively assess the performance of JADEL using specific metrics intended to evaluate the conciseness of the language and possible performance overheads introduced.

Not all adopted metrics can be regarded as complete in every situation, because they cannot fully describe qualitative factors, such as readability, reusability or maintainability. In particular, some of them heavily depend on the type of pseudocode given. However, such measurements help us at evaluating the simplicity of written code and they gives us an idea of the expressiveness and effectiveness of the language. As a matter of fact, the distance of JADEL from pseudocode and from Scala source code is very small. This fact may help JADE developers in translating an idea of distributed algorithms into a working JADE multi-agent system. Moreover, the distance from pseudocode and the number of nested blocks is very high in the case of JADE. This is mainly due to the very high number of implementation details that hide behind JADEL code, and the structure itself of Java language and JADE APIs.

References

1. Bădică, C., Budimac, Z., Burkhard, H.D., Ivanovic, M.: Software agents: languages, tools, platforms. Comput. Sci. Inf. Syst. **8**(2), 255–298 (2011)
2. Bellifemine, F., Bergenti, F., Caire, G., Poggi, A.: JADE - a Java agent development framework. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming: Languages, Platforms and Applications. MASA, vol. 15, pp. 125–147. Springer, Boston (2005). https://doi.org/10.1007/0-387-26350-0_5
3. Bellifemine, F., Caire, G., Greenwood, D.: Developing Multi-agent Systems with JADE. Wiley Series in Agent Technology. Wiley, Chichester (2007)
4. Bergenti, F., Iotti, E., Monica, S., Poggi, A.: Interaction protocols in the JADEL programming language. In: Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!) (2016)
5. Bergenti, F., Iotti, E., Poggi, A.: An outline of the use of transition systems to formalize JADE agents and multi-agent systems. Intelligenza Artificiale **9**(2), 149–161 (2015)
6. Bergenti, F., Monica, S.: Location-aware social gaming with AMUSE. In: Trends in Practical Applications of Scalable Multi-Agent Systems, the PAAMS Collection (PAAMS 2016), pp. 36–47 (2016)
7. Bergenti, F.: An introduction to the JADEL programming language. In: Proceedings of the IEEE 26th International Conference on Tools with Artificial Intelligence (ICTAI), pp. 974–978. IEEE Press (2014)

8. Bergenti, F., Caire, G., Gotta, D.: Agents on the move: JADE for Android devices. In: Proceedings of the Workshop Dagli Oggetti Agli Agenti (WOA 2014). CEUR Workshop Proceedings, vol. 1260 (2014)
9. Bergenti, F., Caire, G., Gotta, D.: An overview of the AMUSE social gaming platform. In: Proceedings of the Workshop Dagli Oggetti agli Agenti (WOA 2013). CEUR Workshop Proceedings, vol. 1099 (2013)
10. Bergenti, F., Caire, G., Gotta, D.: Large-scale network and service management with WANTS. In: Industrial Agents: Emerging Applications of Software Agents in Industry, pp. 231–246. Elsevier (2015)
11. Bergenti, F., Franchi, E., Poggi, A.: Agent-based social networks for enterprise collaboration. In: Proceedings of the 20th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2011). IEEE Press (2011)
12. Bergenti, F., Franchi, E., Poggi, A.: Agent-based interpretations of classic network models. Comput. Math. Organ. Theory **19**(2), 105–127 (2013)
13. Bergenti, F., Gleizes, M.P., Zambonelli, F. (eds.): Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook. Springer, New York (2004). https://doi.org/10.1007/b116049
14. Bergenti, F., Iotti, E., Monica, S., Poggi, A.: A case study of the JADEL programming language. In: Proceedings of the Workshop Dagli Oggetti agli Agenti (WOA 2016). CEUR Workshop Proceedings, vol. 1664, pp. 85–90 (2016)
15. Bergenti, F., Iotti, E., Monica, S., Poggi, A.: A comparison between asynchronous backtracking pseudocode and its JADEL implementation. In: Proceedings of the 9th International Conference on Agents and Artificial Intelligence (ICAART), vol. 2, pp. 250–258. ScitePress (2017)
16. Bergenti, F., Iotti, E., Poggi, A.: Outline of a formalization of JADE multi-agents system. In: Proceedings of the Workshop Dagli Oggetti agli Agenti (WOA 2015). CEUR Workshop Proceedings, vol. 1382, pp. 123–128 (2015)
17. Bergenti, F., Iotti, E., Poggi, A.: Core features of an agent-oriented domain-specific language for JADE agents. In: de la Prieta, F., et al. (eds.) Trends in Practical Applications of Scalable Multi-Agent Systems, the PAAMS Collection. AISC, vol. 473, pp. 213–224. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40159-1_18
18. Bordini, R.H., Braubach, L., Dastani, M., Seghrouchni, A.E.F., Gomez-Sanz, J.J., Leite, J., O'Hare, G., Pokahr, A., Ricci, A.: A survey of programming languages and platforms for multi-agent systems. Informatica **30**(1), 33–44 (2006)
19. Braubach, L., Pokahr, A., Lamersdorf, W.: Jadex: a BDI-agent system combining middleware and reasoning. In: Unland, R., Calisti, M., Klusch, M. (eds.) Software Agent-Based Applications, Platforms and Development Kits, pp. 143–168. Birkhäuser (2005)
20. Challenger, M., Kardas, G., Tekinerdogan, B.: A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems. Software Qual. J. **24**(3), 755–795 (2016)
21. Challenger, M., Mernik, M., Kardas, G., Kosar, T.: Declarative specifications for the development of multi-agent systems. Comput. Stand. Interfaces **43**, 91–115 (2016)
22. Demirkol, S., Challenger, M., Getir, S., Kosar, T., Kardas, G., Mernik, M.: SEA_L: a domain-specific language for Semantic Web enabled multi-agent systems. In: Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 1373–1380 (2012)

23. El Fallah-Seghrouchni, A., Suna, A.: CLAIM: a computational language for autonomous, intelligent and mobile agents. In: Dastani, M.M., Dix, J., El Fallah-Seghrouchni, A. (eds.) ProMAS 2003. LNCS (LNAI), vol. 3067, pp. 90–110. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25936-7_5
24. Fisher, M.: A survey of concurrent MetateM — the language and its applications. In: Gabbay, D.M., Ohlbach, H.J. (eds.) ICTL 1994. LNCS, vol. 827, pp. 480–505. Springer, Heidelberg (1994). https://doi.org/10.1007/BFb0014005
25. Foundation for Intelligent Physical Agents: FIPA specifications, multi-agents system standard specifications (2002). http://www.fipa.org/specifications
26. Haller, P., Odersky, M.: Scala actors: unifying thread-based and event-based programming. Theoret. Comput. Sci. **410**(2), 202–220 (2009)
27. Hindriks, K.V., De Boer, F.S., Van der Hoek, W., Meyer, J.J.C.: Agent programming in 3APL. Auton. Agent. Multi-Agent Syst. **2**(4), 357–401 (1999)
28. Imam, S.M., Sarkar, V.: Habanero-Java library: a Java 8 framework for multicore programming. In: Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ 2014), pp. 75–86. ACM (2014)
29. Imam, S.M., Sarkar, V.: Savina - an actor benchmark suite: enabling empirical evaluation of actor libraries. In: Proceedings of the 4th International Workshop on Programming based on Actors, Agents & Decentralized Control (AGERE!), pp. 67–80. ACM (2014)
30. Kravari, K., Bassiliades, N.: A survey of agent platforms. J. Artif. Soc. Soc. Simul. **18**(1), 11 (2015)
31. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. (CSUR) **37**(4), 316–344 (2005)
32. Monica, S., Bergenti, F.: Location-aware JADE agents in indoor scenarios. In: Proceedings of the Workshop Dagli Oggetti agli Agenti (WOA 2015). CEUR Workshop Proceedings, vol. 1382, pp. 103–108 (2015)
33. Monica, S., Bergenti, F.: A comparison of accurate indoor localization of static targets via WiFi and UWB Ranging. In: de la Prieta, F., et al. (eds.) Trends in Practical Applications of Scalable Multi-Agent Systems, the PAAMS Collection. AISC, vol. 473, pp. 111–123. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40159-1_9
34. Oliveira, N., Pereira, M.J., Henriques, P., Cruz, D.: Domain specific languages: a theoretical survey. In: INFORUM 2009 Simpósio de Informática. Faculdade de Ciências da Universidade de Lisboa (2009)
35. Poggi, A., Bergenti, F.: Developing smart emergency applications with multi-agent systems. Int. J. E-Health Med. Commun. **1**(4), 1–13 (2010)
36. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Van de Velde, W., Perram, J.W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0031845
37. Rodriguez, S., Gaud, N., Galland, S.: SARL: a general-purpose agent-oriented programming language. In: Proceedings of the IEEE/WIC/ACM International Joint Conferences of Web Intelligence (WI) and Intelligent Agent Technologies (IAT), vol. 3, pp. 103–110. IEEE Press (2014)
38. Shoham, Y.: AGENT-0: a simple agent language and its interpreter. In: Proceedings of the 9th National Conference on Artificial Intelligence (AAAI), vol. 91, pp. 704–709 (1991)
39. Shoham, Y.: Agent-oriented programming. Artif. Intell. **60**(1), 51–92 (1993)
40. Shoham, Y.: An overview of agent-oriented programming. In: Bradshaw, J. (ed.) Software Agents, vol. 4, pp. 271–290. MIT Press (1997)

41. Winikoff, M.: JACK intelligent agents: An industrial strength platform. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah, S.A. (eds.) Multi-Agent Programming. MASA, vol. 15, pp. 175–193. Springer, Boston (2005). https://doi.org/10.1007/0-387-26350-0_7
42. Wyatt, D.: Akka concurrency. Artima Incorporation (2013)
43. Yokoo, M., Durfee, E.H., Ishida, T., Kuwabara, K.: The distributed constraint satisfaction problem: formalization and algorithms. IEEE Trans. Knowl. Data Eng. **10**(5), 673–685 (1998)
44. Yokoo, M., Hirayama, K.: Algorithms for distributed constraint satisfaction: a review. Auton. Agent. Multi-Agent Syst. **3**(2), 185–207 (2000)
45. Yokoo, M., Ishida, T., Durfee, E.H., Kuwabara, K.: Distributed constraint satisfaction for formalizing distributed problem solving. In: Proceedings of the 12th International Conference on Distributed Computing Systems, pp. 614–621. IEEE Press (1992)