A Comparison between Asynchronous Backtracking Pseudocode and its JADEL Implementation

Federico Bergenti[1], Eleonora Iotti[2], Stefania Monica[1] and Agostino Poggi[2]
[1]*Dipartimento di Matematica e Informatica, Università degli Studi di Parma, Parma, Italy*
[2]*Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Parma, Parma, Italy*

Keywords: Asynchronous Backtracking, JADEL, Distributed Constraint Satisfaction Problems.

Abstract: In this paper, a comparison between the pseudocode of a well-known algorithm for solving distributed constraint satisfaction problems and the implementation of such an algorithm in JADEL is given. First, background and motivations behind JADEL development are illustrated. Then, we make a description of the problem and a brief introduction to JADEL. The core of this work consists in the translation of the algorithm pseudocode in JADEL code, which is described in details. Scope of the paper is to evaluate such a translation, in terms of closeness to pseudocode, complexity, amount of code written and performance.

1 INTRODUCTION

JADEL (*JADE Language*) is an agent-oriented domain-specific language based on *JADE* (*Java Agent DEvelopment framework*, `jade.tilab.com`). JADE (Bellifemine et al., 2005) is a software framework for implementing agents and multi-agent systems, which are compliant with *FIPA* specifications (Foundation for Intelligent Physical Agents, 2002). It consists of a middleware and it offers several APIs and graphical tools that support multi-agent systems development. JADE can be considered a consolidated tool, and it has been used for many relevant applications, e.g., (Poggi and Bergenti, 2010; Bergenti et al., 2011; Bergenti et al., 2013a; Bergenti et al., 2014). As notable example, JADE has been in daily use for service provision and management in Telecom Italia for more than 6 years, serving millions of customers in one of the largest broadband networks in Europe (Bergenti et al., 2015a). Moreover, JADE allows use of agent technology in various areas, such as agent-based social networks modeling (Bergenti et al., 2013b) and localization (Monica and Bergenti, 2016; Bergenti and Monica, 2016; Monica and Bergenti, 2015). JADE was conceived and developed in the early 2000's, and its main design decisions were based on the available technologies. In particular, one of such technologies is Java, which in those days was a novel and promising technology. Developers wanted to use Java, and the common opinion was that such a technology would have been able to change software implementation processes. Java represented an important step in the fast progress of Web-oriented technologies, and it contributed significantly to the growth of the Web. In such a context, a pure Java approach seemed to be a perfect solution for a software framework that aims at becoming a solid and reliable instrument, and which can be also compatible with most of the other new technologies. Such a choice turned out to be successful, and nowadays JADE is recognized as one of the most popular FIPA compliant agent frameworks (Kravari and Bassiliades, 2015). Nevertheless, our experience in using agent technologies and teaching it to graduate students shows a slow regression of the use of JADE. As a matter of fact, JADE development of multi-agent systems is often perceived as a difficult task, due to two main reasons. First, JADE is constantly expanding and its continuous growth—in terms of features, projects, and available APIs—increases the complexity of the framework. For example, there is a high number of implementation details that a developer must handle, in order to obtain a non-trivial multi-agent system. Second, the language that made success in the early 2000's is now less appealing to multi-agent system developers. In fact, Java does not natively support agent-oriented technologies and methodologies. This is perceived as a limitation and a source of errors. In order to address such problems, we are working on a formal semantics of JADE (Bergenti et al., 2015b).

JADEL project is motivated by the need of sim-

Bergenti F., Iotti E., Monica S. and Poggi A.
A Comparison between Asynchronous Backtracking Pseudocode and its JADEL Implementation.
DOI: 10.5220/0006205902500258
In *Proceedings of the 9th International Conference on Agents and Artificial Intelligence (ICAART 2017)*, pages 250-258
ISBN: 978-989-758-220-2

plification and renovation of JADE users experience. JADEL provides abstractions and constructs which focus on basic agent-oriented features of JADE, and it aims at enforcing the expressiveness of such features and simplifying the construction of multi-agent systems. A first idea of JADEL can be found in (Bergenti, 2014), and more recent developments are discussed in (Bergenti et al., 2016c), where an overview of JADEL syntax and its informal semantics is presented and a first example is used to illustrate the described syntax. Then, in (Bergenti et al., 2016a) and (Bergenti et al., 2016b), the JADEL support at FIPA *Interaction Protocol* is shown. This work shows an usage of JADEL when the pseudocode of an algorithm is given. The scope of such an exercise is to illustrate the steps from pseudocode to implementation, and to analyze the effort spent in doing such a task. 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 paper is organized as follows. First, in Section 2, distributed constraint satisfaction problems are defined and the notation is fixed. Then, in Section 3, JADEL main abstractions and features are presented. Section 4 shows the pseudocode together with the actual implementation, and, finally, Section 5 gives an evaluation of the work. A brief recapitulation of the work and its main results concludes the paper.

2 DISTRIBUTED CONSTRAINT SATISFACTION PROBLEMS

A *Constraint Satisfaction Problem* (*CSP*) is a problem that consists in a finite set of variables and a finite set of constraints over such variables, i.e., predicates defined on variables. As in (Yokoo et al., 1998), we denote variables as $x_1, x_2, \dots, x_n$. Each variable x_i takes values in a domain, called D_i. Constraints are predicates defined on $D_{k1} \times \cdots \times D_{kj}$, indicated by $p(x_{k1}, \dots, x_{kj})$. A constraint is satisfied if the values assigned to its variables make the predicate true. A CSP is *solved* if and only if a value is assigned to each variable, and each assignment satisfies all constraints.

A *Distributed Constraint Satisfaction Problem* (*DCSP*) is a CSP where constraints and variables are distributed among agents. Such agents control a number of variables and they know some predicates. Each agent finds an *assignment* of its variable, i.e., a pair (x_i, d) where $d \in D_i$, that satisfies its known constraints. By interacting among each others, they can obtain the assignments performed by other agents, and check if constraints are still satisfied. Informally, a DCSP is solved if each agent finds a local solution that is coherent with other agents local solutions. More specifically, a DCSP is solved if and only if each agent has assigned a value to all of its variables and all the constraints for all agents are satisfied by such an assignment.

In (Yokoo and Hirayama, 2000), a survey of the main algorithms for solving DCSPs is given. In particular, pseudocode and examples are shown for the asynchronous backtracking, the asynchronous weak-commitment search, the distributed breakout, and the distributed consistency algorithms. We focus on the asynchronous backtracking, called *ABT*. ABT algorithm solves DCSPs that follow 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. We call x_i the agent that owns the variable with the same name. 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 the message. For each agent, the agents who are directly connected with it are called *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 instruments for exchanging knowledge on 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 to 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. Its pseudocode is shown and commented in Section 4, together with its JADEL implementation.

3 A BRIEF OVERVIEW OF JADEL

JADEL provides some main abstractions, namely, *agents*, *behaviours*, *communication ontologies*, and *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 a take-down phase. In the start-up phase the agent performs the instructions given in its `on-create` handler, and in the take-down phase it performs the instructions given in its `on-destroy` handler. Usually, a sequence of tasks is added into the internal list of the agent during its initialization, and such tasks are performed during the execution phase. New tasks can be added dynamically during the agent life cycle, and old tasks can be removed. Note that, as in JADE, agents are single-threaded entities that are provided with an internal scheduler for the management of their tasks. In JADE nomenclature, tasks are called behaviours. Each behaviour describes an action that the agent can perform. In fact, in JADEL, adding tasks to the agent list is done by means of the expression `activate-behaviour`.

JADEL behaviours are associated to a type, which can be `cyclic` or `oneshot`. Cyclic behaviours represent actions that remain in the agent behaviours list after their execution. This means that the action of a cyclic behaviour can be used one or more times during agent life cycle. A one-shot behaviour, instead, contains an action that terminates immediately and is removed from the agent list after one execution. Actions can be triggered by the reception of a message, which is an event, or they can start without waiting for any event. JADEL has a specific construct for handling message reception that provides also a control over the type of message that the agent wants to receive. As a matter of fact, messages can be filtered by using message templates, so the agent can react at the correct message.

Ontologies are used in agents communication, providing a set of propositions, concepts, and predicates. Such terms compose a sort of dictionary, which is usually organized in a hierarchical structure. Agents which share such a dictionary can interact by using common terms as content of their messages. JADEL provides an entity `ontology`, together with a lightweight syntax that aims at simplifying the construction of such dictionaries and their usage.

Finally, JADEL provides an abstraction for managing roles in interaction protocols. Roles are particular behaviours, which are composed of a set of event handlers. Each of such handlers covers a different step of the chosen interaction protocol, by filtering messages through their performatives, as in FIPA specifications. Abstractions and constructs provided by JADEL are a minimal set of entities needed in order to actually run a JADE multi-agent system. JADEL sources are translated into a readable and semantically equivalent Java code that uses JADE APIs. This ensures a tight integration with Java and JADE, and it allows developers to mix JADEL entities with native JADE source code. JADEL agents can be created by using JADE middleware, and they rely on the solid JADE architecture.

4 ABT PSEUDOCODE AND JADEL IMPLEMENTATION

In this Section, the ABT pseudocode is illustrated and compared with the respective JADEL implementation. Pseudocode is taken exactly from (Yokoo and Hirayama, 2000). JADEL syntax and details of ABT implementation are described together with the code.

4.1 Agents

ABT pseudocode describes event handlers and main procedures, but it does not illustrate how agents can 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 *neighbors* set. 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 agent itself. Moreover, `ABTAgent` provides two important methods, namely, `checkConstraints` and `assignVariable`. The former checks if all constraints are satisfied by current assignments in agent view; the latter selects a value which is consistent with agent view and assigns it to the variable owned by the agent. Both methods return a boolean value: `true` if the operation was successful and `false` if it is was not.

4.2 Procedures

As said in Section 2, core 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. The pseudocode of the *check agent view* procedure is shown in Figure 1. In JADEL implementation of ABT, the *check agent view* procedure becomes a one-shot behaviour. In fact, its action has to be performed only once, when the behaviour activates.

```
Procedure check agent view
when agent_view and my_value are inconsistent do
  if no value in $D_i$ is consistent with agent_view
  then
    backtrack
  else
    select $d \in D_i$ where agent_view and d are consistent
    my_value ← d
    send(OK, $(x_i, d)$) to neighbors
  end if
end do
```

Figure 1: Procedure check agent view.

```
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. It denotes an auto-triggering action.

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

The procedure *backtrack* is defined in Figure 2. 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

```
Procedure backtrack
generate a nogood V
when V is an empty nogood do
  broadcast to other agents that there is no solution
  terminate this algorithm
end do
select $(x_j, d_j)$ where $x_j$ has the lowest priority in a nogood
send(nogood, $(x_i, V)$) to $x_j$
remove $(x_j, d_j)$ from agent_view
check agent view
```

Figure 2: Procedure backtrack.

done. JADEL implementation of such a procedure is another one-shot behaviour, whose code follows precisely the pseudocode in Figure 2.

```
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)
    }
  }
}
```

4.3 Event Handlers

Others procedures specified in Yokoo 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, as in Figure 3. The reception of a message requires a cyclic behaviour, which waits

```
when received (OK, $(x_j, d_j)$) do
    revise agent view
    check agent view
end do
```

Figure 3: Reception of an OK message.

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.

```
on message msg
when {
  ontology is ABTOnto 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, as follows: **`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.

```
when received (nogood, $(x_j, V)$) do
    record $V$ as a new constraint
    when $V$ contains an agent $x_k$ that is not its neighbor do
        request $x_k$ to add $x_i$ as a neighbor
        add $x_k$ to its neighbor
    end do
    $old\_value \leftarrow current\_value$
    check agent view
    when $old\_value = current\_value$ do
        send(OK, $(x_i, current\_value)$) to $x_j$
    end do
end do
```

Figure 4: NoGood message reception.

Finally, the pseudocode of the procedure that manages the reception of a NoGood message is shown in Figure 4. In JADEL, such a procedure 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 ABTOnto 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)
}
```

5 EVALUATION

The comparison between 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 domain-specific features of such a code. Finally, we shows an example of usage of ABT algorithm and we measure the time of execution of JADEL, and the number of messages exchanged.

Methods to evaluate domain-specific modeling languages can be found in, e.g., (Challenger et al., 2015), which focuses on multi-agent systems. Other surveys, such as (Mernik et al., 2005) and (Oliveira et al., 2009), highlight the main advantages of domain-specific languages usage. 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 purpose. As far as we know, there are not standard methods for evaluating the closeness of a 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 sections. This choice permits us to make some considerations about the pseudocode.

First, ABT pseudocode is presented by means of procedures and event handlers, with the aid of the keywords `when` and `if`. Second, the notation used inside the ABT pseudocode is the same of the DCSP formalization, shown in Section 2. 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 code. In fact, in the JADEL methodology, 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/event handler, and analyze each of them separately. Calls of procedures in ABT pseudocode translate into the activation of the corresponding behaviour in JADEL. Also sending 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 mappings *theAgent.agentview* and *theAgent.neighbors*, and by defining some ontology terms. As a matter of fact, terms *OK* and *NoGood* are predicate 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. Index and variables are intentionally confused because we identify each agent with its variable. 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*$(\ldots)$ **do**.

In summary, *(i)* ABT procedures are associated with JADEL oneshot behaviours, *(ii)* ABT event handlers are associated with JADEL cyclic behaviours, *(iii)* procedure calls and *send* instructions are associated with the correct behaviour activation, *(iv)* references to agentview or agent neighborhood are associated with the respective JADEL agent fields, *(v)* receptions of messages are associated with JADEL constructs and expressions that concern reception and

Table 1: Metrics for distance between ABT pseudocode and JADEL implementation in terms of number of LOCs, and their complexity in terms of depth. The depth of an equivalent JADE implementation is also shown.

	Distance	Complexity		
	(LOCs)	ABT	JADEL	JADE
Check Agentview	2	2	3	5
Backtrack	6	1	3	6
Receive OK	6	1	1	4
Receive NoGood	7	2	3	4

Table 2: Number of LOC and percentage of Agent-Oriented (AO) features over the total number of LOC, for JADEL and JADE implementation of the ABT example.

	JADEL		
	ABTAgent	ABTOntology	Behaviours
LOCs	57	7	138
AO (%)	12	86	41
	JADE		
	ABTAgent	ABTOntology	Behaviours
LOCs	149	113	380
AO (%)	6	23	20

content extraction from a message. 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.

```
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 5 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 Table 1.

Then, we want to quantify the 'complexity' of the code. In fact, JADEL code sounds similar to ABT pseudocode also because of its structure. We use the depth of each block of code as a measure. As we can see in Table 1, ABT pseudocode and JADEL implementation do not differ significantly in terms of complexity. To this extent, JADEL code often requires one more level (the `do` block), but its structure is usually very similar to ABT pseudocode. The complexity measure makes sense when JADEL code is compared to the equivalent JADE one. Such an equivalent implementation is obtained directly from JADEL compiler, 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 Table 2. In order to emphasize the advantage in using JADEL instead of JADE, the percentage of lines which contains Agent-Oriented (AO) features over the total number of LOCs is also shown. We define as AO features each reference to the agent world. For example, keywords `agent`, `behaviour`, `ontology` are AO features, but also special expressions such as `activate-behaviour`. In JADE, AO features are simply the calls to the API. Table 2 shows that the JADEL implementation is far more lighter than the JADE one, and that it is denser of

Table 3: Average number of messages exchanged, average elapsed time of execution, and average total number of assignments of the n-queens problem, for $n = 4,5,6,7,8$.

# of Queens	Avg. # of Messages	Avg. Elapsed Time (ms)	Avg. # of Assignments
4	20.54	132.05	11.68
5	15.44	113.19	9.40
6	37.00	236.50	17.00
7	63.63	462.25	32.50
8	197.00	1178.50	77.50

AO features. Such measures can be viewed as an indication of simplicity of JADEL code with respect to JADE. Finally, we test JADEL implementation of ABT algorithm on a well-known example of DCSP. As in (Yokoo et al., 1992), we use ABT for solving the n-queens problem. The n-queens problem consists in placing n pieces of chess queens on a $n \times n$ chessboard, so that each queen is safe from others. A solution of such a problem requires, in fact, that a queen does not share a row, a column or a diagonal with any other queen on the chessboard. Formalizing the n-queens problem needs the use of an agent for each queen, and the association of each queen to a row of the chessboard (equivalently, a column). So, the i-th queen, represented by the variable x_i, slides on the $i\text{-}th$ row. This means that the variable x_i takes values into $D_i = \{1,\ldots,n\}$. Hence, the constraints are formalized as follows. For each x_i,x_j, the following inequalities need to hold: $x_i \neq x_j$ and $|x_i - x_j| \neq |i - j|$. The implementation in JADEL of such a problem requires only to extend the `ABTAgent` entity, defining the new constraints into the `checkConstraints` method of the agent, as follows.

```
boolean checkConstraints() {
  if (!checkNoGoodConstraint()) {
    return false
  }

  for(i : 0 ..< myIndex) {
    var key = sortedAgents.get(i)
    if (agentview.containsKey(key)) {
      val value = agentview.get(key)
      if (myValue == value || (Math.
          abs(myValue - value) == Math.
          abs(myIndex - i))) {
        return false
      }
    }
  }
  return true
}
```

We measure the effectiveness of the algorithm in terms of number of messages and time. Distributed systems are evaluated by means of their usage of memory, the time of execution, and the amount of messages exchanged, but we can only measure those last two parameters, due to the garbage collector of Java, that falsify the actual memory usage. Then, an evaluation of the effectiveness of ABT implementation is given in terms of the number of assignments that each agent performs during the computation. Results are shown in Table 3.

6 CONCLUSIONS

In this paper, a comparison between the implementation of ABT algorithm with the novel agent-oriented language JADEL and the ABT pseudocode is shown. First, JADEL and JADE are briefly presented, then DCSP problems and ABT algorithm notations are defined and described. Given the official presentation of ABT pseudocode, we made a point-to-point translation of such a pseudocode in JADEL. Finally, some metrics are defined and some measurements are made in order to evaluate the effectiveness of the language in such a context, and its closeness to ABT pseudocode. The proposed metrics are *(i)* a measure of distance, in terms of LOCs, between ABT pseudocode and its JADEL implementation, *(ii)* a measure of complexity, based on the depth of the block of codes, *(iii)* the number of LOCs and the percentage of AO features of JADEL code and JADE equivalent, and *(iv)* the average number of messages and time of execution of JADEL version of ABT, when used for solving the n-queens problem, for $n = 4,5,6,7,8$.

Not all of such metrics can be interpreted as complete or significant in every situation, because they cannot fully describe qualitative factors, such as readability, re-usability or maintainability. In fact, distance and complexity can be different depending on the type of pseudocode given, and there is not a standard way for evaluating pseudocodes due to their inherent informality. Also, numbers of LOCs and percentage of AO features do not refer to the quality of the code. However, such measurements help us in evaluating simplicity and giving an idea of the expressiveness and effectiveness of the language. As a matter of fact, JADEL distance from pseudocode is very small, and we argue that this fact may help developers in translating an idea of distributed algorithms into a working JADE multi-agent system. When coding it directly in JADE, in fact, the number of LOCs required increases considerably, making its distance from pseudocode very high. 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. In summary, JADE obtains good results in simplifying and shortening the task of writing code. Then, as a last evaluation, the metrics for distributed systems are used, showing good performance of the language. When using JADE instead of JADEL, the number of messages is about the same, while the performance is slightly better, due to the redundancy and overhead introduced by JADEL compiler in generating Java code.

As a future development of this work, JADEL can be tested on other algorithms, making other comparison between pseudocodes and JADEL code, in order to gain a more complete view. Moreover, other famous agent-oriented programming languages, such as 3APL, Jason, SARL, can be compared with JADEL, in terms of translating a given pseudocode. Finally, the best evaluation could be that of JADE developers, when JADEL will be released. In summary, this paper takes place into a larger project of presentation of JADEL and evaluation of its possibilities. This work shows how to produce JADEL code from a pseudocode and measures how simple or complex this task can be.

REFERENCES

Bellifemine, F., Bergenti, F., Caire, G., and Poggi, A. (2005). JADE – A Java Agent DEvelopment framework. In *Multi-Agent Programming*. Springer.

Bergenti, F. (2014). An introduction to the JADEL programming language. In *Procs. IEEE 26th Int'l Conf. on Tools with Artificial Intelligence (ICTAI)*. IEEE Press.

Bergenti, F., Caire, G., and Gotta, D. (2013a). An overview of the AMUSE social gaming platform. In *Procs. Workshop Dagli Oggetti agli Agenti (WOA 2013)*, volume 1099 of *CEUR Workshop Proceedings*.

Bergenti, F., Caire, G., and Gotta, D. (2014). Agents on the move: JADE for Android devices. In *Procs. Workshop Dagli Oggetti Agli Agenti (WOA 2014)*, volume 1260 of *CEUR Workshop Proceedings*.

Bergenti, F., Caire, G., and Gotta, D. (2015a). Large-scale network and service management with WANTS. In *Industrial Agents: Emerging Applications of Software Agents in Industry*. Elsevier.

Bergenti, F., Franchi, E., and Poggi, A. (2011). Agent-based social networks for enterprise collaboration. In *Procs. 20th Int'l Conf. Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2011)*. IEEE Press.

Bergenti, F., Franchi, E., and Poggi, A. (2013b). Agent-based interpretations of classic network models. *Computational and Mathematical Organization Theory*, 19(2).

Bergenti, F., Iotti, E., Monica, S., and Poggi, A. (2016a). A case study of the JADEL programming language. In *Procs. Workshop Dagli Oggetti agli Agenti (WOA 2016)*, volume 1664 of *CEUR Workshop Proceedings*.

Bergenti, F., Iotti, E., Monica, S., and Poggi, A. (2016b). Interaction protocols in the JADEL programming language. In *Procs. 6th Int'l Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*.

Bergenti, F., Iotti, E., and Poggi, A. (2015b). Outline of a formalization of JADE multi-agents system. In *Procs. Workshop Dagli Oggetti agli Agenti (WOA 2015)*, volume 1382 of *CEUR Workshop Proceedings*.

Bergenti, F., Iotti, E., and Poggi, A. (2016c). Core features of an agent-oriented domain-specific language for JADE agents. In *Trends in Practical Applications of Scalable Multi-Agent Systems, the PAAMS Collection*. Springer.

Bergenti, F. and Monica, S. (2016). Location-Aware Social Gaming with AMUSE. In *Trends in Practical Applications of Scalable Multi-Agent Systems, the PAAMS Collection (PAAMS 2016)*.

Challenger, M., Kardas, G., and Tekinerdogan, B. (2015). A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems. *Software Quality Journal*.

Foundation for Intelligent Physical Agents (2002). FIPA specifications. Multi-agents system standard specifications. `http://www.fipa.org/specifications`.

Kravari, K. and Bassiliades, N. (2015). A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, 18(1).

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4).

Monica, S. and Bergenti, F. (2015). Location-aware JADE agents in indoor scenarios. In *Procs. Workshop Dagli Oggetti agli Agenti (WOA 2015)*, volume 1382 of *CEUR Workshop Proceedings*.

Monica, S. and Bergenti, F. (2016). A comparison of accurate indoor localization of static targets via WiFi and UWB ranging. In *Advances in Intelligent Systems and Computing (PAAMS 2016), Special Session on Agents and Mobile Devices (AM)*.

Oliveira, N., Pereira, M. J., Henriques, P., and Cruz, D. (2009). Domain specific languages: A theoretical survey. In *INFORUM'09 Simpósio de Informática*. Faculdade de Ciências da Universidade de Lisboa.

Poggi, A. and Bergenti, F. (2010). Developing smart emergency applications with multi-agent systems. *Int. J. E-Health Med. Commun.*, 1(4).

Yokoo, M., Durfee, E. H., Ishida, T., and Kuwabara, K. (1998). The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5).

Yokoo, M. and Hirayama, K. (2000). Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2).

Yokoo, M., Ishida, T., Durfee, E. H., and Kuwabara, K. (1992). Distributed constraint satisfaction for formalizing distributed problem solving. In *Procs. 12th Int'l Conf. Distributed Computing Systems*. IEEE Press.