ERLANG AND SCALA FOR AGENT DEVELOPMENT

Dejan Mitrović, Mirjana Ivanović, Zoran Budimac

Department of Mathematics and Informatics
Faculty of Sciences, University of Novi Sad
Novi Sad, Serbia
{dejan, mira, zjb}@dmi.uns.ac.rs

Abstract

The actor-based concurrency model and the agent-oriented programming paradigm share many of their core features. When compared to the classical multi-thread model, actor-based software agents bring many benefits to agent developers and the agent technology itself. The benefits include lower run-time resource requirements, as well as faster development due to the existing communication infrastructure. This paper provides an evaluation of two popular programming languages – Erlang and Scala – in the context of actor-based software agents operating in a distributed environment. The presented evaluation considers a number of criteria, including the simplicity of writing and using agent-based actors, transparency of the distributed and remote communication, and the run-time performance under heavy loads. This kind of a thorough analysis and evaluation is important before a long-term commitment to any of the two languages for developing a future multi-agent system.

Keywords - Actors, software agents, distributed programming, platform evaluation.

1 INTRODUCTION

Classical multi-threaded programming model is based on the concept of multiple threads of execution sharing common resources, such as code and memory. This model is directly supported by the majority of modern operating systems, and enables fast and efficient inter-thread communication. However, it also requires a well-designed, *synchronized* access to the shared resource pool. And although modern programming languages and platforms offer advanced synchronization tools (e.g. locks and semaphores), proper design and implementation of a correct multi-threaded behavior is often a difficult task, especially in large software systems.

As an alternative to the classical multi-threaded programming, the *actor model* was first proposed in [10]. Unlike threads, actors utilize a *share-nothing* approach. Each actor is a self-contained entity, and relies on the exchange of messages to communicate with its environment, as well as with other actors. Due to the isolated nature of actors, common synchronization pitfalls, such as *deadlocks*, can easily be avoided. This leads to an easier development and maintenance of large concurrent software system, with a possible loss of performance due to the more complex communication patterns.

Several programming languages offer the "native" support for actors, i.e. at the language level. Out of these, the two that have been accepted by the industry are Erlang [5] and Scala [15]. Erlang combines declarative and functional programming paradigms. It is primarily aimed at developing concurrent, distributed and fault-tolerant software systems. Scala, on the other hand, represents a mix of imperative (object-oriented) and functional programming paradigms. Programs written in Scala are compiled into Java byte code, allowing developers to exploit the large pool of existing Java libraries and frameworks.

The actor model fits perfectly into the *agent-oriented* programming paradigm. *Software agents*, like actors, are self-contained executable entities which communicate via the message exchange. Agents might have additional properties, such as autonomous, intelligent, and goal-directed behavior [20], but the core approach of developing both actors and agents is the same.

When it comes to developing agents and multi-agent systems, Java is currently the predominant implementation platform. The usual design approach is to assign a separate Java thread to each individual agent. However, in terms of the required resources, threads are expensive: while the Java virtual machine (JVM) might host millions of regular objects, it can run only a few thousands of threads

(depending on the underlying hardware). The idea is, therefore, to map agents to actors, and, by doing so, to reduce the resource requirements. However, besides the lower resource requirements, the actors-based agent development has additional important benefits. For example, there is no need to design and implement the communication architecture. This can save significant development time and resources.

This paper analyzes the actor-based programming model in both Erlang and Scala, in the context of agent development. The main goal is to determine which of the two languages represents a *better* tool for developing software agents and multi-agent systems, when considering the following factors:

- Simplicity of the included actor implementation: how easy it is to write and use agents that are based on actors;
- Support for distributed communication: agents often need to communicate with remote, physically distributed agents; and
- Run-time efficiency: how well the Erlang and Scala platforms scale-up in terms of the number of running actors/agents, as well as the number of exchanged messages.

This analysis and evaluation are important before a long-term commitment to any of the two languages for developing future multi-agent platforms.

The rest of the paper is organized as follows. Section 2 provides an overview of the existing work related to actor and agent development in Erlang and Scala. The comparison of the actor-based programming model provided by the two languages and their underlying platforms is given in Section 3. Run-time efficiency of the platforms is evaluated in Section 4. Finally, general conclusions are given in Section 6.

2 RELATED WORK

Erlang's actor implementation has had a major influence on other programming languages, both agent-oriented and general-purpose. For example, purely agent-oriented programming languages *April* and *Go!* include the communication infrastructure inspired by Erlang [4]. Scala actor library implementation is also known to be influenced by Erlang [13].

Many researchers have recognized Erlang as a powerful tool for developing multi-agent systems and software agents, or any kind of *autonomous* systems (e.g. autonomous mobile robots [14]). The *European Coordination Action for Agent-Based Computing* considers Erlang to be "agent software" [1]. As concluded in [19], multi-agent systems based on Erlang can provide advanced features, such as dynamic reconfiguration and fault-tolerance, with the minimum amount of effort. This is because Erlang itself is designed with these features in mind. For example, Erlang actors can be organized into a *supervision tree*. If a supervised actor fails, the supervisor can restart it. This kind of fault-tolerant behavior is desired in many modern multi-agent systems (see e.g. [7]).

eXAT is a fully-featured Erlang-based multi-agent system that can host reactive, intelligent, and social agents [17][18]. Behavior of an *eXAT* agent is modeled as a finite-state machine. To create agents with complex behavior and to allow for code re-use, the system supports finite-state machine composition and inheritance. Agent intelligence is enabled via the included reasoning system *ERESYE* [16]. Finally, *eXAT* includes an *ontology compiler*, allowing developers to define more meaningful agent communication.

Scala, with its actor library and byte-code compatibility with Java, represents an excellent framework for developing modern multi-agent systems. However, since the language is relatively recent (the first version appeared in early 2000s), there appears to be no existing Scala-based multi-agent system. Several prototypes, however, have been proposed. One such prototype is *Actorsim* [11]. Its early evaluation results show that actors perform better for large number of agents. For a relatively low number of agents, classical threads still represent a better solution. As discussed earlier, in the introductory section, these results are expected, since the inter-actor communication if generally slower than the shared-memory approach used by threads.

Unlike Erlang, Scala can be very efficiently used to develop new agent-oriented programming languages (*AOPLs*). An *AOPL* incorporates programming constructs that hide the overall complexity of the agent technology, and simplify the agent-development process. Scala's rich and flexible syntax and its *meta-programming* abilities allow for an easy introduction of new programming constructs into the language, suitable for developing both reactive and intelligent agents [12].

3 ACTORS IN ERLANG AND SCALA

Both Erlang and Scala provide the actor support at the language level. Moreover, actors in Scala are heavily inspired by Erlang [13], so there are some similarities in the way actors are developed and used. This section describes the actor model available in both languages, highlighting the similarities as well as differences. Simple code examples are given as well. A more complex case-study and the run-time efficiency evaluation are presented in Section 4.

3.1 Erlang

Erlang was first developed in the late 1980s at the *Ericsson Computer Science Laboratory* with the goal of enabling straightforward development of concurrent, distributed, and fault-tolerant software systems [3] [5]. The language is a mix of declarative and functional programming paradigms, featuring single-assignment (similar to Prolog) and dynamic typing. Its code is executed in a virtual machine, the officially supported one being *Bogdan/Bjrn's Erlang Abstract Machine* (*BEAM*). Erlang is accompanied by the *Open Telecom Platform* (*OTP*), a collection of libraries and design principles.

In Erlang, actors are referred to as *processes*. They are at the core of the Erlang's concurrent programming model, in the sense that the platform does not provide any means for writing shared-memory threads. The most important programming constructs for working with processes include [2]:

- Primitive *spawn*: creates a new parallel process;
- Infix operator "!": sends a message to an existing process;
- Pattern matching operation *receive*: used by processes to receive messages; and
- Primitive *register*: binds a process to a globally-available name.

Message passing is performed in an asynchronous manner – the "!" operator is executed immediately, without blocking the sender. There are no built-in means for the sender to determine if the message was successfully received by the target process (although an error will be raised if the receiver is not known in the process table). The Erlang's platform, however, guarantees the order of messages: if a sequence of messages is sent, it is guaranteed that they will be received in the same order in which they were sent.

The support for distributed applications is at the Erlang's core. From the developer's point of view, there are almost no distinctions between writing a single-machine and a distributed application. A process communicates with a physically distributed process in almost the same manner as if it would with a process running on the same machine. Additionally, a process can easily create new processes on a distributed machine, running on a different hardware and/or software platform.

A simple example demonstrating the described concepts is shown in Listing 1. The example includes two processes: *ping*, which acts as the initial message sender, and *pong*, which receives messages in a loop and replies to the original sender with the same message content.

Listing 1: Example of an inter-process communication in Erlang

```
-module(example).
-export([ run/0, ping/0, pong/0 ]).
pong() -> receive
  { Sender, Content } -> % accept only messages matching this pattern
    io:format("Received ~s~n", [ Content ]),
    Sender ! Content % reply to the sender including the same content
```

```
end, pong(). % loop
ping() -> pong_process ! { self(), "Hello!" }. % send a message pong, including my PID
run() ->
  % start the pong process and register it under the name of 'pong_process'
  Pid = spawn(example, pong, []),
  register(pong_process, Pid),
  spawn(example, ping, []). % start the ping process
```

In order for this example to work in a distributed environment – that is, to have *ping* and *pong* processes running on different machines – it is enough to update the message sending in the process *ping*. The left-hand side of the operator "!" needs to include the name of the Erlang *node* running the *pong* process, in the form of *{NodeName, pong_process}*. The node name does not have to be fixed, and can be passed to the process as an input argument. The code for the *pong* process does not need to be changed. That is, the process identifier obtained by invoking *self()* can be used in a distributed setting as well.

The next sub-section describes the Scala approach to developing actor-based applications, and outlines similarities and differences with Erlang.

3.2 Scala

Scala is a relatively new programming language, developed in early 2000s. It is a statically typed language, with a concise syntax (when compared to Java), incorporating both object-oriented and functional programming paradigms. One of its most important features is byte-code compatibility with Java. That is, Scala compiler produces Java byte-code that is executed in JVM. This also allows Scala developers to use the existing, large pool of Java libraries and frameworks.

Scala supports both the traditional thread-based and the actors-based approach to developing concurrent software systems. The thread-based approach relies on Java concurrency *API*, while the actor library is heavily influenced by Erlang [13]. Therefore, as in Erlang, the message sender in Scala uses the "!" operator to asynchronously dispatch messages, while the receiver relies on pattern matching to handle different message types. Several additional communication-related functions are introduced in Scala [8]. These include (but are not limited to) functions "!?" for sending a message in a synchronous (i.e. blocking) fashion, *reply* to easily reply to the original sender, and *forward* to forward the message to a third actor, keeping the original sender. Although useful in certain scenarios, the "!?" function can lead to *deadlocks* if the actors' behavior is not designed properly.

Unlike in Erlang, a Scala actor can both *receive* a message, and *react* to it [8]. The programming construct *receive* is used to develop thread-based actors. In this approach, a separate thread is allocated for each individual actor. The thread is blocked while its actor waits for an incoming message. Obviously, thread-based Scala actors are *heavyweight*, in terms of required resources.

The programming construct *react*, on the other hand, is used to develop event-based actors. An event-based actor is registered with the actor runtime as an event recipient. Once a message is received, the actor is assigned a thread, and then executed. Event-based actors in Scala are thus *lightweight*, in the sense that (at the far extreme) a single thread can be recycled to handle multiple actors. As an important optimization step, message-handling code inside *react* always ends abruptly, with an exception [8]. The benefit of this approach is that the receiver's call stack can (and is) discarded, making the thread initialization and recycling process faster. There are also some disadvantages; for example, any code following the message handlers will not be executed.

Listing 2 outlines the same example of communicating actors described earlier, now written in Scala. The *pong* actor is event-based, and, because of the limitations of *react*, relies on the programming construct *loop* to receive multiple messages.

Listing 2: Example of an inter-actor communication in Scala

```
class Pong extends Actor {
```

```
  def act() { // defines the pong's behavior
    loop { // 'react' can only be looped inside 'loop'
      react {
        case str: String => // accept only String-typed messages
          println("Received " + str)
          reply(str) // reply to the sender with the original content
      }
} } }
object Main {
  def main(args: Array[String]) {
    // start the pong actor
    val pong = new Pong()
    pong.start
    // defining the ping actor without a separate class
    actor { pong ! "Hello!" } } }
```

In order to exchange more complex messages in Scala, it is common to define the message structure as a *case class*. Case classes are special types of classes that can be used in pattern matching. However, a special care needs to be taken when exchanging custom messages. Although the actor model defines the share-nothing approach, there is an important optimization step in Scala: if the two actors are hosted by the same JVM, the receiving actor will be given a reference to the sender's object. That is, instead of cloning the sender's object, the two actors will share a reference to the same object. To avoid any data inconsistencies, it is highly advisable for actors to communicate using immutable objects only.

Using actors in a distributed environment is relatively simple in Scala, although not as straightforward as in Erlang. Each receiving actor needs to register itself using a globally-unique name, under a certain *TCP* port. A sender then establishes a *TCP* connection to the remote machine and performs a lookup of the receiver. Additional steps, such as defining the class loader, need to be performed as well. To ease the development for distributed environments, the Scala library provides a dedicated remote library at a higher-level of abstraction, hiding the *TCP* communication details [9].

The next section evaluates the run-time efficiency of the languages' platforms, focusing on a more complex, agent-related execution scenario.

4 RUN-TIME EFFICIENCY

Analysis of the language and library support for actors in Erlang and Scala presented in the previous section leads to the conclusion that Erlang is somewhat easier to use, especially in a distributed setting. This section evaluates how well the languages' underlying platforms (i.e. *BEAM* and *JVM*) scale-up, in order to host large numbers of actors/agents that operate in a distributed environment, and exchange a large amount of long messages. Because the final goal is to determine which of the platforms would be better for developing software agents and multi-agent systems, the evaluation relies on a standardized agent communication pattern.

4.1 Contract Net

The *Contract Net* protocol is a well-known agent interaction protocol, standardized by the *Foundation for Intelligent Physical Agents* (*FIPA*) [6]. The protocol is used when a *manager* agent wishes to hire one or more *contractor* agents to perform a task. The manager's goal is to hire a contractor that can perform the task in an *optimum* way, e.g. at the lowest price, shortest completion time, etc. The manager first advertises a *call for proposals*, describing the task, as well as any restrictions. Potential contractors respond to the call, either by making a proposal, or refusing to participate. The manager analyzes received proposals, trying to select the optimum one. Then, it informs each potential

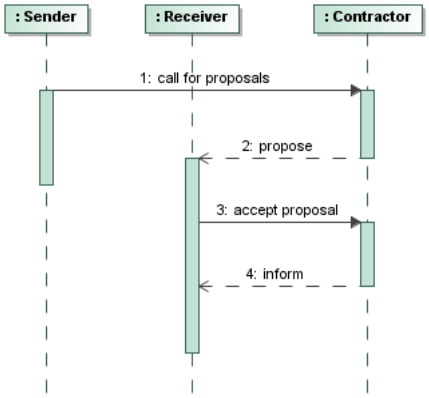

Fig. 1: Flow of messages in the Contract Net implementation used for experiments

contractor whether their proposal is accepted or rejected. The selected contractors perform the task, and finally submit either the result or a failure notification to the manager.

As the standardized and well-known interaction protocol, *Contract Net* was used to experimentally evaluate run-time efficiency of Erlang's and Scala's platforms in a distributed environment. In both Erlang and Scala implementations, the role of manager is actually played by two separate actors – *sender* and *receiver* – for, respectively, advertising the call and accepting proposals. The two separate actors are used because potential contractors might start sending their respective proposals before the *sender* actor has finished advertising the call. The task description included in the call is a fixed-size list of random bytes. Each potential contractor "analyzes" or "performs" the task (after receiving the call for proposals or acceptance message, respectively) by summing the bytes, and responds with the sum. The sum is used as a simple message integrity check.

The general flow of messages between the manager (that is, sender and receiver) and a single contractor is outlined by the sequence diagram shown in Fig. 1. Messages 1 and 3 include a fixed-size list of random bytes, representing the task description. Upon receiving one of these messages, and before sending a reply, the contractor analyzes or performs the task as described earlier.

5 EVALUATION RESULTS

The set of experiments was performed in a heterogeneous distributed environment. The manager was hosted by a lower-end computer, running 32-bit version of *Xubuntu Linux 12.04* on a dual-core processor at 1.6 *GHz* with 2 *GB* of *RAM*. All contractors were hosted on another higher-end computer, running 32-bit version of *Microsoft Windows 7 Professional* on a quad-core processor at 3.6 *GHz* with 4 *GB* of *RAM*. The two computers were connected by a high-speed *LAN*.

Two series of experiments were performed. In the first series, the message content size was fixed to 64 *KB*, and the number of contractors was then gradually increased. In this way, the evaluation results demonstrate how well Erlang and Scala platforms scale-up as the number of agents exchanging relatively large messages increases. The Scala implementation of contractors was based on lightweight, event-based actors.

Fig. 2 shows the results of the first experiment. As it can be concluded, the Scala implementation does not scale-up well. Its average per-message delivery times are significantly higher than the respective values in the Erlang implementation. In the extreme case, when there are 100,000 contractors, the Scala implementation completes the full *Contract Net* protocol cycle in 96 minutes, while the Erlang implementation takes less than 3 minutes. Additionally, the increase in time in the Scala implementation is linear. The cause of this inefficiency in Scala lies in the library for remote actor communication. At the time of performing these experiments, the latest version of Scala was 2.9.2. Unfortunately, its library for remote actor communication still uses the old *blocking* I/O, instead of the *asynchronous NIO API*. Therefore, when remote communication is required, developers can only rely on serial, instead of parallel exchange of messages. On the other hand, Erlang and its underlying *BEAM* platform scale-up very well. The difference in average per-message time between 100 and 100,000 contractors is practically insignificant (0.42 and 1.39 seconds, respectively).

In the second series of experiment, the number of contractors was fixed to 1000. The message content size was then gradually increased from 8 *KB* to 1 *MB*. The goal was to determine how the message size affects the communication infrastructure, and, to a certain extent, to evaluate the computational efficiency of both platforms. Upon receiving both the call for proposals and acceptance messages, each contractor had to calculate the sum of the byte list representing the message content. None of the existing Erlang/Scala functions was used to calculate the sum; instead, the Erlang implementation used a tail-recursive function, while in Scala a plain *for*-loop was implemented.

Results of the second series of experiments were very similar to those presented in Fig. 2. Due to the blocking nature of the remote communication, the increase in time in Scala was linear, while in Erlang, the time difference was practically insignificant. That is, the Erlang implementation was able to very efficiently handle even the scenario of running 1000 actors/agents on a single computer, and remotely exchanging 4000 messages, 2000 of which were carrying a payload of 1 *MB* each.

The results of these experiments, besides providing an insight into the efficiency of each platform, confirm an important premise given in the introductory section – an actor-based agent implementation is more resource-efficient than an implementation based on classic threads. For example, the thread-based implementation would not be able to run such high numbers of contractors on a single machine.

6 CONCLUSIONS

The purpose of this paper was to determine which of the two languages – Erlang or Scala – are *better*

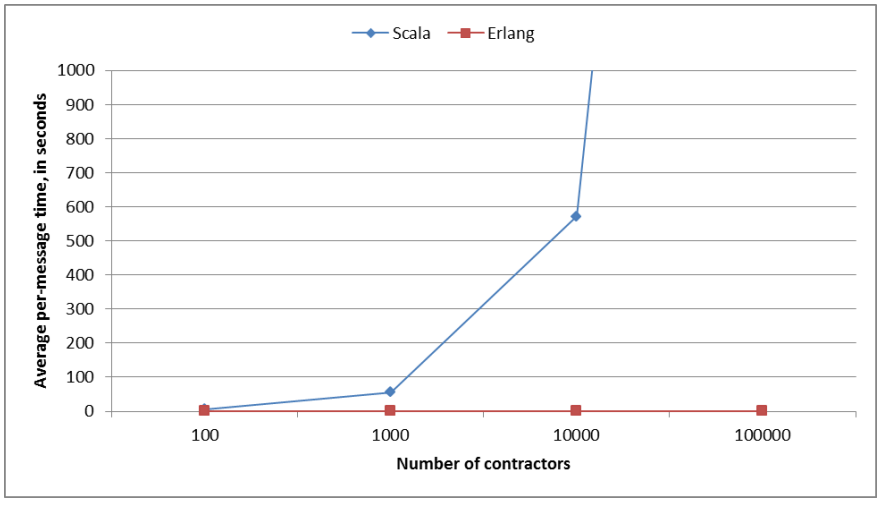

Fig. 2: Run-time efficiency of the Contract Net protocol implementation in Erlang and Scala

for developing actor-based agents and multi-agent systems. The evaluation was based on several criteria: the simplicity of writing and using actor-based agents, the support for distributed environments and remote actor communication, and the run-time efficiency. For the last criterion, a set of experiments was executed in order to determine how well the languages' platforms scale-up when faced with large numbers of actors, exchanging relatively large messages.

Based on the thorough evaluation presented in the paper, the general conclusions are as follows. Both languages make it relatively simple to write and use actor-based agents. Erlang might provide a smaller set of actor-related functions, but the set is sufficient for writing even large software systems. In a distributed environment, the approach offered by Erlang is truly transparent. Agent developers in Erlang do not have to think in advance whether their agents will operate on a single machine, or in a network, and can focus on solving the problem in question.

Scala support for distributed actors and remote communication is not much more complicated. It requires only a few additional steps, but the actor does have to be designed for either remote or local communication. However, a serious issue lies in its run-time performance. The remote library implementation in Scala performs on the orders of magnitude worse than that in Erlang. The core issue is the blocking I/O *API* used by the remote Scala library, which can provide only serial, instead of parallel message exchange.

When compared to Erlang, Scala might offer some benefits. For example, its object-oriented nature makes it easier to design and implement large software systems. Byte-code compatibility with Java enables Scala to re-use the extensive pool of existing Java libraries and frameworks. The Prolog-like syntax of Erlang, its single variable assignment policy, and a heavy dependency on recursion might appear unappealing to mainstream developers. However, based on the evaluation presented in this paper, the overall conclusion is that for actor-based multi-agent systems requiring distributed, remote communication, Erlang represents a better implementation platform than Scala – both because of its ease of use, and run-time performance.

Acknowledgements

The work is partially supported by Ministry of Education, Science and Technological Development of the Republic of Serbia, through project no. OI174023: "Intelligent techniques and their integration into wide-spectrum decision support".

References

[1] AgentLink, European Co-Ordination Action for Agent-Based Computing, http://eprints.agentlink.org/view/type/software.html, retrieved on December 22, 2012.

[2] Armstrong, J.: Erlang – a survey of the language and its industrial applications. In *Proceedings of the symposium on industrial applications of Prolog (INAP96)*, Hino, Tokyo, Japan, October 16-18, 1996.

[3] Armstrong, J.: The development of Erlang. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pp. 196-203, ACM Press, 1997.

[4] Clark, K. L., McCabe, F. G.: Go! for multi-threaded deliberative agents. In *Lecture Notes in Artificial Intelligence*, vol. 2990, pp. 54-75, 2003.

[5] Erlang Homepage, http://www.erlang.org.

[6] FIPA Contract Net Interaction Protocol Specification, http://www.fipa.org/specs/fipa00029/SC00029H.pdf, 2002.

[7] Geraci, S., Giacalone, L., Leone, C., Mangano, S., Pitarresi, G., Scaglione, A., Sorce, S., Genco, A.: Fault tolerance. In *Mobile agents: principles of operation and applications*, edited by Genco, A., pp. 139-179, 2008.

[8] Haller, P., Odersky, M.: Scala actors: unifying thread-based and event-based programming. In *Theoretical Computer Science*, Vol. 410, Issues 2-3, pp. 202-220, 2009.

[9] Haller, P., Sommers, F.: Actors in Scala. *Artima Press, Walnut Creek, California, USA*, 2011.

[10] Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on artificial intelligence (IJCAI'73)*, pp. 235-245, 1973.

[11] Keller, A., Kelly, M., Todd, A.: CASEsim usability and an actor-based framework for multi-agent system simulations, 2010.

[12] Mitrovic, D., Ivanovic, M., Budimac, Z.: Towards an agent-oriented programming language based on Scala. In *Proceedings of Symposium on Computer Languages, Implementations, and Tools (SCLIT2012) held within International Conference on Numerical Analysis and Applied Mathematics (ICNAAM2012)* (ed. Theodore E. Simos), AIP Conf. Proc. 1479, pp. 478-481, 2012.

[13] Odersky, M., Spoon, L., Venners, B.: Programming Scala, Second Edition. *Artima Press, Walnut Creek, California, USA*, 2010.

[14] Santoro, C.: An Erlang Framework for Autonomous Mobile Robots. In *Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop (ERLANG'07)*, pp. 85-92, 2007.

[15] Scala Homepage, http://www.scala-lang.org.

[16] Stefano, A. D., Gangemi, F., Santoro, C.: ERESYE: Artificial Intelligence in Erlang Programs. In *Erlang Workshop at 2005 International ACM Conference on Functional Programming (ICFP 2005)*, Tallinn, Estonia, September 25, 2005.

[17] Stefano, A. D., Santoro, C.: Designing Collaborative Agents with eXAT. In *IEEE International Workshops on Enabling Technologies*, pp. 15-20, 2004.

[18] Stefano, A. D., Santoro, C.: On the use of Erlang as a Promising Language to Develop Agent Systems. In *AI*IA/TABOO Joint Workshop on Objects and Agents (WOA 2004)*, Torino, Italy, November 29-30, 2004.

[19] Varela, C., Abalde, C., Castro, L., Gulias, J.: On modelling agent systems with Erlang. In *Proceedings of the 2004 ACM SIGPLAN workshop on Erlang (ERLANG'04)*, pp. 65-70, 2004.

[20] Wooldridge, M., Jennings, N.: Agent theories, architectures, and languages: a survey. In *Lecture Notes in Computer Science*, Vol. 890, pp. 1-39, 1995.