A Comparison of the Basic Principles and Behavioural Aspects of Akka, JaCaMo and Jade Development Frameworks

Massimo Cossentino, Salvatore Lopes, Angelo Nuzzo, Giovanni Renda, Luca Sabatucci
National Research Council
ICAR Institute, Palermo, Italy
{name.surname}@icar.cnr.it

***Abstract*—Akka, JaCaMo, and Jade are three Java-based frameworks for agent/actor system programming. They present substantial differences both in the reference models and the behavioural aspects of the main entities (actors vs agents). The objective of this work is to compare the basic principles and behavioural aspects of these three frameworks, also giving an overview of other comparison categories in which we briefly discuss other criteria like reasoning and knowledge, interaction/communication model, sociality. In each sub-category, the characteristics of the three frameworks will be analysed, and finally, the relative differences will be discussed. The analysis highlights a substantial difference between Akka actor-based system and agent-based ones, such as JaCaMo and Jade. The results of the analysis reveal that each framework has some competitive advantages over the others. In particular, the orientation to the reasoning and the pro-activity of the agents, the presence of native tools for communication and ontology and the predisposition to the widespread deployment of the code require a careful analysis of the software requirements for the choice of the most suitable framework.**

I. Introduction

The computational characteristics of modern processors and the nowadays available cloud network infrastructures have paved the way for the development of languages and platforms that respond to the need of creating concurrent e distributed programming applications. This work will specifically discuss two models that have developed within this road: actors and agents. In particular, the focus will be on their foundational principles and their behaviour models. Although the two concepts come from different studies and have developed in different periods, both share some principles and functionalities and are useful for responding to common application needs. Both have been used to create many applications, some for research other for industrial/commercial uses, which are very different from each other. Furthermore, each development framework offers functionality whose usefulness must be evaluated by the specific needs and requirements of the application to be developed. The present work aims to partially fill the gap in the literature by providing a comparative analysis of actor/agent systems. In particular, a "panoramic" comparison of these three systems has been not published yet, while a lot of articles focus on more specific aspects. Among the projects developed on the base of the two models, we choose to analyse those that combine the needs of concurrent and distributed programming with a Java-based development environment, namely: Akka, JaCaMo, Jade. Their comparison has been carried out through the consideration of two main categories, to which eight sub-categories are associated. Due to space concerns, for this paper, we limit ourselves to briefly present four further comparison categories: nevertheless, the relative analysis will be summarily outlined to give the reader a wider scenario. The information reported for each platform is the result of a comparison between the available scientific literature and, where present, the official development documentation of each framework. In the second section, the three platforms will be presented, highlighting the main characteristics, fields of application and reference models. Subsequently, in the third section, we will proceed to describe the six main benchmark's categories, discussing why each category has been selected and the criteria through which the three platforms will be evaluated. In the fourth section, we will discuss the comparison results for each platform, focusing only on the first two categories, to highlight some final considerations at the end of each category. In the fifth section, an overview of the other categories will be presented. Finally, some conclusions will be reported.

II. Platforms/Frameworks Overview

In the next paragraphs, we will propose a general description of the three platforms analysed in this paper. Although there are several platforms that respond to the requirements needed to develop concurrent and distributed applications (among these we can also cite FIPA-OS, Jack, 2APL, PROFETA [1]) we choose to analyse these three platforms by their common Java-based environment and their diffusion. The description of the three platforms will also present the main Industrial-strength applications as suggested by Mascardi et al. [2].

A. AKKA

Akka is a middleware for programming concurrent and distributed actor systems. The platform is open source, and it is distributed through libraries implemented with the SCALA programming language. The middleware is based on the concept of Actor Model developed by Hewitt in 1973 [3]: as in the OOP, everything is an object, in the actor model, every entity in the system is an actor. In this work, we

refer to version 2.5.12 of the platforms, released on April 2018. Actors interact with each other by exchanging messages, and they are reactive concerning the linked communications (differently from an agent, an actor acts only if stimulated by the arrival of a message). The model is based on the prevalence of asynchronous communications between actors to guarantee the possibility of concurrent computational operations in the absence of blocking and locks methods. The prevention of computational conflicts is also guaranteed by the absence of states of shared memory between actors. The structure of Akka actors is hierarchical, and this allows the implementation of powerful exception management strategies. This feature, together with the ease of deployment of network distributed actor systems, has allowed Akka to feed a large community of developers and researchers and to be applied in a lot of commercial and industrial applications.

B. JaCaMo

JaCaMo is a framework for multi-agent systems programming, created by the combination of three technologies, each having a specific scope: *Jason*, *Cartago* and *Moise* [4]. They have been developed individually over the years. Therefore they are theoretically well-founded, stable and robust. Jason is an extension of *AgentSpeak* - a programming language which the main scope is to develop programs in the form of Plans and to exploit the BDI model (Belief - Desire - Intention). The main objective of the Cartago platform is to simulate the environment of the system. Such environment is represented by means of resources, called artifacts. Artifacts contain information that can be perceived or manipulated by agents. For instance, in an IoT (Internet of things) application, a temperature sensor and the heater are rendered as a couple of artifacts. Moise is a framework whose purpose is to model the organisation of entities, which are part of a multi-agent system. Management takes place through rules, groups and missions. Numerous papers propose possible multi-agent systems programmed with JaCaMo, for the management of industrial productions [5] [6].

C. JADE

Jade is a Java-based platform that simplifies the implementation of distributed applications by adopting the multi-agent systems paradigm. One of the noteworthy features of this platform is its complete compatibility with the FIPA standard. JADE was born in 1998, from the need to validate the FIPA specifications. Nowadays, updates to the development framework are released on an annual basis. In this work, we will refer to version 4.5.0 released on June 2017 [7]. A JADE platform is made up of a series of agents, distributed in various hosts. Each agent owns a container for the management of all of its services. At least one of the hosts, the so-called main container, will moreover keep a log containing all the information from the other containers, thus allowing agents to discover other ones on the network by using a white page directory. In this directory, agents are identified by an AID and yellow page discovery service where agents may be searched according to the services they offer. Jade is one of the richest frameworks in terms of literature, tools and extensions. We mention just a couple of examples: *JADEX* for working with goal-oriented agents, and *JADEMX*, a bean generator allowing an agent to expose attributes, operations, and notifications in a *JMX-compliant* manner [7].

III. COMPARISON CATEGORIES

The proposed comparison will be based on a set of criteria, clustered in some main categories:

1) Principles of the agent model,
2) Behavioural model,
3) Reasoning and knowledge,
4) Interaction/communication model,
5) Social aspects.

The focus of this paper is on the first two categories, whereas the last fours categories fall apart the scope of this paper because of space concerns. Therefore, a detailed discussion of the first two categories will be presented in the following subsections. The other categories will be briefly presented in order to provide a complete picture of the overall comparison approach in subsect. III-C.

A. Principles

Our analysis will begin with the presentation of the theoretical principles upon which the three examined platforms are based. In particular, we will refer to:

- The **reference paradigm** adopted by the specific development framework. This refers to the theoretical model/architecture that characterizes the system.
- The **main entity** adopted by the specific development platform; in particular the definition of agent or actor will be presented in this section. This will help in identifying the common characteristics and the differences, at the conceptual level, among the three platforms.

B. Behavioural Model

The criteria contained in this category concern the approach and the granularity the three platforms adopt to define: the behaviour of system's entities, the job scheduling, the execution flow and the errors handling. We will analyse these aspects from two points of view: macroscopic (management of tasks) and computational (management of threads). More in details, the criteria in this category are defined as follows.

- The **behavioural entity** criterion describes the constructs, the three platforms provide, to implement actions to be performed by system's entities.
- The **scheduling management** criterion presents the instruments each platform offers to customize the management of the queue of tasks to be processed.
- The **main entity loop** criterion focuses on how the main entities of the three platforms evaluate and execute the active tasks in their runtime cycle.
- The **behaviour control flow** of the actors/agents will describe the constructs with which agents/actors activate their tasks.

- The **errors handling** criterion provides an overview of the runtime instruments for the management of execution errors and exceptions.

C. Other Categories

For the sake of brevity, this paper does not illustrate all the categories with the same level of detail. However, in order to provide the overall picture of the proposed comparison approach, all the remaining categories (and their inner criteria) are mentioned and briefly discussed in the following.

Reasoning and knowledge. The elements in this section examine the three platforms from the point of view of knowledge management and reasoning. This category clusters the following comparison criteria: the *knowledge model*, *ontology*, *decision making* and *environment*.

Interaction/communication model. This is one of the pillars of distributed systems, it represents the facilities provided by the development framework for the interaction among its entities. The interaction/communication capabilities allow such entities: 1) to enable collaboration/cooperation/competition, 2) to synchronise different activities and also 3) to improve the awareness of the execution context. This category includes the following elements: *communication support*, *synchronous vs asynchronous* messages, *speech act support*, *protocol support* and *content language support*.

Social aspects. In the examined frameworks, social structures that can be created for the runtime organisation of the entities play a fundamental role. This category includes the following elements: 1) *support for creating social structures* and 2) *coordination/government of social structures*.

IV. Comparison: Basic Principles and Behavioural Aspects

The results of this comparative study are discussed in the following with the support of Table I that is useful to highlight the main characteristics of each framework.

A. Principles

In this subsection we will discuss the *reference paradigm* and *main entity* rows of Table I. The discussion terminates presenting the main differences among the three platforms.

a) Reference paradigm: In the Akka platform, the reference paradigm is the **Actor model**. The first theorization of this concept was postulated by Carl Hewitt in 1973 [3]. Actors are the fundamental entity of this model: as well as in the Object Oriented Programming everything is an *object*, with respect to the actor model, everything is an *actor* [8]. Hewitt's work comes from the desire to postulate a model useful for the development of concurrent applications, eliminating the presence of blocks and semaphores for the management of thread traffic (related to the simultaneous invocation of methods) and instead entrusting the dynamism of the system to the only exchange of (asynchronous) messages between actors. This constraint, together with the absence of shared states of memory between the actors, allows to evade some of the typical obstacles of concurrent calculation.

In particular, the model requires that each actor is associated to its own *mailbox* that acts as a queue for incoming messages. Each actor processes only one message at a time while, in a totally decoupled manner, the mailbox allows other parts of the system to send messages without having to wait for the actor to finish the current activity. The model foresees that the behaviour of the actor is univocally determined immediately before and immediately after that the same has processed a message. In other words, the processing phase of a message constitutes a possible transition from one behaviour to another.

Each actor has its own base of independent knowledge and the model discourages the presence of states of shared memory and the exchange of mutable objects among actors [9]. Even if actors are conceived as single-threaded entities, they reserve the ability to dynamically spawn additional actors during runtime. This allows to balance the computational load by accordingly resizing system resources [10]. Thanks to these constraints, concurrency in actors is limited only by the availability of hardware resources and by the logical dependence inherent in the computation [11]. The general concepts of the model find specific application in some Akka features that we will see in detail below.

The JaCaMo framework is based on the agent model, and more specifically on the **BDI agent model**. The BDI model was created [12] looking at the human behaviour. The BDI model is based on the following concepts:

- *Beliefs*: the information the agent owns about the world.
- *Desires*: all the possible activities the agent can potentially complete.
- *Intentions*: those activities the agent has decided to enact for reaching some Goals.

One of the three JaCaMo components, namely JASON, is an implementation of the AgentSpeak language. The purpose of JASON is to develop programs in the form of plans by using Beliefs-Desires-Intentions model [13].

It is possible to identify Jade as one of the first middleware born to comprehensively support the FIPA Architecture. [14] The Foundation for Intelligent Physical Agents has, in fact, worked on the creation of schemes and standards that regulate the interactions between agents in order to develop increasingly scalable and open systems. In this direction, Jade is a platform enabling the realization of concurrent applications with agents that turns out to be fully FIPA compliant. The model with which the platform is built refers to the FIPA specification of an agent that is more focused on interaction properties of the entity rather than its mental or reasoning features. [14] The middleware consists of an environment in which agents are instantiated and operate, providing reference classes through which agents are implemented and a graphical tool for displaying system status at runtime.

b) Main entity: Among the three platforms, Akka is the only one that is not based on the concept of Agent but on that of **Actor**. Actors are reactive entities that perform methods and change their state according to messages reception. In

TABLE I
COMPARISON TABLE

	Akka	JaCaMo	Jade
Principles			
Reference Paradigm	Actor Model	BDI	FIPA Architecture
Main Entity	Actor	Agent	FIPA Agent
Behavioural model			
Behavioural entity	Agents are classes. Methods contain actions.	Agent specified in ASL language. Plans contain internal/external actions.	Agents are classes. Behaviours (classes) contain actions (methods).
Scheduling	Dispatcher message scheduling	Round-robin with priority among active intentions	Cooperative
Main Entity Loop	Incoming messages are translated into events.	Classical perceive-reason-act loop.	Finite State Machine
Behaviour Control Flow	Method invocation	Plan activation via Goal/Plan matching	Method Invocation
Errors Handling	Internal support for father/child error management	Plan failure produces events for plan recovery	Standard JAVA exception handling

particular, when an actor receives a message, it can: perform one or more methods, change its behaviour/status, create other actors or send messages [15]. Actors' pro-activeness is limited since they do not actually execute anything until they receive a message. Each actor is composed of three distinct elements i) a unique actor name (used as an address for communication); ii) an expression which describes the actor's behaviour; and iii) a mailbox that stores received messages as a *FIFO* list [10].

The Akka actor is basically a class with a single *receive* method that will be called when a new message is available to be processed in the mailbox [9].

The system supports a powerful hierarchy relationship between actors: each actor is the son of another one and its parent is responsible for delegating some tasks and managing the exceptions of its sons.

JaCaMo and Jade are both based on the **Agent Model** but these frameworks follow a slightly different definition of agents.

JaCaMo derives from AgentSpeak, which definition is aligned with the classical vision of Wooldridge and Jennings in 1995 [16] that identifies the following characteristics for an agent:

- **autonomy**: agents acts without any human control.
- **social ability**: agents interact with other agents and other through the agent-communication-language.
- **reactivity**: agents perceive information from the environment and respond in short time if it is possible.
- **pro-activeness**: agents not only react to perceptions but take the initiative to achieve goals.

In JaCaMo, agents are mainly defined by their beliefs, plans, and goals.

Jade derives from the FIPA specifications, for which an agent is "a computational process that implements the autonomous, communicating functionality of an application" [14]. Agents are implemented as an instantiation of the Agent class that contains some basic attributes and methods needed to start and manage the agent. In particular, the *setup()* method is responsible for the creation of the agent and for its registration in the Agent Management Service (a kind of white-page service).

A noteworthy feature of Jade systems is its Peer-to-Peer architecture. In particular, the system is: fully distributed, it has an efficient transport of asynchronous messages via a location-transparent API, it implements the white pages and yellow pages for the knowledge of the agents within the system, it fully supports ontologies and many other features [7].

c) Discussion: The analysis conducted on the reference principles of the three frameworks easily identifies two macro-categories: actors and agents. The two models characterize the analysed frameworks from the conceptual point of view and this tendency can be found in the attention given by the same towards the reasoning of the agent, the interaction between agents and the management of events. In particular, it is possible to note the attention given by the actor model to the events management that influences the workflow of the system: this approach exploits the reactivity of the agents instead of their reasoning capacity. In Akka's implementation, the lightness of the actor model favours the mobility of the code and therefore the flexibility of application deployment. The agent model focuses on a completely different direction, especially regarding its implementation in JaCaMo through the BDI model: in this case, it is the agent's reasoning that is exploited and the internal actions play in this context a fundamental role. As already highlighted, the model proposed by Jade, and therefore the standard created by FIPA for agents systems, stresses the concept of interaction between agents, providing a functional architecture for communication between agents.

B. Behavioural model

In this subsection we will discuss the *Behavioural entity*, *Scheduling*, *Loop*, *Control Flow* and *Errors Handling* subcategories of Table I. The section is organised as follows: the first five paragraphs will detail the features of the selected platforms as regards the five subcategories while the last paragraph will discuss the main differences.

a) Behavioural entity: The behavioural entity of Akka actors is linked to the invocation of methods contained within the actor *receive* construct. The latter consists of a *partial function* [1], supported by Scala, that is responsible for performing a pattern matching with the messages received from the actor [9]. The behaviour of the actor, and consequently the set of methods associated with its *receive* construct, is not static and can be modified by means of the *become* and *unbecome* functions. These functions allow, depending on the events recorded by the system, to characterize the abilities of the actor based on the specific task it is called to perform [13].

Plan is the Behavioural entity in JaCaMo. Even if JaCaMo is entirely implemented in JAVA, it provides a specific language to support a logical programming paradigm. Plans are the main elements in this programming approach. A plan is described by a trigger event (an event to which the plan can respond), an activation context (a logic condition that must be true in order the plan can be activated) and a body. The body of the plan contains instructions such as internal actions, operations on the knowledge, expressions and events raising. For instance, a plan can request the satisfaction of a goal, that will generate a new event, and in turn, that will activate another plan.

In Jade, the agent behaviour may be specified by exploiting the Behaviour abstract class. Each subclass defines an atomic portion of the agent's behaviour by overriding the *action* method. The platform provides many ready to be used types of behaviour classes [7]. Among them we mention the 'OneshotBehaviour' in which the *action* method is executed only once. The 'CyclicBehaviour' in which the *action* method is executed in a loop until a condition is true. Another noteworthy class is the CompositeBehaviour class which contains subclasses useful for complex tasks. The subclasses are SequentialBehaviour, FSMBehaviour and the ParallelBehaviour, they will be discussed in the Scheduling subcategory.

b) Scheduling: Akka actors are single-threaded. Computational concurrency is committed to their ability to create new actors to which they delegate tasks to be processed. According to the Scala programming language, each thread has its own priority level that guarantees processing preemption with respect to all the lower priority threads. The processing of equal priority threads is delegated to the JVM [9]. In Akka, tasks to be performed consist of messages to be processed and the platform devotes a specific library for the implementation of tools and methods for managing such workload. The *Message Dispatcher* class handles how each request, received by the actor, is processed. There are two executors available with the default dispatcher. These are the *fork-join-executor* and the *thread-pool-executor*. The first executor forks a separate thread for each request to the Actor and then waits to rejoin that thread before continuing. The thread pool executor uses a pool of threads to process multiple requests in parallel across multiple instances of an Actor [9]. The overhead of an actor system is relatively low in terms of memory consumption. Each actor consumes about 300 bytes of memory, which allows for roughly 3 Million actors for GB of main memory [18]. *Router actors* are another important feature of Akka platform. This kind of actors is able to deliver messages from one actor to another, choosing the recipient according to specific strategies that respond to developer's needs. A normal actor can be used for routing messages, but a single-threaded actor may become a bottleneck. Routers can achieve much higher throughput with an optimization to the usual message-processing pipeline that allows concurrent routing [19].

Scheduling in JaCaMo manages the Intention queue through the so-called Intention Selection function (Si). There are two types of intention: internal and external. Internal intentions are created when a plan is executed, and a sub-plan must be executed inside it. External intentions are created when an event occurs, for example changes in the environment are perceived or a goal is delegated from another agent. The predefined Si function uses a Round-Robin strategy for external intentions, giving priority, as if it was a stack, to internal intentions. Therefore, the sub-plan obtains a higher priority than the external intention. [20].

Scheduling in JADE is not-preemptive (cooperative). This means that if an *action()* method of a behaviour is launched, it will block the agent until it completes. When dealing with complex tasks, it is not convenient to create a 'unique' *action()* method that would require long time to complete. The suggested solution is to use the CompositeBehaviour class. An instance of the CompositeBehaviour class is itself a Behaviour composed of other child sub-behaviours. In particular, the platform provides three different execution strategies for a CompositeBehaviour: the SequentialBehaviour, the FSMBehaviour and the ParallelBehaviour [7]. The SequentialBehaviour class allows performing sub-behaviour sequentially. The FSMBehaviour class allows the sub-behaviour to be performed according to a finite state machine. The ParallelBehaviour class schedules the sub-behaviour in parallel.

c) Main Entity Loop: The execution of an actor in Akka takes place according to the *event loop* algorithm. According to this scheme, the actor is at any moment in a well-defined state within an infinite loop waiting for the arrival of an event at the event queue. As soon as one event is available, it is fetched and a corresponding event handler is executed, if available. When the event handler execution is terminated, the control flow goes back waiting for the next event [13]. Transitions in a state are triggered by new messages and the atomic execution of the message handler represents the effect of the transitions, atomically changing the state. This type of main entity loop is well suited to the implementation of entities modelled as *Finite State Machines* [13].

In JaCaMo, the execution cycle is a control loop [13]. The peculiarity of this architecture is the ability to join two fundamental aspects of the agents: pro-activity and reactivity. At each loop cycle, the agent's reasoning cycle, based on the BDI model, is composed of three macro stages: perceive, reason, act. In the perception phase, the agent acquires information

[1]Accordingly to mathematic definition, a partial function of type *PartialFunction[A, B]* is a unary function where the domain does not necessarily include all values of type A. The function *isDefinedAt* allows to test dynamically if a value is in the domain of the function [17].

from the environment to update its beliefs, in the 'belief base'. Jason provides a series of classes for managing the environment. In addition to acquiring information from the environment, the agent receives messages from other agents. For each reasoning cycle, only one message at a time is processed by the interpreter. At this point, if the agent has perceived changes in the environment or changes in its goals, events are generated. For each reasoning cycle, only one event will be selected, events are scheduled in a FIFO queue, but that can be changed by the programmer [20]. To react to the selected event, the agent must find a plan that matches the event. If there are multiple plans it will create a list of 'desires'. This list will be filtered through the "Check Context" which will select only applicable plans. Finally, these 'intentions' will be inserted in the queue of intentions where the scheduler will appropriately select which one to execute at the next reasoning cycle.

The Jade loop is managed through a Finite State Machine (FSM) where each agent can be in different states during its lifetime. The states are [21]:

- *Initiated*: the agent was created with the *setup()* method. It can not go to the Active state until it is registered on the AMS (Agent Management System).
- *Active*: the Agent has been registered in the AMS so it can perform his behaviour.
- *Waiting*: the agent is blocked, wait until activation messages arrive.
- *Suspended*: the agent is stopped, so none of its methods is performed.
- *Transit*: the agent stays in this state until he has completed his migration.
- *Unknown*: the agent has terminated his life-cycle, so the AMS cancels it.

d) Behaviour Control flow: The control flow in Akka is essentially managed by the *receive* method. This construct, as already said, is a *partial function* of Scala that has the task of carrying out the pattern matching action between the messages received from the actor and the types of tasks the same is able to manage. The construct is implemented at the syntax level like the *switch case* [18]. Each time the actor receives a message, the *receive* method is invoked and the system verifies the correspondence of the content with one of the cases contained in the method. The execution of the task by the actor is subordinated to the pattern matching phase. When an actor has a nonempty mailbox and it is executed within a thread, the first message is taken out of the mailbox, processed by the actor's code (maybe modifying some internal state, creating actors, sending messages, etc.), and finally the execution finishes [18]. As mentioned above, the *receive* method could be replaced to modify the behaviour of an actor. This modification is made possible by the *become/unbecome* methods invocation [19].

The control flow in JaCaMo is implemented through plan activation by goal/plan matching. A plan is formed by three elements: the triggering event, the context and the body. More in details [20]:

- triggering-event: the proactive behaviour of an agent is determined by the fact that it has to achieve long-term goals. While dealing with these goals, it must also be concerned with both the changes in the environment and the information coming from other agents because they could change the objectives. This happens through changes in beliefs and changes in goals to be achieved. These changes cause events and agents are activated by them. Plans are courses of actions that are executed as a consequence of such events. If an event matches the triggering-event, then it is said that the plan is "relevant" to that particular event.
- Context: there may be several 'relevant' plans, but the agent can execute only one. To select the specific plan, a "filtering" is carried out, choosing the foreground that respects the constraints defined in the context.
- Body: the body of the plan that will be executed.

The control Flow in Jade is implemented through method invocation. In particular, when an agent is in the "Active" state, then it can execute its behaviour as specified in the *action()* methods.

e) Errors handling: Akka has a specific feature for runtime errors handling. This functionality is based on the hierarchical structure of actors committing to each parent the management of the error circumstances in which their children incur. In particular, in case of an error, an actor suspends its functioning and that of all its children actors, then it sends a message to its parent actor where it specifies the error type [9]. The error handling phase is managed through a pattern matching procedure that is similar to that of the *receive* method [19]. The default supervisor strategy requires the parent actors to reboot their child when it incurs an error. The restarting process restores the initial conditions of the actor while keeping intact its mailbox (only the message that led to the error may be lost). Overriding the supervisor strategy it is possible to customize the action to be taken on the base of the type of error [18].

JaCaMo does not contain specific tools like Akka for error handling. The failure of a plan is one of the errors that can be handled. Regardless of why a plan can fail, the Jason interpreter generates a goal cancellation event. There are three main reasons for a plan to fail [20]:

- Lack of the agent knowledge, usually due to the lack of the sub-goal within a plan. This can happen either because the programmer has not provided the required plans, or because, even if there is the plan to the goal associated, there are no beliefs that match with the context.
- Failure of a test goal, because in the belief base there is no information necessary to meet that goal. At this point, the agent generates an event for a plan to be executed, hoping for a positive answer for the goal test. If this is not successful, then it will generate a failure for the plan.
- Action failure. In JaCaMo the actions can be internal, or

external. If an action fails, then the plan will appear to have failed.

Errors Handling in Jade is managed through the classic Java exception handling.

f) Discussion: The analysis of the behavioural model of the three frameworks reveals already outlined differences. In fact, each sub-category presents native tools and functions specifically designed to support the agents' reasoning and the concurrent execution of the processes, according to the Actor model, the BDI Paradigm and the FIPA standard. In particular, the orientation towards communication and messages adopted by Akka and Jade is evident, as opposed to the attention dedicated by JaCaMo to internal actions and therefore to perception of the environment and to the agent's reasoning. The same logic is clearly reported in the contribution of Ricci where the difference between *event loops* and *control loops* is clearly discussed, also differentiating agents and actors for what concern their predisposition to reasoning and proactivity. From this point of view, the *FSM* logic adopted by Jade centralizes the communication in the behaviour model of the agent, nevertheless leaving the latter a certain degree of autonomy. Besides, at the programming level, there are relevant differences in the approach between the pattern matching of messages made by Akka and the JaCaMo agents' continuous search for a plan to be performed in order to achieve a goal. All the three platforms, while primarily committing to the JVM thread scheduling operations, offer specific tools for controlling the workload at the computational level. Finally, it is important to underline the specific functionality that Akka offers in the dynamic management of errors at runtime.

V. Comparison: Overview of Other Categories

In the following the features of the three platforms with regard to the other categories of the proposed comparison framework will be briefly summarized in order to provide the reader with a comprehensive view on the results of the comparison.

A. Reasoning and Knowledge

In this subsection we will briefly discuss the *Knowledge Model*, *Ontology Support*, *Decision Making*, and the *Environment* subcategories.

a) Knowledge Model: From the point of view of the programming paradigm, information available to each actor consists of objects encapsulated within its own class and methods contained within the *receive* construct. This structure is compliant with the absence of shared memory between actors [9].

One of the most important elements in JaCaMo is the information that the agents store both external (environment) and internal. This information is stored in a list called "belief base" [20].

In Jade, the agent's knowledge is stored as attributes of the agent class. It is relevant to note that agents do not have a shared memory [7].

b) Ontology Support: Akka does not support any specific language for the implementation of ontologies. Nevertheless, there are some examples of libraries developed to facilitate the integration of ontological languages within Akka. One of these libraries is *Scowl* [22].

The JACAMO platform does not support any specific ontology language, but the user can optionally choose to set one according to the system-to-be (ad hoc ontology) [23].

In Jade, it is fundamental to define a shared ontology among the different agents in the same JVM therefore the Jade platoform provides a great support for that [7].

c) Decision Making: As highlighted above, the behaviour of actors in Akka is not proactive. They have no independence in taking decisions and each message activates the invocation of methods that do not represent, in the actor perspective, a part of a wider goal [13].

JaCaMo is an agent-based system, so agent behaviour is both pro-active and reactive. So the agent achieves its goals not only driven by events but also taking an initiative, recognizing the opportunities it has in the environment [20].

With available extensions like Jadex, Jade can perform high-performance goal-based reasoning in a BDI way.

d) Environment: In Akka both data control modules and sensors may be modelled as actors that communicate through messages. [19]. It is also appropriate to refer to the already mentioned *Event bus* that is a kind of blackboard in which actors can publish information and subscribe for news published by other actors [19].

The environment plays a fundamental role in the design and development of a multi-agent system. The environment is a source of facilities and services that agents can adequately use at runtime to support and improve their individual and social activities. As already discussed, in JaCaMo the environment is modelled with Cartago through an extension of the Java Artifact class [24].

JADE does not offer a specific support for interaction with the environment.

B. Interaction/Communication Model

In this subsection we will briefly discuss the *Implicit/Explicit Interactions*, *Synchronous/Asynchronous Interactions*, *Speech Act Support*, *Communication Support*, *Agent Interaction Protocol Support*, *Agent Content Language Support*, and the *Communication Channel* subcategories.

a) Implicit/Explicit Interactions: Akka platform supports both explicit and implicit communications. Explicit communication can be processed, for instance, through the Event Bus tool that provides a kind of blackboard for actors' implicit interactions [19].

JaCaMo platform supports both explicit and implicit communications.

JADE only supports explicit communications based on FIPA-ACL.

b) Synchronous/Asynchronous Interactions: Messages in Akka are natively asynchronous. Synchronous message exchange is implemented at the syntax level by the *Ask* operator [18].

In JaCaMo, the asynchronous method is predominantly used. The synchronous method could be used by adding the "reply" parameter when sending messages, with the only "askOne" and "askAll" speech acts. In this case, the sender will wait for the message ACK [20].

JADE message exchange is asynchronous.

c) Speech Act Support: Actors can send messages through two methods. The *tell* method sends a non-blocking message to another actor, while the *ask* method sends a message and returns a *Future* object. [19].

Speech acts in JaCaMo are limited. The list of available performatives is [20]: tell, untell, achieve, unachieve, askOne, askAll, tellHow, untellHow.

Jade fully supports FIPA-ACL and therefore it offers the possibility to adopt every performative.

d) Communication Support: The communication support between actors is implemented only through the *Ask* construct. In other types of communication there is no link between messages in a conversation [19].

The communication support in JaCaMo is implemented exclusively through the synchronization of messages, in particular with the speech act "askOne" and "askAll" [20].

As already mentioned, Jade uses FIPA-ACL for communication. One of the parameters in this standard is "conversation-id" that permits to uniquely identify a conversation thread [7].

e) Agent Interaction Protocol Support: Interaction protocols are not natively implemented in Akka. Despite that, some works tried and succeed to implement a *DSL* that can make Akka compliant with the FIPA interactions protocol specifications [8].

JaCaMo provides no specific support for agent interaction protocols.

JADE has a package named "jade.proto" that contains all the classes providing a full support for implementing standard interaction protocols [7].

f) Agent Content Language Support: Akka provides an agent management service through the hierarchical organization of actors, and a message transport service allowing communications across machines. However there is no directory facilitator so there is no full Agent Content Language support [8].

JaCaMo does not have any content language support but its language is very similar to KQML. If the JADE extension is used, JaCaMo acquires a full support for all the standard languages used by JADE [20].

As mentioned before, the Jade framework is fully FIPA compliant. This property is guaranteed by the presence of all the infrastructure services prescribed by the standard (AMS, DF and MT) [7], this ensures the best support for the agent content language.

g) Communication Channel: Although Akka Routers are mainly used to balance the workload of actors, they also act as a communication channel between actors, allowing them to be uniquely identified even across different nodes in a distributed architecture [19].

JaCaMo uses messages for explicit communications, and artifacts as an implicit communication channel. An artifact is used as a sort of blackboard by agents thus allowing them to acquire information from other agents and the environment. [20].

JADE fully implements a FIPA compliant Message Transport Service that is responsible for transporting FIPA-ACL messages between agents within and outside the system. According to the MTS each message contains an envelope that comprises parameters representing information about the communication [7].

C. Sociality

In this subsection we will briefly discuss two subcategories: *Support for Social Structures*, and *Coordination/Government Models for Social Structures*.

a) Support for Social Structures: In Akka each actor is generated by another actor which is labelled as 'parent'. Each parent is interested in the execution of the tasks committed to its children and it is responsible for the relative supervisor strategy [9].

Moise is the framework that allows the definition of organizations in JaCaMo. There are two main approaches for creating an organizational structure within Moise: an agent-centred one and an organization-centred one [25].

Jade natively does not offer any specific tool that supports the definition of social structures.

b) Coordination/Government Models for Social Structures: When creating an entity, an actor assumes the role of delegator towards the generated children: each actor can delegates the tasks to be performed to actors specifically generated in its own *context*. [19].

In Moise, coordination is managed through a set of rules that constrain the behaviour of the agents. When an agent is placed in an organisation, it becomes part of a network of obligations, interdictions, and permissions [25].

Since in Jade there are no social structures, there are no rules or models for agent coordination.

VI. Conclusions

In the article, we presented the main models' characteristics and the functionalities of the three frameworks analyzed. For reasons of brevity, we have given ample space to the description of the first two categories (Principles and Behavioral model), briefly summarizing the results obtained in the other three categories of the comparison. The descriptions are made using both the already existing scientific literature and the official documentation and reference manuals of the three frameworks. From the reported analysis clearly emerges how the three frameworks adopt different perspectives to reach the common objective of developing applications for distributed and concurrent systems. In fact, they present significant differences. For instance, despite all the three platforms adopt asynchronous communications to cope with the risk of conflicts in the processing of threads, each one differs from the others for particular functionalities. These functionalities

may be linked to the model of knowledge, to the reasoning of agents, to agents' reactions to events, to the scalability of the system or to communication protocols. These features are reflected in the reference models of the three frameworks and enrich each platform with specific libraries and tools for developers' benefit. In the future we will deepen the comparison by enriching the analysis of the categories reported in sect. V and we will also compare the programming language by applying them to a notable case study as already done for the sole Akka and JaCaMo in [13].

REFERENCES

[1] L. Fichera, F. Messina, G. Pappalardo, and C. Santoro, "A python framework for programming autonomous robots using a declarative approach," *Science of Computer Programming*, vol. 139, pp. 36–55, 2017.

[2] V. Mascardi, D. Demergasso, and D. Ancona, "Languages for programming bdi-style agents: an overview." in *WOA*, 2005, pp. 9–15.

[3] C. Hewitt, P. Bishop, and R. Steiger, "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence," vol. 3. Stanford Research Institute, 1973, p. 235.

[4] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi, "Multi-agent oriented programming with jacamo," *Science of Computer Programming*, vol. 78, no. 6, pp. 747–761, 2013.

[5] M. L. Roloff, M. R. Stemmer, J. F. Hübner, R. Schmitt, T. Pfeifer, and G. Hüttemann, "A multi-agent system for the production control of printed circuit boards using jacamo and prometheus aeolus." IEEE, 2014, pp. 236–241.

[6] R. Martins and F. Meneguzzi, "A smart home model using jacamo framework." IEEE, 2014, pp. 94–99.

[7] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing multi-agent systems with JADE*. John Wiley & Sons, 2007, vol. 7.

[8] G. Weichhart and C. Stary, "A domain specific language for organisational interoperability." Springer, 2015, pp. 117–126.

[9] J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*. Springer, 2018.

[10] J. Masini and A. Francalanza, "Typing actors using behavioural types," 2015.

[11] G. A. Agha, "Actors: A model of concurrent computation in distributed systems." Tech. Rep., 1985.

[12] M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge, "The belief-desire-intention model of agency," in *International Workshop on Agent Theories, Architectures, and Languages*. Springer, 1998, pp. 1–10.

[13] A. Ricci, "Programming with event loops and control loops–from actors to agents," *Computer Languages, Systems & Structures*, vol. 45, pp. 80–104, 2016.

[14] "Fipa specification." [Online]. Available: http://www.fipa.org/specs/fipa00001/SC00001L.html

[15] A. Rosà, L. Y. Chen, and W. Binder, "Profiling actor utilization and communication in akka." ACM, 2016, pp. 24–32.

[16] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *The knowledge engineering review*, vol. 10, no. 2, pp. 115–152, 1995.

[17] M. Odersky, "Scala documentation." [Online]. Available: https://www.scala-lang.org/api/2.12.1/index.html

[18] M. Thurau, "Akka framework," *University of Lübeck*, 2012.

[19] J. Bonér, "Akka documentation." [Online]. Available: https://akka.io/docs/

[20] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons, 2007, vol. 8.

[21] F. Bellifemine, G. Caire, T. Trucco, and G. Rimassa, "Jade programmer's guide," *Jade version*, vol. 3, pp. 13–39, 2002.

[22] Balhoff, "Scowl: a scala dsl for programming with the owl api," 2016.

[23] A. Freitas, D. Schmidt, A. Panisson, R. H. Bordini, F. Meneguzzi, and R. Vieira, "Applying ontologies and agent technologies to generate ambient intelligence applications," in *Agent Technology for Intelligent Mobile Services and Smart Societies*. Springer, 2015, pp. 22–33.

[24] A. Ricci, M. Viroli, and A. Omicini, "Cartago: A framework for prototyping artifact-based environments in mas," in *International Workshop on Environments for Multi-Agent Systems*. Springer, 2006, pp. 67–86.

[25] M. Hannoun, O. Boissier, J. S. Sichman, and C. Sayettat, "Moise: An organizational model for multi-agent systems," in *Advances in Artificial Intelligence*. Springer, 2000, pp. 156–165.