# Finding new routes for integrating Multi-Agent Systems using Apache Camel*

**Cleber J. Amaral[1,2], Sérgio P. Bernardes[1], Mateus Conceição[1]**
**Jomi F. Hübner[1], Luis P. A. Lampert[1], Otávio A. Matoso[1], Maicon R. Zatelli[1]**

[1]Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brazil

[2]Instituto Federal de Santa Catarina (IFSC)
São José – SC – Brazil

cleber.amaral@ifsc.edu.br, {sergiopb1998,lp.lampert}@gmail.com
mateusconceicao1@hotmail.com, {jomi.hubner,maicon.zatelli}@ufsc.br
otaviomatoso@yahoo.com.br

***Abstract.*** *In Multi-Agent Systems (MAS) there are two main models of interaction: among agents, and between agents and the environment. Although there are studies considering these models, there is no practical tool to afford the interaction with external entities with both models. This paper presents a proposal for such a tool based on the Apache Camel framework by designing two new components, namely* camel-jason *and* camel-artifact*. By means of these components, an external entity is modelled according to its nature, i.e., whether it is autonomous or non-autonomous, interacting with the MAS respectively as an agent or an artifact. It models coherently external entities whereas Camel provides interoperability with several communication protocols.*

## 1. Introduction

MAS literature has plenty of research about agents' interactions, i.e., agents sending and receiving messages to and from other agents (A-A). Many approaches model almost any entity as an agent and thus the interaction remains something among agents. However there are new approaches that questioned the *agentification* method proposing an MAS where non-autonomous entities are conceived as artifacts in the environment. In these approaches, the development of an MAS considers the design of both agents and artifacts. The environment is not simply what is outside the system (the exogenous environment), but it is designed accordingly to the system purpose (the endogenous environment) [Ricci et al. 2006, Omicini et al. 2008]. In this sense, we have two models of interactions: agent-to-agent (A-A) and agent-to-environment (A-E). In the former, an agent *communicates* with another agent using an Agent Communication Language (ACL) and in the latter, an agent *perceives and acts* upon artifacts in the environment.

When we consider the integration with other applications, those two models are adopted by the current development platforms. On the one hand, we have approaches that use ACL for that purpose and other applications are seen by the agents as other agents (having a mental state, implied by the ACL semantics). On the other hand, we

*Supported by Petrobras project AG-BR, IFSC and UFSC.

have approaches where other applications are seen as part of the environment and agents perceive and act on them. Some platforms provide an A-A approach while others an A-E approach, but, as far as we know, no platform provides both. The designer is forced to conceive some other application either as an agent or as an artifact, despite the application properties.

In this paper, we propose to apply the same argument as [Omicini et al. 2008] for the integration of MAS and external applications: some external applications are autonomous and should be modelled as agents while others are non-autonomous and should be modelled as artifacts. For instance, in the Industry 4.0 context, it is expected the interaction of many entities such as an autonomous planner sending commands to a non-autonomous machine, which signalises what was done. Later, the planner must choose a supplier after an auction to hire a freight to take the product to the destiny, which is usually an human. This short example gives an idea of how comprehensive and challenging the integration can be. We can notice that both models of integration are required: the *autonomous* planner and the human are better modelled as agents, performing A-A interactions, and the *non-autonomous* machine should be integrated as an artifact which when communicating with and agent performs and A-E interaction. Following this concept, we have developed two components for JaCaMo platform, for integration among agents, and between agents and the environment. The referred components are used to set communication routes for the MAS and external entities, using the framework Apache Camel [Ibsen and Anstey 2010], a mediation tool to provide interoperability with many technologies.

## 2. MAS integration approaches

MAS are being applied as a core technology for distributed systems that needs cooperation and negotiation [Roloff et al. 2016]. The integration of MAS and external entities, i.e. any entity which was not defined its totality within the MAS itself, regards concerns such as compatibility with standards, interoperability and portability. We have found two main forms of integration: (i) among agents (A-A); and, (ii) between agents and the environment (A-E).

### 2.1. Integration among agents (A-A)

The communication among agents is usually done by speech-acts which considers utterances as actions, usually intending to change the mental state of recipients. The utterance can inform beliefs, desires and intentions of rational agents that attempt to influence other agents. The Knowledge Query and Manipulation Language (KQML) is the first speech-act based language providing high level communication in the distributed artificial intelligence applications [Vieira et al. 2007]. In fact, once speech acts became widely accepted in MAS community, the integration among different agent's platforms was facilitated.

Currently FIPA-ACL, which is very similar to KQML, is the main standard for agents communication. FIPA-ACL uses performatives to make explicit an agent's intention for each sent message, for instance, *inform* is used to influence the recipient to believe in something, and *request* to influence the recipient to add something as a goal [Vieira et al. 2007].

Another communication aspect is related to the expected sequence of messages. Conversations among agents usually follow some patterns which are often referred to

as interaction protocols. Typical patterns such as negotiation, auction, and task delegation are defined using FIPA standards [Bellifemine et al. 2005]. In addiction, there are communication infrastructures that allow agents to be distributed over a network. The challenge in A-A is the integration between an agent, for instance, using FIPA-ACL, and another agent using another language, for instance, an human. This situation leads to the necessity of some tool to make both end-points compatible.

### 2.2. Integration between agents and the environment (A-E)

There are systems or parts of a system which are better seen as resources or tools that can be used by agents to achieve their goals. These entities, called artifacts, have no internal goals, they are not autonomous and neither proactive, but they supply useful functionalities for agents. An analogy for agents and artifacts is the interaction between humans, as autonomous entities, and tools they exploit in their activities [Ricci et al. 2006]. For instance, a blackboard shared by agents would be modelled as an artifact, being predictable and deterministic, if not, it would perform undesirable autonomous behaviour.

Artifacts are placed in workspaces which represent areas of the MAS environment. Agents can perceive changes and act within the workspaces they are occupying, and on artifacts they are watching, i.e., being aware of events and signals due to interactions with non-autonomous entities held inside of virtual boundaries. The environment can reflect the effects of agents' actions and other phenomena. It is being treated as a first-class programming abstraction with similar importance of agents programming abstraction [Ricci et al. 2006].

Besides modelling non-autonomous entities, the artifacts can also be used for other purposes: (i) for agents coordination using shared artifacts such as organisational boards or coordination marks; (ii) for indirect communication among agents, for instance, by blackboard artifacts; (iii) for implementing the user interface of a system; (iv) for controlling transactions over environment elements through distribution and synchronisation facilities; and (v) for integration between the MAS and external entities [Boissier et al. 2019].

Regarding the use of artifacts to integrate external entities, the integration is done usually through specific Application Programming Interfaces (APIs). The main concern with this approach regards to the high programming effort when there are different protocols in scenarios of heterogeneous devices.

## 3. Integrating A-A and A-E using Camel

Back to our example in the Industry 4.0 context, Figure 1 shows a process that begins with a packed product, in a production line, up to its delivery to the customer. On the first step, there is an industrial device, a non-autonomous entity, that communicates using an industrial protocol. Once the device signalises the end of the production, the order is checked out on the Enterprise Resource Planning (ERP) software, another non-autonomous entity. The supplier should choose the best offer for a freight, which may be done by accessing suppliers' systems to then interact with the winner, an autonomous entity. Later, the delivery should be tracked by a monitoring system, which is non-autonomous. Finally, when the product is near the destination, a message must keep the client, an autonomous entity, informed.

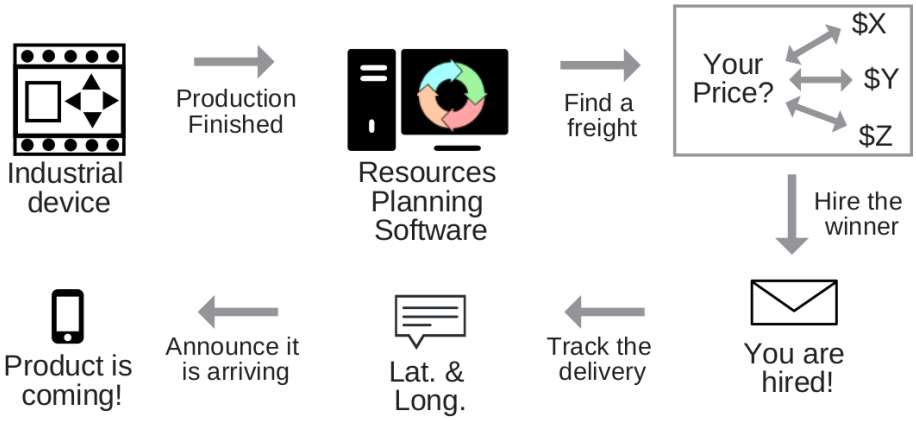


**Figure 1. Finishing production and delivering the product in Industry 4.0 context**

This scenario illustrates the requirements for the integration: heterogeneous endpoints with different kinds of interactions due to the autonomous and non-autonomous nature of entities. We propose the use of the framework Apache Camel for both integration models, A-A and A-E. We have thus two components: *camel-jason* for integrating MAS's internal agents with external entities modeled as agents, and *camel-artifact* for integrating the former agents with external entities modeled as environmental artifacts.

### 3.1. Apache Camel

Apache Camel is a lightweight Java-based framework message routing and mediation engine [Ibsen and Anstey 2010]. Camel achieves high-performance processes handling multiple messages concurrently, and provides functions such as routing, exception handling, and testing. It uses structured messages and queues based on Enterprise Integration Patterns (EIP) [Hohpe and Woolf 2003], preserving loose coupling among the resources. Camel works as a middleware that can be incorporated into an application through the use of *components*. Communication among Camel components is defined in so-called *routes*, which set and manage how messages will be exchanged, possibly following sets of rules and using data manipulation.

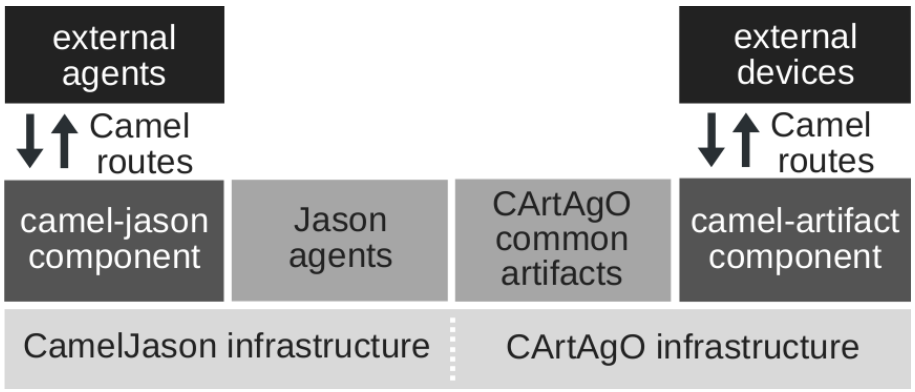


**Figure 2. MAS architecture using *camel-jason* and *camel-artifact* components.**

Routes define a single endpoint for each entity with an unique address. The defined endpoint may receive data, through a *producer* and send data through a *consumer*. Consumers are entities that admit data in specific formats and encapsulates it in a camel

exchange object, an item that any other producer understands and is able to decode. Producers are entities that receive encapsulated data from the consumer decoding it in its entity's message structure.

In our implemented components, Camel is being embedded in two slightly different manners regarding the models of integration, A-A and A-E, as shown in Figure 2. In the case of A-A, it works as a communication infrastructure that is used when the recipient is not found locally. In the case of A-E, the external device is usually modelled reflecting real operations and signals that it generates, typically having their individuals routes. In both cases, the components are able to define tuned integration, covering a range of endpoints features. Notice that, the complexity of each supported protocol is processed in a Camel component, which works as a bridge to Camel routes. There are more than two hundred components available on Camel's website[1] and many others on the community's repositories.

### 3.2. *camel-jason* component

The *camel-jason* component enables agents to communicate with external entities through ACL, whilst fulfilling the need of understanding those entities as agents when modeling the MAS. In our proposal, the external entity has a kind of virtual counterpart inside the MAS, a *dummy agent*. This counterpart is seen by the agents as an ordinary agent of the system. Doing so, agents can directly communicate with external entities assuming that they are other agents (as receivers and senders of ACL messages).

*Camel-jason* component provides a communication flow that is illustrated in Figure 3 where an agent interact with a service A (an external entity). Since the agent sees the service as another agent, it uses ACL for the communication. When the service wants to contact the agent, the camel-jason component translates the message into ACL and the agent receives it as if it comes from an agent.

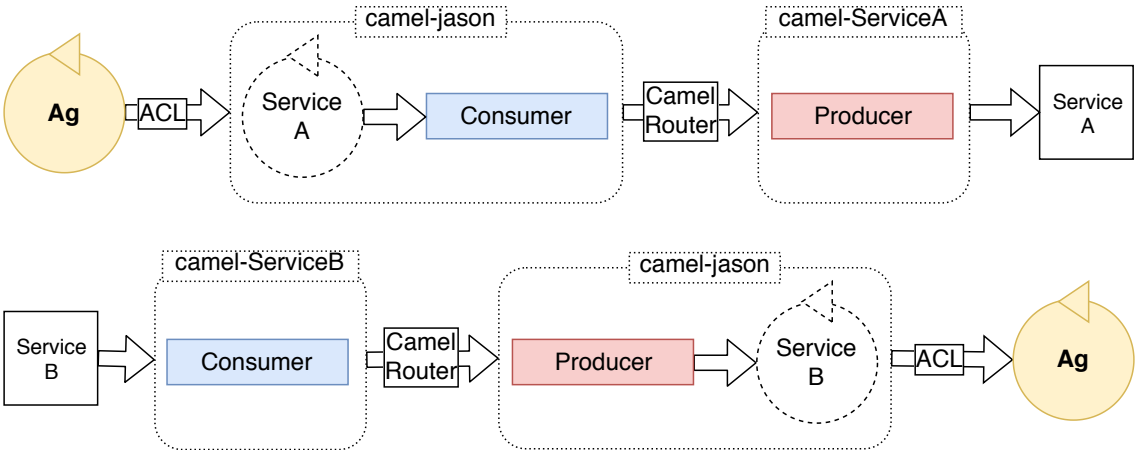


**Figure 3. Communication flow using *camel-jason* component.**

On the top communication flow showed in Figure 3, we have an agent sending a message to an external autonomous entity: (i) the agent send an ACL message addressed to a dummy agent, which is created by *camel-jason* component, referring to *Service A*;

[1]Supported Camel components are listed in http://camel.apache.org/components.html

(ii) the message is consumed by *camel-jason* component consumer; (iii) the message is exchanged to the other side of the route, possibly being transformed; and (iv) the message is processed by *Service A* component producer which prepares a service A compliance message, which will be sent to some network address to be effectively consumed by the *Service A*.

In the other way around, on the bottom of Figure 3, we have: (i) *Service B* sends some data through the network reaching *Service B* component consumer by its network address; (ii) the message is exchanged through Camel route, possibly being transformed; (iii) the message is processed by *camel-jason* component producer which generates an ACL message; and (iv) the receiver agent effectively consumes the ACL message.

The component uses a simplistic method to define the communication routes, in which for many cases no actual programming is required, only XML definitions. The user should know how to fill camel endpoint parameters according to the compatible endpoint of the application. In cases data transformation is required, camel brings some tools for simple transformation as well as complex ones, using embedded programming codes if needed.

### 3.3. *camel-artifact* component

In order to sustain the A-E model, the CArtAgO infrastructure is used, and the *camel-artifact* component was developed. This component allows agents to perceive and act upon artifacts that represent external entities inside the MAS.

Notwithstanding, *camel-artifact* also allows the definition of communication routes between CArtAgO artifacts and external entities. Routes for the *camel-artifact* component are implemented using the Java language. The user should be aware of regular camel routes and how to define endpoints and their respective parameters.

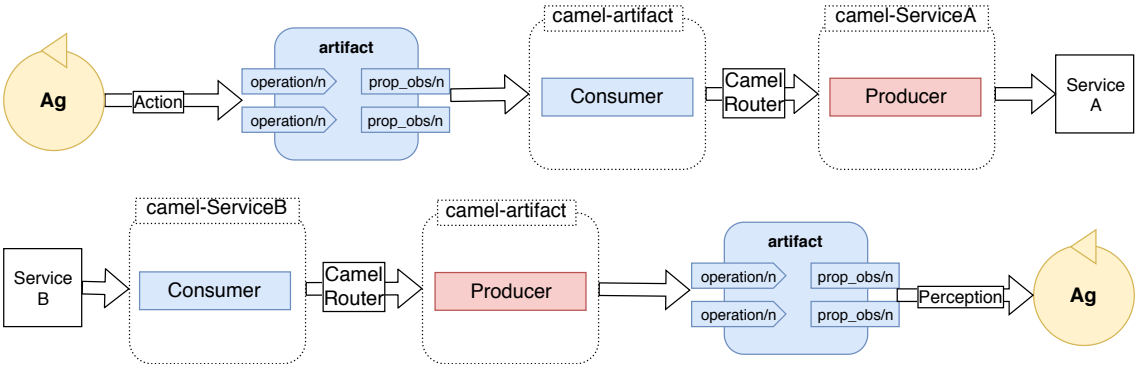


**Figure 4. Communication flow using *camel-artifact* component.**

On the top communication flow showed in Figure 4, we have an artifact integrated with some service A, an external non-autonomous entity. The interaction between them is as follows: (i) the artifact sends a message to to *Service A*, through specific methods provided by *camel-artifact*; (ii) the message, in form of an operation request, is consumed by *camel-artifact* component consumer which generates a camel standardised message to be sent to the external entity; (iii) the message is exchanged to the other side of the route, possibly being transformed; and (iv) the message is processed by *Service A* compatible component producer which prepares a compliance final message, with the proper format

and structure, which will be sent to some network address to be effectively consumed by the *Service A*.

In the other way around, on the bottom of Figure 4, we have: (i) *Service B* sends some data through the network reaching *Service B* component consumer by its network address; (ii) the message is exchanged through Camel route, possibly being transformed; (iii) the message is processed by *camel-artifact* producer which generates an artifact operation request; and (iv) the recipient artifact effectively consumes the operation request executing the referred method.

## 4. Illustrative application

For a better understanding of how the *camel-jason* and *camel-artifact* components can be used, we will resume the example of Industry 4.0, presented in Figure 1, and will build an implementation of this system.

The Figure 5 shows the MAS fully designed, with agents, external entities and the camel components used to implement the integration. These components are represented in the middle layer as artifacts and dummy agents. This hypothetical scenario implements an MAS to integrate the production and distribution stages of a product. The whole course can be divided in five stages: (i) a Programmable Logic Controller (PLC) finishes the product manufacturing, (ii) the information about the product is uploaded to an Enterprise Resource Planning (ERP) software, (iii) a research starts in order to contract the best freight company, (iv) the hired company starts transporting the product, providing its tracking information, and (v) warns the client via chat when it is near the final destination. The MAS is designed to unify those stages and to be responsible for managing each process. Moreover, Camel components are used as middleware between the MAS and external entities to integrate them.

One common question when designing the MAS is how many agents should be used. This is not mandatory, but a natural thought is to divide the process into sections and designate a single agent to be responsible for each part. In this case, we will consider that PLC and ERP stages represent the production part of the process, so one agent, named *production_agent*, will be responsible for managing these processes. Next, there is the hiring stage, that comprehends searching and hiring the best delivery company, for which the *distribution_agent* will be designated. The final stage could be thought as the delivery process, where the last agent, named *delivery_agent*, will be responsible for consulting the tracking information and sending the message to the costumer.

Another part of the designing process is the identification of which model, agent or artifact, is more suitable to represent each external entity. A common way to decide is observing its nature, i.e. autonomous or non-autonomous. Following this idea, it could be decided that the PLC and ERP software would be modeled as artifacts, since they are non-autonomous entities; and the customer as an agent, an autonomous entity. Another possibility to decide is looking at which type of communication the agents will perform with each external entity, i.e. via message exchanging or perception-action. For example, the action of hiring the delivery company, informing the company about some new contract via email, seems natural to be modeled as an A-A interaction. For this situation, the best suitable performative for the message is *tell*, since the agent is telling the supplier about a new hired delivery. On the other hand, when the *distribution_agent* is searching

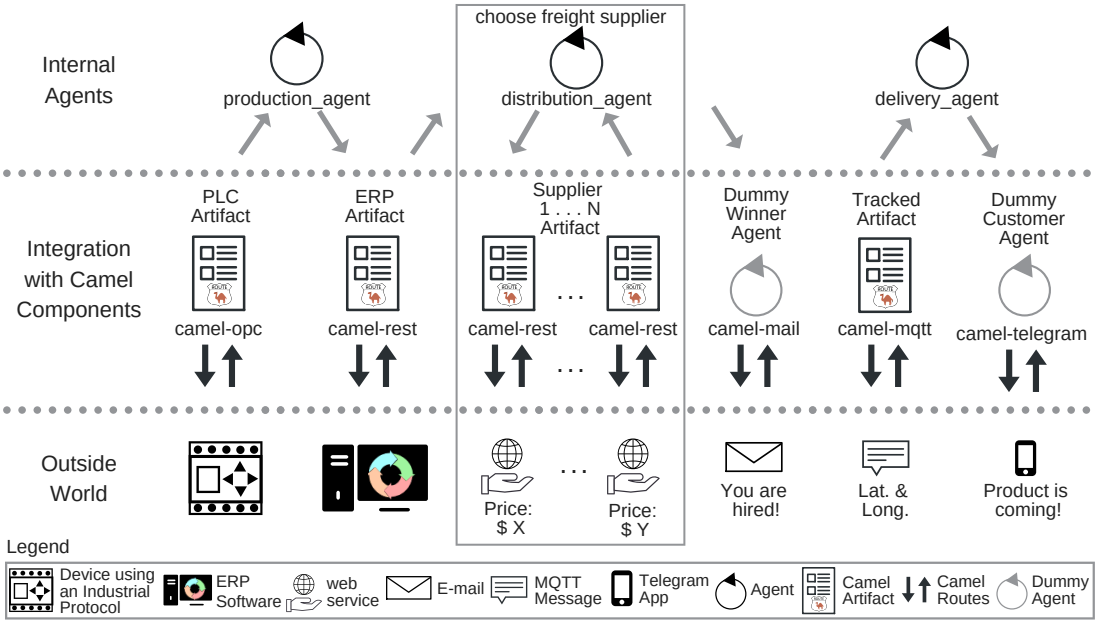


**Figure 5. An industrial process illustrating the integration of an MAS with a manufacturing device, updating an enterprise management software, choosing a supplier for delivering a tracked product to the customer.**

for the best delivery company, consulting their prices and conditions, it seems more suitable for this information to be perceived, like people do in a websearch. The same idea can be used when the *delivery_agent* tracks the position of the product, the information again is perceived by the agent, like looking at a screen. If message exchange was used in this case the agent would be flooded with unnecessary messages.

In fact, both interpretations, i.e., perception-action vs message exchanging and autonomous vs non-autonomous, could point to the same conclusion. This statement can be tested in the integration between the *delivery_agent* and the customer. The chat is done by message exchanging and it is performed by autonomous entities. Therefore, both interpretation reinforces the idea that the costumer should be modeled as an agent.

With the external entities modeled as MAS elements we could use *camel-artifact* and *camel-jason* to integrate the internal agents with the artifacts and agents, respectively. It is worth commenting that the external entities could be easily exchanged, for instance making the auction via email instead of using a web service. The interaction via email suggests modelling the participants as agents and use the *camel-jason* component for the integration.

Now the camel routes can be developed, depending on which type of technology the external entities use. As explained before, Camel have more than two hundred end-points available. In this example, OPC-DA, Rest, email, MQTT and Telegram end-points are being used to create the routes.

The code, in XML, for the route from the *delivery_agent* to Telegram can be seen in Listing 1. The `from` tag signalises the consumer part of the route, and `to` signalises the producer. In this case, the consumer address is the name of the dummy agent to which the message had been sent (in this example, `customer`), and the producer address is the

authorisation token followed by the `chatId` option.

```
<route>
  <from uri="jason:DummyCustomerAgent"/>
  <to uri="telegram:bots/sometoken?chatId=-364531"/>
</route>
```

**Listing 1. Example of *camel-jason* route definition**

In this example, the *delivery_agent* also uses the *camel-artifact* in order to obtain the delivery's position from a MQTT server. The route, in Java, is shown in Listing 2. When the position is published on the topic of interest (`latLong`) the route redirects it to the artifact by specifying its name (`TrackedArtifact`) and the operation (`giveDistance`) as headers. Here, we are assuming that the calculations will be done by the artifact, but they could be done in the route through a transformation, before sending to the artifact.

```
from( "mqtt : foo? host=tcp://broker & subscribeTopicName=latLong" )
  .setHeader( "ArtifactName" , constant ( "TrackedArtifact" ) )
  .setHeader( "OperationName" , constant ( "giveDistance" ) )
.to( "artifact : cartago" );
```

**Listing 2. Example of *camel-artifact* route definition**

## 5. Related research

In this section, we went over works that have addressed agent technology in an integrating context. Maturana and Norrie [Maturana and Norrie 1996] have proposed a mediation and coordination tool for MAS. They have used mediator agents as manufacturing coordinators. Following similar idea, Olaru et al. [Olaru et al. 2013] have developed an agent-based middleware, which creates a sub-layer of application layer that allows agents to mediate context-aware exchange of information among entities. We think an autonomous entity as middleware may increase complexity and compromise performance. Instead of creating some kind of hierarchy, our approach gives connectivity power to MAS entities.

Leading industrial suppliers are also providing solutions using agents such as the Agent Development Environment (ADE), designed by Rockwell Automation [Tichý et al. 2012]. It provides connectivity with common shop floor devices and supports the development of agents. The limitation we have seen regards especially connectivity with all sorts of entities (e.g. IoT sensors and mobile devices, ERP and other software etc), which in our case is provided by Camel.

Other research address the combination of MAS and Service-Oriented Architecture (SOA). One way to achieve this merge is based on the creation of a proxy function to provide interoperability between MAS and SOA, as found in [Nguyen and Kowalczyk 2005, Shafiq et al. 2005, Greenwood and Calisti 2004, Fayçal et al. 2010]. Another way is by implementing services as agents as we found in [Mendes et al. 2009, Tapia et al. 2009, Carrascosa et al. 2009, Argente et al. 2011]. The approaches using SOA are more mature to be applied in practice. The ones that *agentified* the services have also the advantage to use MAS background, i.e., using ACL messages they are able to use interaction protocols. In these studies integration is usually done through specific APIs and they lack differentiation over autonomous and non-

autonoumous entities, and interoperability with heterogeneous entities, both aspects increase development complexity.

Vrba et al. [Vrba et al. 2014] propose a gateway for wrapping an MAS as a service to be used as a loosely coupled software component into the Enterprise Service Bus (ESB). This gateway transforms agent messages to ESB messages and vice versa, enabling communication between agents and ESB services. This solution is closely to ours, only lacking support to the A-E approach, where interaction is based on agents perceiving and acting upon artifacts in the environment. The indiscriminately *agentification* may increase complexity and affect performance.

Cranefield and Ranathunga [Cranefield and Ranathunga 2013] developed a *camel-agent* component for *Jason* agents. It is very similar to our developed *camel-jason* component. Essentially, the difference is that we are embedding Apache Camel since our component works as an infrastructure, being transparent to the agents. In their work, Jason was actually embedded in an Apache Camel project where agents were smoothly placed in containers.

## 6. Conclusion

In this paper, we introduced two Camel components aiming the integration of MAS with external entities: *camel-jason* and *camel-artifact*. The former integrates agents with external entities modelled as agents. The latter integrates agents and external entities modelled as artifacts. The decision of which component to adopt for each entity depends on the characteristics of the external entity and the MAS developer *can* choose the most suitable component. For instance, he/she is not obliged to "agentify" every external entity, even those that do not have agent properties.

The two components introduced in this paper, along with the communication infrastructure provided by Camel and its existing components, makes the integration between MAS and different entities simpler. Issues related to interoperability, routing, and data transformations are partially solved in the camel routes. Another advantage of using such components is that the agent program does not need to deal with integration issues. Agents continue to interact only with another agents and artifacts.

Finally, this is an ongoing work. In a future step we intend to compare our approach with related works and to evaluate other aspects of using the developed Camel components to integrate MAS and external entities, such as the impact on the performance, security, openness, scalability, among others.

## References

Argente, E., Botti, V., Carrascosa, C., Giret, A., Julian, V., and Rebollo, M. (2011). An abstract architecture for virtual organizations: The thomas approach. *Knowledge and Information Systems*, 29(2):379–403.

Bellifemine, F., Bergenti, F., Caire, G., and Poggi, A. (2005). *Jade — A Java Agent Development Framework*, pages 125–147. Springer US, Boston, MA.

Boissier, O., Bordini, R. H., Hübner, J. F., and Ricci, A. (2019). Dimensions in programming multi-agent systems. *The Knowledge Engineering Review*, 34:e2.

Carrascosa, C., Giret, A., Julian, V., Rebollo, M., Argente, E., and Botti, V. (2009). Service oriented mas: an open architecture. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 1291–1292. International Foundation for Autonomous Agents and Multiagent Systems.

Cranefield, S. and Ranathunga, S. (2013). Embedding Agents in Business Processes Using Enterprise Integration Patterns. pages 97–116.

Fayçal, H., Habiba, D., and Hakima, M. (2010). Integrating legacy systems in a SOA using an agent based approach for information system agility. *2010 International Conference on Machine and Web Intelligence, ICMWI 2010 - Proceedings*, pages 338–343.

Greenwood, D. and Calisti, M. (2004). Engineering web service-agent integration. In *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No. 04CH37583)*, volume 2, pages 1918–1925. IEEE.

Hohpe, G. and Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman, USA.

Ibsen, C. and Anstey, J. (2010). *Camel in Action*. Manning Publications, USA, 1st edition.

Maturana, F. P. and Norrie, D. H. (1996). Multi-agent mediator architecture for distributed manufacturing. *Journal of Intelligent Manufacturing*.

Mendes, M., Electric, S., Restivo, F., Colombo, A. W., and Electric, S. (2009). Service-oriented Agents for Collaborative Industrial Automation and Production Systems. 2744(August).

Nguyen, X. T. and Kowalczyk, R. (2005). Enabling agent-based management of web services with WS2JADE. *Proceedings - International Conference on Quality Software*, 2005:407–412.

Olaru, A., Florea, A. M., and El Fallah Seghrouchni, A. (2013). A context-aware multi-agent system as a middleware for ambient intelligence. *Mobile Networks and Applications*, 18(3):429–443.

Omicini, A., Ricci, A., and Viroli, M. (2008). Artifacts in the A&A meta-model for multi-agent systems. *Journal of Autonomous Agents and Multi-Agent Systems*, 17(3):432–456.

Ricci, A., Viroli, M., and Omicini, A. (2006). Programming MAS with artifacts. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3862 LNAI:206–221.

Roloff, M., Amaral, C., Stivanello, M., and Stemmer, M. (2016). Mas4ssp: A multi-agent reference architecture for the configuration and monitoring of small series production lines. In *INDUSCON*.

Shafiq, M. O., Ali, A., Ahmad, H. F., and Suguri, H. (2005). Agentweb gateway-a middleware for dynamic integration of multi agent system and web services framework. In *14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE'05)*, pages 267–268. IEEE.

Tapia, D. I., Rodríguez, S., Bajo, J., and Corchado, J. M. (2009). Fusion@, a soa-based multi-agent architecture. In *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, pages 99–107. Springer.

Tichý, P., Kadera, P., Staron, R. J., Vrba, P., and Mařík, V. (2012). Multi-agent system design and integration via agent development environment. *Engineering Applications of Artificial Intelligence*.

Vieira, R., Wooldridge, M., and Bordini, R. H. (2007). On the Formal Semantics of Speech-Act Based Communication in an Agent-Oriented Programming Language. 29:221–267.

Vrba, P., Fuksa, M., and Klima, M. (2014). JADE-JBossESB Gateway: Integration of Multi-Agent System with Enterprise Service Bus. *2014 Ieee International Conference on Systems, Man and Cybernetics (Smc)*, pages 3663–3668.