Limitations and Divergences in Approaches for Agent-Oriented Modelling and Programming

Artur Freitas, Rafael C. Cardoso, Renata Vieira, and Rafael H. Bordini

Postgraduate Programme in Computer Science, School of Informatics (FACIN)
Pontifical Catholic University of Rio Grande do Sul (PUCRS). Porto Alegre, RS - Brazil
{artur.freitas, rafael.caue}@acad.pucrs.br,
{renata.vieira,rafael.bordini}@pucrs.br

Abstract. This paper shares our experiences in applying two well-known methodologies that play different roles in agent-oriented software engineering: modelling with Prometheus and programming with JaCaMo. First, we modelled a realistic multi-agent scenario using Prometheus to support its design and specification. Afterwards, JaCaMo was used as the development platform for programming our case study. Then, we were able to compare the outcome of combining these approaches, allowing us to identify some gaps, limitations and conceptual divergences when such approaches are used together to engineer a complex case study. This empirical study is our basis for discussing the lessons learned and for showing the theoretical and practical aspects of applying these two traditional agent-based approaches. Therefore, this paper highlights advantages and drawbacks of using Prometheus and JaCaMo to design and implement a complex multi-agent scenario, and how suitable is their integration for improving agent-oriented software engineering.

Keywords: Agent-oriented software engineering, Prometheus, JaCaMo

1 Introduction

Producing software code for complex and highly detailed systems directly in programming environments without first using any specification, modelling or design mechanism can cause many problems. For example, without a proper modelling of the system's environment it can be difficult to find potential bugs when they eventually appear in the implementation, since they might be bugs that were introduced during the design and specification of the system. These problems can cause further delay in the development of complex MAS, which are already notoriously hard to debug, mostly due to the lack of debugging support in MAS development platforms. Originally, methodologies for Agent-Oriented Software Engineering (AOSE) were aimed at agent-oriented programming languages that were mostly concerned with programming individual agents. Since then, programming abstractions covering the social and environmental dimensions of Multi-Agent Systems (MAS) have emerged [2], and are resulting in new MAS development platforms with multiple levels of abstractions. AOSE methodologies though, are not following the same pace. Although traditional methodologies can be used with these new MAS development platforms, they may diverge in

several points due to their differences regarding abstraction levels, and, thus, it is important to be aware if such combination can be successfully used, and which are the advantages, limitations and consequences of doing so. Therefore, our work discusses how these AOSE approaches can complement each other and which are the implications of such combination, rather than using each one of them in isolation.

Agent-oriented methodologies and programming languages for engineering MAS are often disjointed, resulting in limitations, gaps and conceptual divergences between the modelling and programming phases. In this paper, we apply two agent technologies in a complex case study in order to comparatively investigate the modelling and programming approaches for MAS. The case study allows us to point out and discuss the challenges we faced during this process, as well as possible ways to overcome them. Our case study is the Multi-Agent Programming Contest of 2016[1], an annual competition carried out as an attempt to stimulate research in the area of MAS development and programming. The performance of a particular system is determined through a series of simulation rounds, where systems compete against each other. This year the scenario consists of solving logistics problems in the realistic streets of cities, by buying, building, and delivering goods. First, Prometheus [6] is employed to specify our models, which are, then, used to code the system in the JaCaMo [1] programming framework. In other words, this work investigates the suitability of Prometheus for the specification of agent systems considering that the codification will take place in JaCaMo. In such context, it is important to be aware of divergences when these approaches are used in the development of complex MAS. Therefore, on the light of our case study, we analyse and discuss differences and problems (with possible solutions) between the combined use of the Prometheus methodology with the JaCaMo framework. As result, we point out lessons learned from using these technologies in combination.

This paper is structured as follows. Section 2 explains the basic concepts of the two agent-oriented modelling and programming approaches investigated in this paper: Prometheus and JaCaMo. Then, we describe the scenario and show our Prometheus models and pieces of JaCaMo code in Section 3, which is followed with a discussion about limitations, conceptual gaps, and possible solutions. Finally, Section 4 discusses related work, and our final remarks are given in Section 5, along with an outline of future research directions.

2 Background

The Prometheus methodology has been developed for over a number of years as a result of being used in the industry [3]. While Prometheus [6] is a methodology for modelling intelligent agent systems, the Prometheus Design Tool (PDT) is a graphical tool that follows the Prometheus methodology in order to build the design of MAS [8]. PDT started as a stand alone tool, but it is nowadays being developed as a plug-in for Eclipse. Prometheus contains three phases: *system specification*, *architectural design*, and *detailed design*. The *system specification* focuses on identifying basic system functionalities, along with inputs (percepts), outputs (actions), and any important shared

[1] https://multiagentcontest.org/

data sources. This phase defines what the system is intended to do. The *architectural design* uses the outputs from the previous phase to determine the system's agents and how they will interact. Thus, it establishes the structure of the system being developed. The *detailed design* looks at the internals of each agent and how it will accomplish its tasks within the overall system, that is, it establishes the plans that the agents require in order to achieve their goals.

JaCaMo [1] combines three separate technologies into a framework for MAS programming that makes use of multiple levels of abstractions, enabling the development of robust MAS. Each technology (Jason, CArtAgO, and Moise) was developed separately for a number of years and are fairly established on their own when dealing with their respective abstraction level. Jason is responsible for the agent level, it is an extension of the AgentSpeak language. Based on the BDI (Belief-Desire-Intention) model, agents in Jason react to events in the system by executing actions on the environment, according to the plans available in each agent's plan library. CArtAgO is based on the A&A (Agents and Artefacts) model, and deals with the environment level. Artefacts are used to represent the environment, storing information about the environment as observable properties and providing actions that can be executed through operations. Agents focused on artefacts can obtain percepts from them and execute operations on the artefacts. Moise handles the organisation level, enabling an explicit specification of the organisation in a MAS. This level adds new elements to the MAS, such as roles, groups, organisational goals, missions, and norms. Agents can adopt roles and compose groups. Missions are defined to achieve organisation goals, and the behaviour of the agents that adopt roles to execute these missions is guided by norms.

3 Modelling and Implementation of the Case Study

Our work uses PDT, a plugin for the Eclipse platform that adopts the graphical notation depicted in Fig. 1. This image identifies the symbols used in PDT's graphical models, serving as a notation for many of the models presented throughout this paper. Our case study is based on the multi-agent programming contest scenario of 2016. In this scenario two teams of autonomous vehicles are controlled by agents, competing against each other in order to accomplish logistic tasks in the streets of a realistic city. Completing tasks rewards the team with money, and the team with the most money by the end of the simulation is the winner. Tasks can be created by the environment or by one of the agent teams. A task can require the acquisition, assembling, and transportation of goods. In our simulations teams have four agents, with each agent having a different type. These types define the agent's speed, if they move by air or land, battery charge, maximum load, and what tools they can use. The types of agents are: car, drone, motorcycle, and truck. The map contains facilities of several types such as shops, warehouses, charging stations and storage facilities. Items can be bought, crafted, given to a teammate, stored, delivered as part of a job completion, recovered from a storage facility, and dumped. These action may only happen at their respective locations/facilities. Some overall characteristics assumed for our case study are given next. Each team is composed of only collaborative agents. The two teams are competing with each other to win money. The strategies for opponent teams are unknown (since we develop models

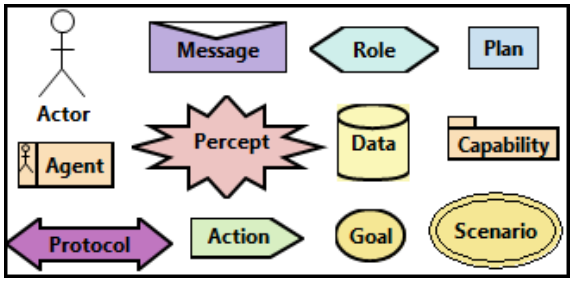


Fig. 1. Graphical notation of the Prometheus elements (obtained from the PDT in Eclipse)

and codes that correspond only to our team). It is assumed for our agents that they are truthful, their communication is reliable and their knowledge is imperfect.

3.1 System Specific Design

In Prometheus, the specification usually starts with the **analysis overview** diagram, which identifies `actors`, `percepts`, `actions`, and `scenarios` [6, 3]. `Actors` are external entities that will use or interact in some way with the system. Actors can be other software systems or humans. `Percepts` are the inputs to each `scenario`. `Actions` are produced by the system for each `scenario`. `Scenarios` describe the interaction and correspond to the main functionalities of the system. A `scenario` is a sequence of structured steps where each step can be one of: `percept`, `action`, `goal`, or (sub)`scenario`. `Goals` are defined as desires of `agents` and may trigger the execution of `plans`.

Figure 2 shows the analysis overview diagram. When mapping Prometheus to JaCaMo, Actors correspond to the artefacts of CArtAgO, and Actions can be seen as the operations of artefacts. The `round procedure` scenario, which represents each round of the simulation, models that agents interact with a `server` actor by receiving the `round start` and `round end` perceptions. A round consists of a series of steps in which agents receive the `request action` perception and then send an `action` to the actor. We used a contract net protocol mechanism, based on the original design of Smith [7], to distribute tasks among agents, as shown in the `task announcement procedure`, `bid procedure`, and `award procedure` scenarios in Fig. 2. The contract net was modelled using artefacts to control and mediate communication among agents by taking advantage of the shared resources available in CArtAgO, instead of using the usual method of message passing, and thus, improving performance during execution. Since agents have a deadline to send an action during each step, performance is an important feature in the strategy of any team. The `task board` is where tasks are announced. Each announced task has a respective `contract net board`, where agents make bids and the task is awarded to the best bid.

We observe the following limitations so far. In our case study, what is modelled as Actors in Prometheus is translated to Artefacts in JaCaMo. However, there is not any visual way to represent that an Actor, an Agent, or a Scenario instantiates an Actor. That is, none of these elements can connect directly with an Actor. In CArtAgO, artefacts can be created by: (i) agents; (ii) other artefacts; or (iii) when the MAS starts

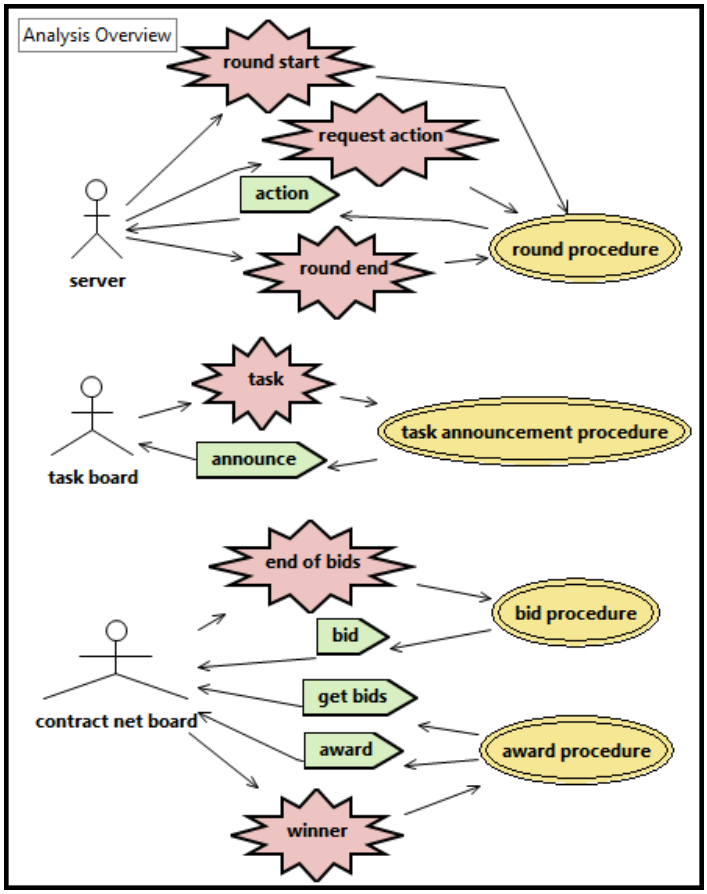


Fig. 2. Analysis overview diagram

its execution. We did not found any way to represent these things without having to change the meta-model of Prometheus and/or PDT. When using CArtAgO, both agents and artefacts can execute operations (i.e., actions). However, it is not possible to represent that an actor can execute an action in the Prometheus analysis overview diagram (e.g., a connection from an actor to an action). This diagram allows us only to represent that agents can execute actions, and that actors receive actions. To overcome this limitation, a scenario must be created between these elements, so that the actor connects with the scenario and the scenario connects with the action. Relations between two actors cannot be represented, thus, we cannot specify hierarchies among artefacts (e.g., to define inheritances/specialisations of artefacts/actors). The same limitation is true for hierarchies of agents, and hierarchies of roles. Another limitation is that agents cannot connect directly with actors, so there is no way of establishing that an agent can create an actor, or vice-versa. In CArtAgO, agents can only obtain perceptions and execute operations over artefacts after successfully focusing on them, and agents can only focus on artefacts if they are on the same workspace (localities where artefacts are situated). We did not found out how to represent these access restrictions.

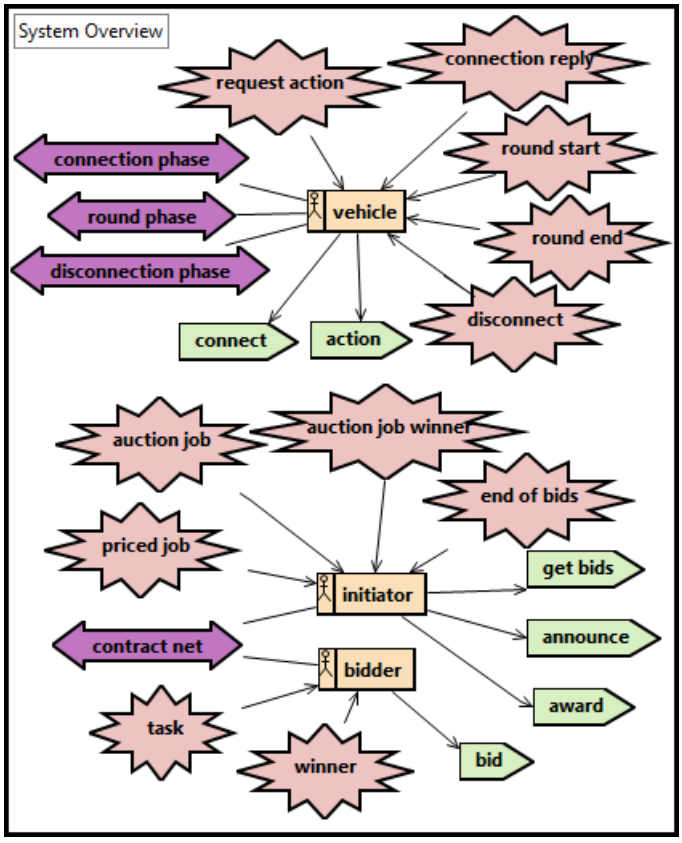


Fig. 3. System overview diagram

3.2 Architectural Design

The **system overview** diagram ties together the agents, events, and shared data objects [6]. `Agents` perform `actions`, receive `percepts`, bring together `roles`, exchange `messages`, and participate in `protocols`. `Protocols` define interactions between `agents` in terms of the allowable sequences of `messages` passed between them and the interactions with things outside the system (`actors`). `Messages` are sent and received by `agents` in order to accomplish the various aims of the system. `Roles` are intended as relatively small and easily specified chunks of `agent` functionality for grouping `goals` into cohesive units.

The system overview diagram, as shown in Fig. 3, provides a general understanding of how the system as a whole will function, and adds agents and protocols to the models. It also relates them with the previously described actions and percepts, and adds new ones if required. In our case, we have `vehicle`, `initiator` and `bidder` as agents. The elements introduced here are propagated and detailed better in further phases of Prometheus. Protocols and actors cannot be visually connected in any diagram (except in the sequence diagrams, as shown in Fig. 4 and Fig. 5). When we define that an agent participates in a protocol, this automatically propagates to the system overview diagram.

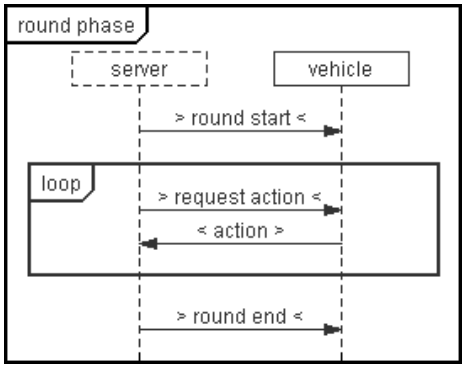


Fig. 4. Sequence diagrams of the `round phase` protocol

The same does not happen when it is defined that an actor participates in a protocol (for example, such relation does not propagate to the `analysis overview` diagram). It is not possible to define what triggers or initialises a protocol, as this information cannot be included in any diagram. The very first thing in the protocol indicates that it has begun, but it does not indicate the reason that made it start.

Percepts, actions, and actors can participate in protocol specifications, in addition to messages and agents. In protocols, percepts (represented by >perception_name<) originate from actors (represented by dashed rectangles) and go to agents (represented by solid rectangles); and actions (represented by <action_name>) always originate from an agent and go to an actor. Figure 4 depicts the sequence diagram for the `round phase` protocol and Fig. 5 illustrates the `contract net` protocol. Our contract net protocol starts when an `initiator` agent performs an `announce` action in a `task board` artefact. Then, the `task board` creates a `contract net board` artefact, however Prometheus and PDT do not allow for such information to be included in the diagram, and thus, it is not visually represented. Then, the `contract net board` produces a `task` percept for `bidders` that will `bid` for it. The `contract net board` controls the `deadline` and sends the `end of bids` percept to the `initiator`, which will `get bids` and `award` the best offer (`deadline` is achieved when all agents that are able to bid do so, or when a timeout is reached). In the end, the `bidder` receives a `winner` percept indicating who won the task.

As we previously commented, it is not possible to connect two actors in the sequence diagram. This is needed to define, for example, that an actor creates another actor. In our case, `task boards` make instances of `contract net boards` when an `initiator` announces a task. We already discussed the representation of actors' instantiations when analysing limitations in the analysis overview diagram, and the same observations apply to the sequence diagrams. Figure 6 shows (some parts of) the `contract net board` CArtAgO artefact code. We made adaptations from the contract net protocol artefacts that come with the CArtAgO package in order to use it in our implementation. This artefact corresponds to the `actor` represented in Prometheus using the same nomenclature. Its operations (`bid` and `getBids`) are traced to the `actions` in Prometheus that previously appeared in several diagrams (Figures 2, 3, and 5). The `bid` operation adds the bid received as parameter in a bid list if the state

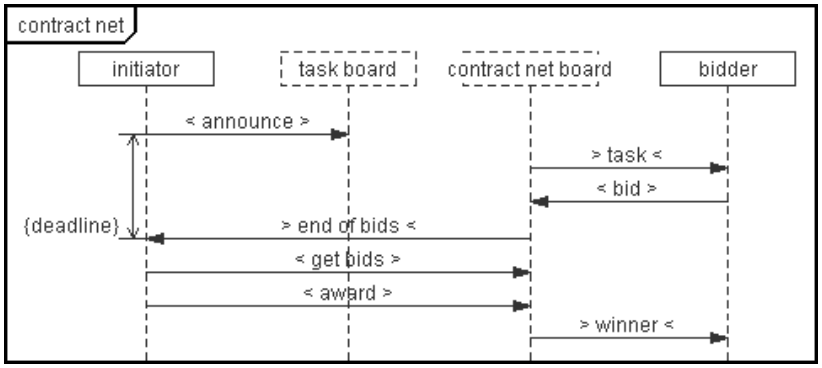


Fig. 5. Sequence diagrams of the `contract net` protocol

of this `contract net board` is `open`. The `getBids` operation returns the bid list after passing the guard condition that checks if the bidding window is `closed`. Moreover, the observable properties of this CArtAgO artefact (defined in the `init` method) are related to the `percepts` of Prometheus' models. Some parts of the code are omitted for simplicity and due to the lack of space. One example is the Bid class (appears on lines 2, 13, 22), which is used to manage the bids and contains as attributes an identification number and a value.

Our **data coupling overview** diagram is depicted in Figure 7. In Prometheus, the data coupling diagram contains the `roles` (functionalities) and their relations to all identified `data` (not only persistent data, but also data required by the functionalities). Therefore, in our case study, we found that the data can come from two different sources: (i) from the environment (more specifically, the artefacts); or (ii) in the form of beliefs. In both cases, we found that, in this context, Prometheus presented conceptual divergences regarding the data representation adopted in JaCaMo.

3.3 Detailed Design

The detailed design phase consists of a list of agent overview diagrams, one for each agent has their own capability overview diagrams, one for each capability included in the agent [3]. The **agent overview** diagram provides the top level view of the agent internals [6]. It is similar in style to the system overview diagram, but instead of agents within a system, it shows capabilities within an agent. Then, the **capability overview** diagram goes even further, describing the internals of a single capability. At the bottom level these will contain plans, with events providing the connections between plans, just as they do between capabilities and agents. These diagrams are similar in style to the system and agent overview diagrams, although plans are constrained to have a single incoming (triggering) event.

`Capabilities` of `agents` are defined in terms of `plans`, `events`, and `data` [6]. `Capabilities` are described by a capability descriptor which contains information about its external interface – the events that are used as inputs, and the events that are produced as inputs. The `capabilities` of an `agent` usually (at least initially) correspond to the `roles` that were assigned to it, though `roles` may also be split into

```
1 public class ContractNetBoard extends Artifact {
2     private List<Bid> bids = new ArrayList<Bid>();
3     private int bidId = 0;
4
5     void init(String taskDescr, long duration){
6         defineObsProperty("task_description", taskDescr);
7         defineObsProperty("deadline", duration);
8         defineObsProperty("state","open");
9         // ...
10    }
11    @OPERATION void bid(int bid, OpFeedbackParam<Integer> id){
12        if (getObsProperty("state").stringValue().equals("open")){
13            bids.add(new Bid(++bidId,bid));
14            id.set(bidId);
15        } else
16            failed("cnp_closed");
17    }
18    @OPERATION void getBids(OpFeedbackParam<Literal[]> bidList){
19        await("biddingClosed");
20        int i = 0;
21        Literal[] aux = new Literal[bids.size()];
22        for (Bid p: bids)
23            aux[i++] = p.parseBid();
24        bidList.set(aux);
25    }
26    // ...
27 }
```

Fig. 6. CArtAgO code for the `contract net board` artefact

multiple smaller `capabilities` or merged into a larger one. `Plans` are procedures that can be triggered by events such as the desire to achieve a `goal` or the belief of perceiving something. The concept of `Data` in Prometheus allows representation of domain information and entities that are outside of the agent paradigm [3].

Figure 8 shows a snippet of the agent overview diagram for `initiator` agents. This diagram shows the following 5 capabilities: *priced job analysis*, *auction job analysis*, *auction job winner verification*, *announcement procedure*, and *award procedure*. The `priced` or `auction job` percept triggers its corresponding `analysis` capability, which uses as data `map information` and/or `vehicle information`. When our `initiator` decides that a priced job will be taken, it creates the goal of `separate tasks`, which will be the input of the `announcement procedure`. When it is decided that an auction job is worth to take, the path is a little longer, because we have to make a bid to the server, and our team may not be the bid winner. So, we make a bid and save it, until we get a percept from the server of who is the `auction job winner`. If we verify that we won, then we can add the goal of `separate tasks`. This goal starts the `announcement procedure`, which divulges the task to our bidder agents. The next and final step is the `award procedure`, which collects the bids and `award` one of our agent to execute the desired task.

Figure 9 shows a snippet of the agent overview diagram for `bidder` agents, focusing on the `bid procedure` and its corresponding capability overview diagram. The `bid procedure` is responsible to analyse the context and calculate a bid for a task. Thus, it receives as input the percept of a `task`, and data about the map and vehicles;

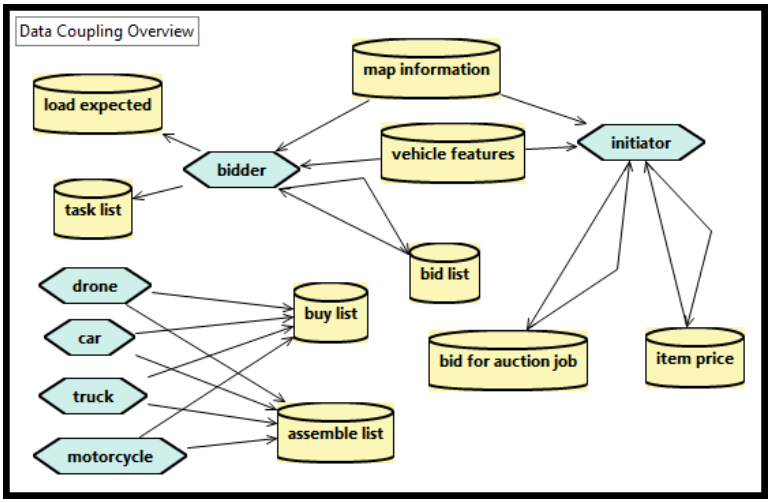


Fig. 7. Data coupling overview diagram

and produces as output the `bid` action, storing its value as a belief. The `focus` action happens on a `contract net board` artefact, which receives the `bid` action. Among the `vehicle features` data, it is the vehicle speed, load capacity, battery, tools, and so on. Figure 9 also shows code in JaCaMo which corresponds to the capability overview diagram of `bid procedure`. It shows two plans in JaCaMo (`+task` and `+!calculate_bid`) that correspond to the two plans in PDT. The icons of plans in PDT are directly converted to plans in Jason with their corresponding triggering events and plan bodies composed of actions, goals and data manipulations.

Figure 10 illustrates the capabilities and plans of vehicles agents, such as, for example, the `find items` and `get items` capabilities. The plan to `find items` build a list of items to buy and a list of items to assemble. Then, the plan to `get items` uses such lists as inputs to generate goals corresponding to the actions of `goto`, `buy` and `assemble`. Figure 10 shows Jason code for the `vehicle` agent, with the `plans` to achieve the `goals` of `find items` and `get items`. These code is traced to the elements represented in the Prometheus diagram illustrated in Figure 10. The `find items` plan has queries to discover which products must be bought and which are the best shops and workshops for that vehicle to acquire its desired resources. Some items cannot be bought directly in shops, but can only be obtained from assembling some specific items in a workshop. Thus, the `find items` creates beliefs about which items should be bought in each shop, and which workshops should be visited in order to assemble the composite items. The plan to `get items` creates the goals of `go to` each of these locations, `buy` the items and make the desired `assembles`.

Visually, the diagrams lack some graphical notion of ordering among the elements in a plan in the agent overview and capability overview diagrams. Plans are usually defined as an orderly or step-by-step conception or proposal for accomplishing an objective. A visual notation that could represent such things would be very interesting for modelling plans of MAS in the diagrams of Prometheus. In Moise, plans are composed as a list of sub-goals and an operator that specifies *sequence*, *choice* or *parallelism*.

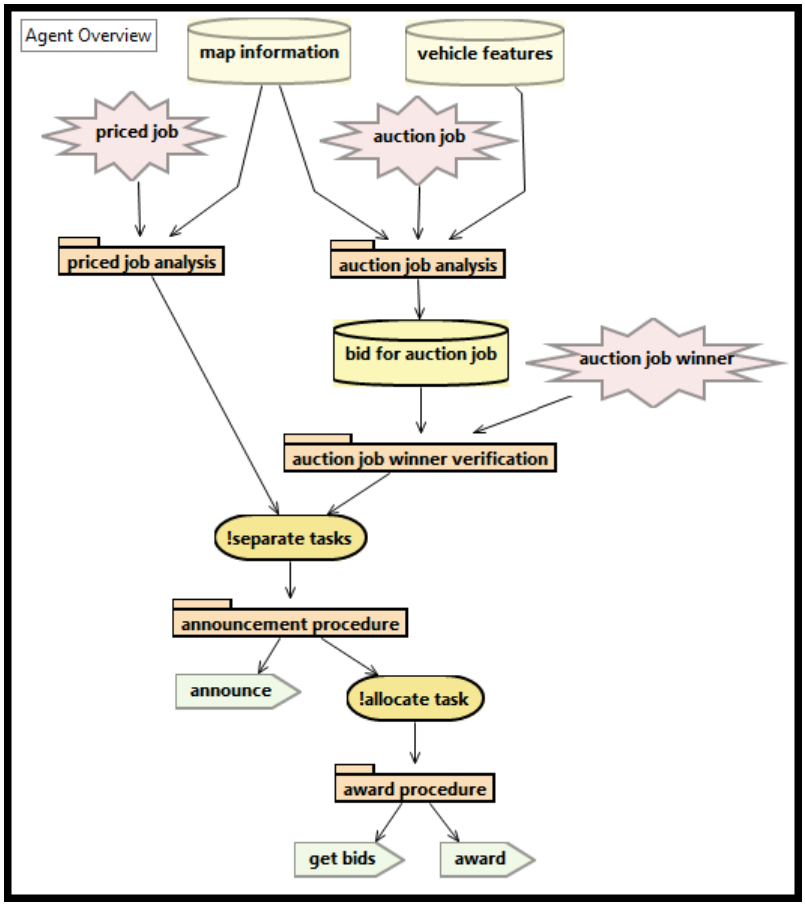


Fig. 8. Agent overview diagram for `initiator` agents

Thus, there is a mismatch between Prometheus and Moise where information would be lost or missing when converting from one specification to another.

We found confusing the fact that it is possible to place plans in the agent overview diagram. Prometheus works with the concept of capability, and each capability must be implemented through at least one plan in its corresponding capability overview diagram. Then, if a plan is placed in the agent overview diagram, it does not fit neither contribute to any capability that the agent is supposed to have.

Some aspects that did not contribute significantly during the modelling or development of our case study are now discussed. The **scenario overview** diagram shows only one kind of element (scenarios) and they cannot be linked, which is not very useful from the viewpoint of graphical models. When clicking on a scenario, it is possible to define its properties in textual formats and the steps of the scenario in tables (not very visually friendly formats). Also, it is not possible to define properties or additional information in relationships among PDT elements. For example, consider that we could link roles, then we could say that they are disjoint, or that a role specialises another, and

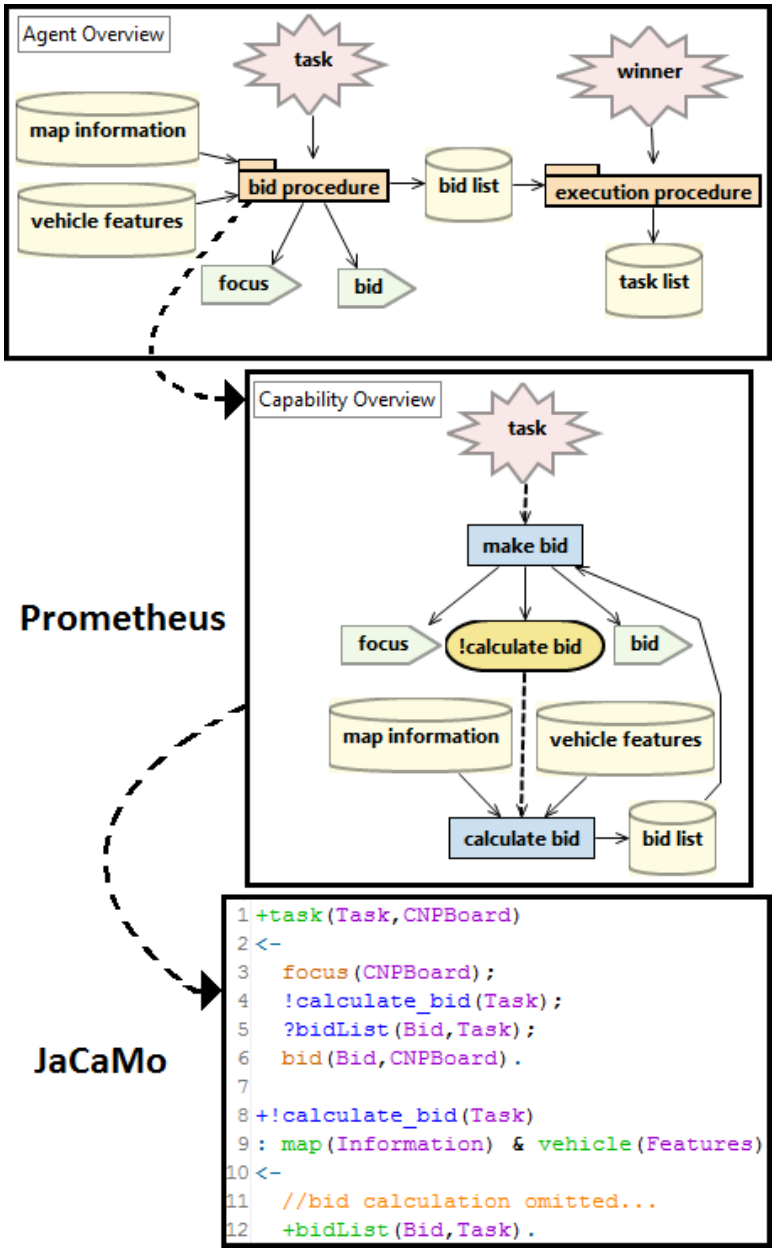
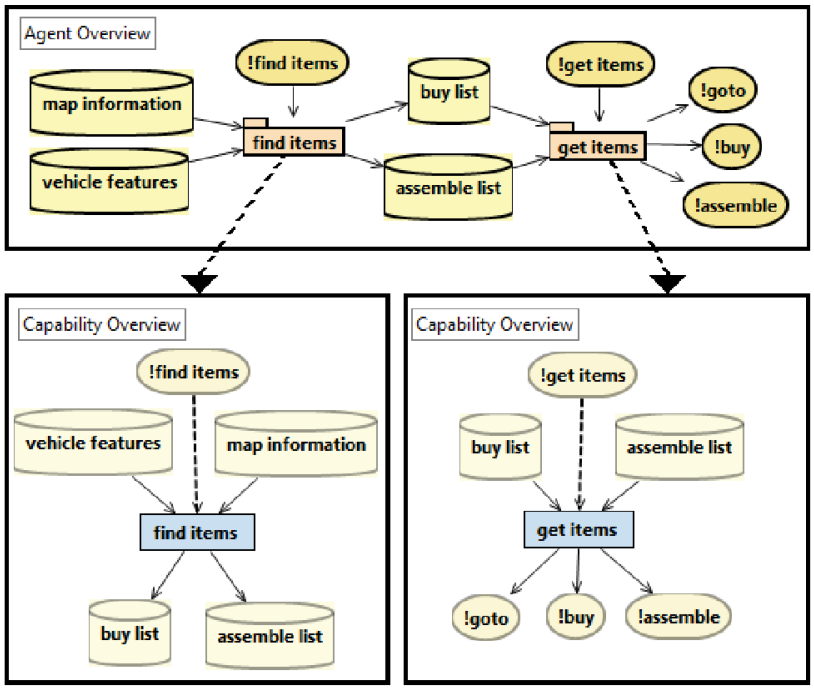


Fig. 9. Snippets of the agent overview diagram for `bidder` agents, capability overview diagram for `bid procedure`, and corresponding JaCaMo code (more specifically, Jason code)




```
1+!find_items
2     : working(JobId,ItemList,StorageId)
3<-
4     for ( .member(item(ItemId,Qty),ItemList) ) {
5         ?product(ItemId,Volume,BaseList);
6         if (BaseList == []) {
7             ?find_best_shop(ItemId,Shop);
8             +buyList(ItemId,Qty,Shop);
9         }   else {
10            ?decomp(BaseList,BaseItems);
11            for ( .member(item(BaseItemId,BaseQty),BaseItems) ) {
12                ?find_best_shop(BaseItemId,Shop);
13                +buyList(BaseItemId,BaseQty,Shop);
14            }
15            ?find_best_workshop(Workshop);
16            for ( .range(I, 1, Qty) ) {
17                +assembleList(ItemId,Workshop);
18            }
19        }
20    }.
21
22+!get_items
23<-
24    for ( buyList(_,_,Shop) ) {
25        !goto(Shop);
26        while ( buyList(Item,Qty,Shop) ) {
27            !buy(Item,Qty);
28        }
29    }
30    while ( assembleList(Component,Workshop) ) {
31        !goto(Workshop);
32        !assemble(Component);
33    }.
```

Fig. 10. Capabilities and plans of `vehicle` agents in Prometheus and corresponding Jason code

so on for other types of relationships. According to the papers of Prometheus [3], the steps of scenarios can assume only one of the following types: percept, action, goal or (sub)scenario. However, PDT allows roles to be steps of scenarios, and the meaning for this is not clear for us. We think that protocols should be able to be added as steps of a scenario and, currently, PDT does not allow for such thing.

The **goal overview** diagram addresses only goals. Since PDT creates a goal for each scenario, and the scenarios correspond to the main functionalities of the system [3], the goals are best viewed as system's use cases. However, later diagrams treats goals as triggering of plans, and then goals are better interpreted as individual desires of agents. We believe that these different views on goals can be confusing and their meaning should be defined more precisely. The BDIMessage addresses only goals, however messages could also transmit beliefs and plans (when considering JaCaMo). Some other diagrams, such as **system role**, **data coupling** and **agent role grouping overview**, did not add much information for our case study.

Plans can have only one triggering condition, and there is not a direct and simple graphical representation of the context for plans (i.e., predicates that must be true to consider a given plan applicable). Also, it would be interesting to represent conditions (semantically different from the plan trigger) that can lock or unlock the execution of a plan, such as the presence of a belief resulted from a percept of something (e.g., vehicles can have a plan that waits until a percept of request action comes from the server). Two entities cannot have the same name [3], for example a goal and a plan. This impossibility can result in more advantages than limitations, but in our case it was the opposite. For example, a goal can be the triggering condition of a plan in Jason, so we would like to use to same name to refer to these two different (but related) entities. Also, it is not possible to have references to non-existent entities, since creating a reference will create the entity if it does not exist and when an entity is deleted all references to it are deleted as well. The propagation of changes is, usually, a good thing, but can generate unexpected results when we would like to change something only in one place, or we are not aware about other changes that resulted from a single modification.

4 Related Work

Prometheus and PDT are used to develop the design of a conference management system case study [5]. This work pointed out the importance of integrating Prometheus with other agent software design tools and methodologies. Then, the conference management system case study was also modelled with O-MaSE and Tropos in another paper [3], which compared Prometheus with its alternatives based on such example. We differ from these papers since, besides modelling a more complex scenario, we focus on pointing out limitations and divergences of Prometheus with regards to JaCaMo. Thus, rather than showing cases that could be successfully modelled, this paper highlights and discusses situations that were impossible to model or that turn out to be more confusing than enlightening.

Prometheus AEOlus [9] allows the integrated development of the three MAS dimensions (agent, environment and organisation) which contributes with: *(i)* a new meta-model that combines the meta-models of Prometheus and JaCaMo; *(ii)* a new interactive

incremental process based on the Prometheus process; and *(iii)* a code generation approach for JaCaMo based on this new meta-model. Prometheus AEOlus improves modelling, code generation and reduces the conceptual gap between the analysis and implementation phases. It extends Prometheus to include concepts that improve the modelling and code generation of the environment and organisation dimension for JaCaMo programming platform, where JaCaMo concepts were used to improve Prometheus development process to ensure that concepts used during the design and analysis stages will be used in the implementation stage. The code generation in Prometheus AEOlus requires the refinement of entities in the model to generate code (for JaCaMo components, i.e., Jason, Moise and CArtAgO). Thus, the models must be refined to include platform-specific information, and once the first version of code is generated, the models are no longer used during the programming step to complete the MAS development.

Research in the direction of tools for developing MAS through exploiting model-driven engineering techniques have led to a new proposal [4] of using Ecore with Prometheus. Ecore is used by the Eclipse Meta-modelling Framework to define meta-models, and it is applied to develop the meta-model concepts specific to Prometheus. More specifically, it addressed the generation of MAS graphical editors based on the models and how agent code generators can be developed from such visual models. In the end, MAS programming code can be automatically generated from the models, ranging from code skeletons to completely deployable products. To demonstrate this claim, templates have been created to automatically generate code in JACK language. Once the model is converted to code, the developer must continue the programming without using the model.

5 Final Remarks

This paper uses a MAS case study to highlight a number of discrepancies between the paradigms behind methodologies for AOSE and programming languages for MAS. In particular, the paper focuses on Prometheus [6] as the agent-oriented methodology and JaCaMo [1] as the framework for multi-agent programming. On one hand, Prometheus is one of the most well-known MAS model and methodology for developing intelligent agent systems. On the other hand, JaCaMo is a framework for MAS programming that combines three separate technologies: Jason for coding autonomous agents in AgentSpeak, CArtAgO for programming the environment as artefacts in Java, and Moise for specifying MAS organisations in XML. This paper highlights some aspects of MAS not covered or not aligned by models in Prometheus [6] when JaCaMo [1] is considered as the coding platform. The identification of mismatches between software engineering toots and programming languages can help in improving both and could result in a better alignment between them. Even for the concepts referred by the same name, there are differences and mismatches in the precise meaning of them. In fact, we provide evidences of gaps between the investigated approaches which allows to derive some important conclusions. For example, while many methodologies were proposed for agent programming in the past, they are not sufficient for the new and emerging techniques in agent programming, such as dealing with the multiple abstraction levels and not focusing only on the agents as individuals.

Our analysis of these techniques in practice allowed us to identify points for improvements. Such empirical study evaluates the state of the art in technologies and guide the development of new technologies based on the limitations of current ones. An interesting continuation of our work is to explore the same case study using alternatives for Prometheus [6] and JaCaMo [1]. We plan on expanding our models with more detailed strategies in Prometheus for this case study. Based on that, we want to advance our explorations on the relations of Prometheus with JaCaMo. The findings described in this paper are supported by a single case study, so we have to be careful to expand or generalise such discoveries for different contexts. Eventually, after trying other case studies and confirming the limitations shown here it will be possible to propose new approaches or enhancements for such AOSE approaches. Some limitations we faced when using Prometheus were already pointed out by their authors and claimed as future work [6], such as, for example, the introduction of social concepts to improve the models. However, these improvements are not available in the latest official version of PDT. One could question some of our modelling decisions, however, this paper is less about the details of the case study, and more about the broader vision towards AOSE approaches. Another interesting discussion is the relation of how similar problems exist and/or have been solved over the past 20 years in the general Software Engineering (SE) literature. A clear understanding of the evolution in mainstream SE can be crucial for evolving AOSE in the right direction.

References

1. Boissier, O., Bordini, R.H., Hübner, J., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Science of Computer Programming 78(6), 747–761 (2013)
2. Bordini, R.H., Dix, J.: Programming multiagent systems. In: Weiss, G. (ed.) Multiagent Systems 2nd Edition, chap. 11, pp. 587–639. MIT Press (2013)
3. DeLoach, S.A., Padgham, L., Perini, A., Susi, A., Thangarajah, J.: Using three AOSE toolkits to develop a sample design. International Journal of Agent-Oriented Software Engineering 3(4), 416–476 (2009)
4. Gascueña, J.M., Navarro, E., Fernández-Caballero, A.: Model-driven engineering techniques for the development of multi-agent systems. Engineering Applications of Artificial Intelligence 25(1), 159–173 (2012)
5. Padgham, L., Thangarajah, J., Winikoff, M.: The prometheus design tool a conference management system case study. In: Agent-Oriented Software Engineering VIII, LNCS, vol. 4951, pp. 197–211. Springer Berlin Heidelberg (2008)
6. Padgham, L., Winikoff, M.: Prometheus: A methodology for developing intelligent agents. In: Agent-Oriented Software Engineering III. LNCS, vol. 2585, pp. 174–185 (2003)
7. Smith, R.G.: The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Trans. Comput. 29(12), 1104–1113 (Dec 1980)
8. Sun, H., Thangarajah, J., Padgham, L.: Eclipse-based prometheus design tool. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems. vol. 1, pp. 1769–1770. IFAAMAS (2010)
9. Uez, D.M., Hübner, J.F.: Environments and organizations in multi-agent systems: From modelling to code. In: 2nd International Workshop on Engineering Multi-Agent Systems. pp. 181–203 (2014)