Agent Oriented Programming: from Revolution to Evolution (Position Paper)

Alex Muscar

University of Craiova, Romania
amuscar@software.ucv.ro

Abstract. Despite being around for quite some time, agents have failed to gain wide acceptance. Their AI heritage has forced them into a niche from which they cannot seem to escape: being the vehicle of AI experimentation. Even though the premises of Agent Oriented Programming (AOP) is a revolutionary departure from Object Oriented Programming (OOP) the vision has not materialized.
In this paper we propose taking a step back and looking at AOP as an evolution from OOP. Rather than viewing agents as specialized AI tools, we adopt a view on agents as a generic metaphor for building complex software systems. The guiding lines of our approach are: (i) overall conceptual simplicity; and (ii) the use of programming languages as the main means of expression. We will explore some paradigms and approaches that we think can greatly benefit the view of agents as an evolution from OOP.

1 Introduction

When introduced by Shoham almost two decades ago [16], Agent Oriented Programming (AOP) was intended as a higher level means of developing complex software systems – when compared to Object Oriented Programming (OOP). Soon after its introduction is was regarded by many as a "revolution in software" [7]. But the promises of the agent community have failed to materialize and agents haven't gained wide acceptance. They are mostly regarded as experimentation tools for the AI community instead of a technology for developing practical applications. Motivated by the agent paradigm's lack of success there are voices that predict its downfall [8]. At the same time some members of the agent community are trying to raise awareness to the lack of coherence and perspectives in the community [3,15].

We believe that the very fact of looking at AOP as a *revolutionary* step from OOP instead of an *evolutionary* one has been one of the main reasons that led to the current state of affairs. Given the AI heritage of the agent community it is no wonder that most of its efforts went into models of rational agents. While this has led to a series of interesting results it also meant that other aspects of agent systems such as the organizational ones or the ones regarding the environment in which agents operate have not been given the attention they deserve [6]. This

Andrew V. Jones (Ed.): Proceedings of the 1st Imperial College Student Workshop (ICCSW '11), pp. 52–58. September 29th–30th 2011, London UK.

was most unfortunate as those two aspects are essential to using agents as a generic metaphor for building complex software systems.

The purpose of this paper is twofold. First, it takes a conceptual step back, and looks at agents as an evolution of the OOP and the Actors model [1]. Second, it it tries to distill the core concepts that will go into the making of a new agent language, which we plan to develop in the near future. We have chosen two basic concepts which we plan to use as the basis of our language: *reactive objects* [11] and *dynamic delegation* as featured in *prototypical* languages such as SELF [17,4,18]. Based on this two concepts we believe that more advanced topics such as concurrency, autonomy and security can be tackled.

In [6], Dastani identifies the development of programming languages that combine the three key abstractions of multi-agent system (MAS) – agents, organizations and environments – as one of the main challenges of the agent community. After a brief survey of some popular agent languages in sec. 2 we focus on each of the aforementioned aspects in sec. 3. We conclude in sec. 4.

2 Motivation and Background

As stated above we adopt the view that languages profoundly influence the way programmers design systems. Thus, we think that a language needs to provide: i) a *uniform* model (i.e. everything is an agent); ii) *self-sufficiency* (i.e. the capacity to extend the language from within itself); and iii) *reified* concepts from the problem domain (e.g. agents, environments, organizations). Given this requirements, in table 1 we make a succinct overview of three of the most popular agent languages: *JASON*, *GOAL* and *MetateM.*

	JASON	**GOAL**	**MetateM**
Uniformity	no	yes	yes
Self-sufficiency	no	no	no
Reified concepts	none	none	agents

Table 1. Comparing agent languages

Each of the languages considered in our overview fails for one for more of our requirements. JASON is neither uniform – agents and environments are separate entities, nor does it feature reified entities. If we consider belief and knowledge bases as being part of the agent we can consider that GOAL is a uniform language since it only operates with agents, but this does not prove to be that useful since agent operations are limited and environments are declared implicitly. MetateM is the closest language to our desired model. It is uniform in the sense that agents are used to represent other agent's environments reifying the concept of agents in the process. All three languages fail when it comes to self-sufficiency since they can only be extended in Java.

3 An evolutionary approach

We will address each of the three concepts identified by Dastani in [6] – agents, organizations and environments – while trying to provide uniformity, self-sufficiency and problem domain reification.

3.1 Individual agents

In their simplest definition agents are *autonomous and interacting entities* [6]. Agents are autonomous in the sense that they can *decide* which action to perform next in order to reach their objective. This is indeed a very lose definition of agents. Instead, for the rest of this paper, we shall operate with the following definition: an agent is *a computational entity that (i) has its own thread of control and can decide autonomously if and when to perform a given action; (ii) communicates with other agents by asynchronous message passing*[1]. This definition implies *concurrently* executing agents with *reactive* behaviors. This is in line with the definition of AOP systems provided by Shoham in [16] where AOP is a specialization of OOP in the sense of the Actor model.

We begin by looking at three possible abstractions for single agent entities: objects, actors and *reactive objects* [11]. We have chose the former two since our purpose is going back to the origins of the agent metaphor, and the latter as an evolution of objects which borrows from actors.

	Objects	**Actors**	**Reactive Objects**
Granularity	object	actor	object
Execution model	sequential	concurrent	concurrent
Communication	sync	async	sync and async
Message ordering	strict order	no restriction	strict order

Table 2. Comparing agent abstractions

It is obvious from table 2 that reactive objects combine the best features of OOP and the Agent model. They are autonomous units of execution that are either executing the *sequential* code of exactly one method, or passively maintaining their state [11]. While this almost fits our definition of agents, there is still one aspect that we haven't fully addressed: autonomy. Reactive objects are autonomous in the sense of having their own thread of control there is still the issue of *decision making*, but in order to be fully autonomous an agent needs a decision making component [6]. A popular choice in the agent community is the *BDI model* which can be easily mapped on the concepts already introduced above. This view of agents gives us a higher level view which is especially useful when tackling the inherent concurrency in the execution model that we have

[1] We consider asynchronous programming as being characterized by many simultaneously pending reactions to internal or external events

defined for our agents. An interesting approach to dealing with concurrency by focusing on the execution of plans instead of programs or processes has been proposed for the E language: *communicating event loops* [9].

The communicating events loops model can be seen as a refinement of the reactive object semantics. When a thread of control[2] needs to send an asynchronous message send it ads an entry to a queue of *pending deliveries.* During a *turn* a pending delivery is dequeued, the message is sent, and all the resulting synchronous calls are executed. When a turn finishes, another entry from the pending queue is dequeued and the process starts all over again.

To address distributed scenarios a further refinement is introduced: objects running in the same event loops can be invoked both synchronously and asynchronously while objects running in remote event loops can only be invoked asynchronously. This offers isolation for plans executing in different event loops.

The communicating events loops model offers two key properties for concurrent systems:

Asynchrony messages between two event loops are sent asynchronously and the event loop controls when they are sent the risk of deadlocks is eliminated because an event loop can never interrupt its currently executing plan to wait for another event loop to execute its plan; and

Serializability an event loop reacts to incoming events serially the risk of race conditions is eliminated.

For these reasons this model has also been adopted by AmbientTalk [5] which deals with ad hoc networks, highly distributed and concurrent systems.

3.2 Organization

In order for a multi-agent system to achieve its purpose the behavior of individual agents has to be organized. Furthermore, the system needs to maintain some global invariants. This can be done endogenously, by making the organizational and regulatory aspects part of the agent. We think that the endogenous approach is cumbersome and that it obscures the development of individual agents. We find the exogenous approach much more appealing: agent's actions are externally controlled. This leads to a development style that's more modularized and decoupled.

Let's once again look at the OOP community for inspiration: dynamic delegation as featured in prototypical languages, such as SELF, seems to offer the flexibility we are looking for when it comes to structuring adaptive distributed systems, and we have chosen it for its elegance and conceptual simplicity. In [18], the authors introduce the concept of *traits* for organizing large prototypical systems. Traits are akin to abstract types: they are used to factor out common functionality shared multiple objects. Thus, altering the behavior of a trait also alters the behavior of all the objects that share it.

[2] For the purpose of this article we can think of *thread of control* and *event loop* as interchangeable terms.

Some of the research in agent organization has focused on using *coordination artifacts* – from the *Agents and Artifacts* (A&A) theory [12]. According to [13] coordination artifacts are entities designed to provide some kind of functionality or service, they have a well-defined interface, providing operations that can be invoked by agents. Indeed, coordination artifacts have much in common with traits. Another benefit of traits is that hey naturally handle dynamically changing behavior either by altering the trait as stated before, or by changing an object's traits on the fly.

Another advantage of adopting traits has to do with security. Since traits act as abstract types defining reusable behavior they are suitable for a system implementing object capabilities [10,2]. In this model there is no *ambient authority.* All actions are performed via unforgettable references to objects. The only component of the system which has full authority is the *powerbox* which, based on some security criteria, can hand references to various capabilities to the requesting objects. This system is extremely flexible since instead handing a reference to the actual capability, the powerbox can pass a reference to a stripped down version of the capability (e.g. in the case of a file system access capability it can pass a version that can only read files, but not delete them). Traits are especially suitable for this model since new traits can be defined in terms of existing one by *refining* their behavior [18].

3.3 Environment

One of the main models proposed for representing environments for MAS is A&A [14]. As stated in sec. 3.2, there is a close resemblance between coordination artifacts and traits. But this resemblance is not limited to coordination artifacts. Artifacts are generic bundles of behavior that provide a usage interface. They can be co-constructed and co-used by agents and they can be grouped in *namespaces.* This view of artifacts maps directly on prototypical objects, which thanks to their generality can act as traits, regular objects and namespaces. This opens up new ways of organizing systems.

3.4 A coherent view

By combining the communicating event loops model with the delegation semantics of prototypical languages we gain uniformity. We can define agents, artifacts and environments as event loops. Furthermore, since we can change delegation links at runtime, we can also reify agent behavior as traits[3]. This buys us flexibility and adaptability: an agent can adapt to varying environmental conditions by delegating to different traits. The advantages span to the distributed aspects of the system as well. By introducing a distinction between local and remote event loops we can take advantage of uniformity further: an agent can delegate to either a local or remote counterpart. In turn this can benefit agent mobility

[3] Which are also agents.

since agents are self contained entities. As long as their delegation links are appropriately adapted, agents can continue to function the same way on the new host as they did on the original one.

While this features could be implemented as an Domain Specific Language (DSL) – or even Embedded Domain Specific Language (EDSL) – we think that the approach would benefit most by being implemented as a full fledged language. Together, the features from the previous paragraph, make up for a language that conforms to our initial criteria: uniformity, self-sufficiency and reification of concepts from the problem domain. It also exposes the three main features identified by Dastani in [6] to the programmer.

4 Conclusion and Future Work

In this paper we have presented our view on agent languages. We have addressed the three main abstractions of such languages: agents, organizations and environments. In doing so we took a step back and tried to gear agent development toward a more generic audience while still keeping its high level view on computation.

We also looked at what hints the agent paradigm can take from OOP, and especially from the prototype-based model. We showed how the flexibility of prototypical objects can benefit the organization of MAS. We also briefly addressed how representing environments for agents can use the same model thus leading to a uniform representation.

In the near future we will closely focus on the ideas proposed in this paper, especially on implementing an agent language based on communicating event loops and we will investigate the use of traits in providing security, organization and the reification of environments. We will also address some open issues that we glossed over in this paper because of the lack of space: i) the interaction of the inherent dynamism in prototypical systems like SELF with the object capability model and event loops; ii) re-introducing some static checks in the context of such a highly dynamic system; and iii) language extensibility both at the syntactic and semantic level.

Some additional problems are related to runtime system of such an agent language. In order to offer concurrency, most of the current agent languages, use one Operating System (OS) thread per agent. But OS threads are expensive so such a solution does not scale well for systems with many agents. Using a *thread pool* seems like a good solution, but have yet to study the interaction of event loops and thread pools. To complicate things further, choosing a concurrency model (e.g. futures, promises) has a deep impact on the design of the language (e.g. using promises with callbacks for when they are resolved leads to the well known problem of inversion of control).

These are all serious issues that need to be settled, but we feel that the basic concepts (reactive objects with dynamic delegation) make for a solid foundation for an agent language targeted at MAS.

References

1. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, MA, USA (1986)
2. Bracha, G., von der Ahé, P., Bykov, V., Kashai, Y., Maddox, W., Miranda, E.: Modules as objects in newspeak. In: Proceedings of the 24th European conference on Object-oriented programming. pp. 405–428. ECOOP'10, Springer-Verlag, Berlin, Heidelberg (2010)
3. Castelfranchi, C.: Bye-bye agents? not. IEEE Internet Computing 14, 93–96 (March 2010)
4. Chambers, C., Ungar, D., Chang, B.W., Hölzle, U.: Parents are shared parts of objects: inheritance and encapsulation in self. Lisp Symb. Comput. 4, 207–222 (July 1991)
5. Cutsem, T.V., Mostinckx, S., Boix, E.G., Dedecker, J., Meuter, W.D.: Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In: Proceedings of the XXVI International Conference of the Chilean Society of Computer Science. pp. 3–12. IEEE Computer Society, Washington, DC, USA (2007)
6. Dastani, M.: Programming multi-agent systems (May 2011)
7. Guilfoyle, C., Warner, E.: Intelligent Agents: the new revolution in software. Ovum (1994), incomplete ref
8. Hewitt, C.: Perfect disruption: The paradigm shift from mental agents to orgs. IEEE Internet Computing 13, 90–93 (January 2009)
9. Miller, M.S., Tribble, E.D., Shapiro, J.: Concurrency among strangers: programming in e as plan coordination. In: Proceedings of the 1st international conference on Trustworthy global computing. pp. 195–229. TGC'05, Springer-Verlag, Berlin, Heidelberg (2005)
10. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. Ph.D. thesis, Johns Hopkins University, Baltimore, Maryland, USA (May 2006)
11. Nordlander, J., Jones, M.P., Carlsson, M., Kieburtz, R.B., Black, A.P.: Reactive objects. In: Symposium on Object-Oriented Real-Time Distributed Computing. pp. 155–158 (2002)
12. Omicini, A.: Formal respect in the a&a perspective. Electron. Notes Theor. Comput. Sci. 175, 97–117 (June 2007)
13. Ricci, A., Viroli, M.: Coordination artifacts: a unifying abstraction for engineering environment-mediated coordination in MAS. Informatica 29(4), 433–443 (2005)
14. Ricci, A., Piunti, M., Viroli, M., Omicini, A.: Environment programming in cartago. In: El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H. (eds.) Multi-Agent Programming:, pp. 259–288. Springer US (2009)
15. Santi, A.: From objects to agents: Rebooting agent-oriented programming for software development
16. Shoham, Y.: Agent-oriented programming. Artif. Intell. 60, 51–92 (March 1993)
17. Smith, R.B., Ungar, D.: Self: The power of simplicity. Tech. rep., Mountain View, CA, USA (1994)
18. Ungar, D., Chambers, C., Chang, B.W., Hölzle, U.: Organizing programs without classes. Lisp Symb. Comput. 4, 223–242 (July 1991)