Method Calls Not Considered Harmful for Agent Interactions

Lars BRAUBACH [a] and Alexander POKAHR [a]
[a] *Distributed Systems Group, University of Hamburg, Germany*
Email: {braubach, pokahr}@informatik.uni-hamburg.de

AbstractSince the very beginnings of agent technology it is considered a fundamental property that communication between agents is done using asynchronous message passing. One important reason for this assumption is that a method call on an agent would break its autonomy. Conceptually, it would allow an agent to exert control over the behavior of another agent by directly telling it to execute a specific method. Furthermore, more than one agent could call methods on the same agent concurrently possibly leading to state consistency problems. In contrast to agents, in mainstream programming paradigms like object, component or service orientation method calls are the dominant communication means. Being able to define interfaces from method signatures represents an important advantage of those approaches currently not available for agents. In this paper we will argue how it is possible to introduce service interfaces and method calls for agents without breaking their autonomy and endangering their state consistency. To achieve this, concepts from active objects, services and components are brought together. The usefulness of the approach is underlined with an example application from the area of distributed image rendering.

1. Introduction

Many definitions of the agent concept have been proposed and it can be observed that these definitions typically consider an agent under a specific perspective, e.g. as personal assistant, as design metaphor or as extended object, and highlight corresponding properties [1]. In order to unify these different points of views Wooldridge and Jennings proposed the weak and strong notions of agency [2], which represent agent definitions based on important agent characteristics. The weak notion emphasizes that agents are *autonomous* with respect to their behavior decisions, capable of *reactive* and *proactive actions* and *social* by being able to communicate with each other. The strong notion adds *mentalistic notions* as description means for agents, i.e. an agent is programmed using e.g. beliefs, desires and intentions.

In this paper a software centric perspective is emphasized and the social dimension of agents is analyzed with respect to the question whether and how method calls fit in agent programming. It is a fundamental assumption that agent interactions are based on asynchronous message passing, e.g. pushed forward by introductory literature [3, p. 132 ff.], existing standards like FIPA [4] and KQML [5] that focus on message level communication and finally also by well-known agent platforms like JADE [6] or LS/TS [7]. In contrast, method invocations are object-level interactions that violate the autonomy of an agent by interfering with its behavior control from the side of the method invoker. Despite this obvious danger method calls are the dominant interaction scheme in most successful software paradigms used in practice such as objects, components and services. They are a fundamental ingredient for describing functionalities of entities in a precise manner using interfaces composed of publicly available method signatures. Their importance for building distributed systems is especially highlighted by Web Services and corresponding standards like WSDL [8], which in core deal

with how invocations can be described in a machine independent way. In this paper we will argue how agents can be conceptually designed and extended to also allow method invocations without losing their autonomy.

For this purpose, first in Section 2 a running example will be introduced. In Section 3 the problem is concretized and further explained using the example application. Thereafter, in Section 4 the conceptual solution is presented and its implementation is shortly sketched in Section 6. In Section 6 related work is discussed. A discussion is given in Section 7 and in Section 8 some concluding remarks and ideas for future work are presented.

2. Running Example

In order to illustrate the existing problems and motivate our solutions a running example called *Mandelbrot* is introduced. It allows users to render fractal images based on well-known fractal algorithms (e.g. Mandelbrot, Lyapunov, etc.). To speed up the rendering process, the system should be able to distribute the computation across different hosts in a network. The application can be coarsely divided into three different units of functionality. The *display* provides interaction capabilities for a user of the system. It is responsible for presenting rendered images to the user and allowing the user to issue new rendering requests (e.g. by zooming into the picture or by manually entering values). A *generator* handles user requests and decomposes them into smaller rendering tasks. It acts as a coordinator responsible for task distribution and result collection. A *calculator* accepts rendering tasks and returns results of completed tasks to the generator. It has access to different fractal algorithms and is thus able to provide the color values for pixels of the image to be rendered.

3. Problem Statement

In order to understand better the status quo of communication in agent systems first several problems of message-based interactions will be discussed:

- Messages are low-level building blocks which decompose even simple request reply schemes to sending, waiting for response and reacting on response activities that explicitly have to be considered and arranged by an agent programmer. As for example waiting for a response is part of the application code errors may be introduced by e.g. using a wrong conversation identifier etc. Thus, messages do not allow constructing flexible and robust interaction in an easy way [9].
- Messages are a concept that is often considered as being orthogonal to internal agent architectures. This forces developers to think in additional concepts when it comes to multi-agent programming tasks. As an example consider intentional BDI agents which allow thinking in terms of goals and plans for achieving goals. If such a goal cannot be fulfilled by an agent itself a developer would like to think in terms of goal delegation to another agent and not in terms of low-level messages. This is one of the reasons why alternatives like goal-based interactions have been developed to hide message passing [10].
- Agent message structures are complicated as FIPA language specifications introduce a lot of different parameters besides sender, receiver and content, e.g. for content encoding used ontology and interaction flow control. Additionally, FIPA messages are ontology-based so that it is not trivially possible to pass complex data structures. These have to be encoded as part of an ontology and often special tools like code generators have to be used to make them transferrable. For a developer, even simple tasks like requesting an action from another agent leads to a lot of preparatory efforts to be taken. As a result, the code to be written for assembling the messages is often considerably complex and typically requires dealing with communication issues (e.g. data representation formats such as FIPA-SL) as part of the

functional code. Taken together, preparatory actions and cumbersome assembly slows down agent development considerably.
- Messages do not contribute to distribution transparency. They require specifying receivers explicitly so that an application developer cannot easily abstract from the targets identity although this identity might be dispensable to know for the sender. In order to alleviate the tight coupling of senders and receivers e.g. topic-based message communication has been introduced, e.g. as part of service busses and also in some agent platforms like e.g. Cybele[1].

One often quoted advantage of agent messages is the semantic layer on which they operate. We will deal with this issue in depth later in Section 7. To complement the problems of messages in the following several advantages of the method-based invocation scheme are highlighted:

- *Methods are important for software engineering characteristics:* Method signatures are necessary for interface specifications which are the conceptual means for separating functionality specification and implementation. In turn, without interfaces desirable software engineering properties like modularity and extensibility are difficult to achieve.
- *Methods are part of most practice-oriented software engineering paradigms except agents:* Looking at the historical evolution of programming paradigms one can see that the transition from hardware near approaches to procedural programming was an important step forward that introduced the notion of a procedure for encapsulating functionality and making it subject to further reuse. The method concept has been adopted in the context of object orientation, which brings together local state and behavior specification using classes. Object-oriented systems are typically composed of object hierarchies, in which communication between objects is performed primarily by method calls. Other contemporary programming paradigms such as component and service orientation also embrace this basic communication scheme and support interaction based on method call semantics.
- *Methods are natural and sufficient for many real world application scenarios:* Due to the success of object, component and service-oriented approaches in diverse projects it is apparent that these paradigms offer suitable abstractions for building complex applications, i.e. they are sufficient for many real world application scenarios. Such scenarios can often naturally be described with providers and consumers of functionality interacting in a simple request/response scheme in the sense of service-oriented architectures. Such an interaction style can directly be mapped to method calls and does not call for elaborated coordination strategies. In this sense method calls simplify the development of many applications as other communication schemes like message passing often add complexity without providing additional value.
- *Methods are well-known with easy to understand semantics:* Nowadays, object orientation is the dominant programming paradigm that is part of many University software engineering curricula and other education programs. Therefore object-oriented concepts are well known and understood by most developers and thus the thinking of developers is often grounded in object concepts.

For the above mentioned reasons it can be concluded that object orientation in general and method-oriented communication in particular are important criteria for the wide acceptance of any new programming approach. In an earlier paper we claimed that agent technology is only to succeed if it clearly convinces people that it a) has something new and important to add that is currently missing and b) does not remove anything essential from existing approaches [11]. In our view, removing methods from the spectrum of available communication means represents a serious software technical limitation that hinders the wide adoption of agents.

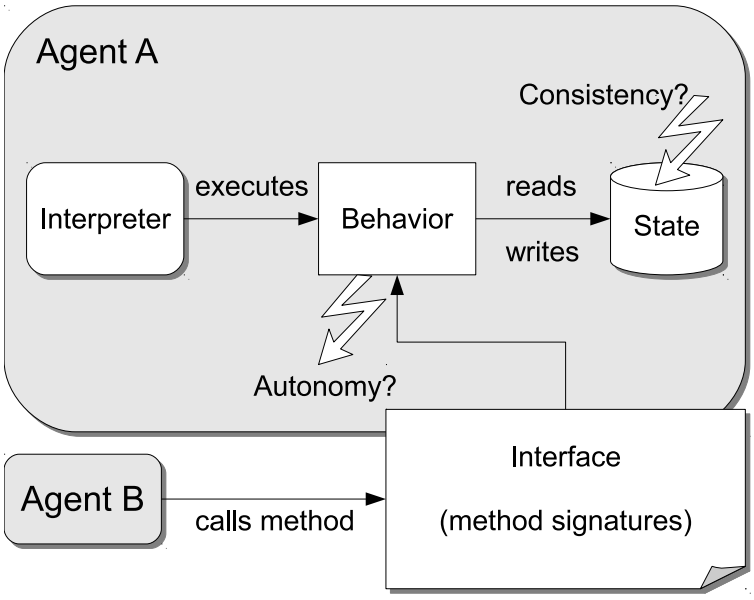

Figure 1. Problems of method calls

3.1. Fundamental Problems with Methods and Agents

The main problems regarding agents that communicate with method calls are depicted in Fig. 1. It is assumed that an agent, regardless of its concrete internal architecture, always consists of an interpreter that executes domain specific behavior. This behavior may access the agent's state at any time for read and write operations. In this simple setting an agent may directly expose parts of its behavior via an interface composed of method signatures. In the following the implications are discussed when an agent B invokes a method on another agent A from its publicly available interface.

- *Autonomy is endangered:* In the scenario agent B directly accesses behavior of agent A by calling its method. With the call it forces agent A to execute an action it may not want to perform at all. Additionally, the question arises how the behavior from the invoked method fits to the internal behavior agent A is currently executing. If these behaviors are semantically contradicting this may lead to irrational behavior of agent A.
- *Consistency is endangered:* Both agents are autonomous entities so that they can behave independently of each other, i.e. conceptually they represent separate processes. Given that agent B directly executes behavior of agent A using a method call, this behavior may read and write data from/to its state. As both agents possibly act concurrently, typical data consistency problems like data corruption and lost updates can occur. State consistency is also endangered by transferring object references between agents, which may occur when an agent calls a method with parameters having call-by-reference semantics. Here concurrent access to objects may occur as more than one agent can work with the same object from its state.

3.2. Problem Analysis

This section discusses requirements that should be met when considering method calls as an interaction mechanism for agents. The requirements are based on the assumption that the interaction should exhibit the above mentioned advantages of method calls, but should also be in line with the agent metaphor by not violating the fundamental properties of autonomy and state consistency. First, the

[1] `http://www.i-a-i.com/default.asp`

main consequences of autonomy and state consistency are identified and afterwards it is analyzed with the help of the running example, how they can be reflected in method-call-based programming styles.

Autonomy is defined as *"independence or freedom, as of the will or one's actions"*[2]. With respect to method calls on an entity this can be interpreted as the ability of the entity to decide for itself, if and when a requested action will be performed. More specifically, an entity should not be forced to directly execute method calls as they happen, but it should be able to *reorder* (e.g. prioritize) calls, *delay* calls for other reasons (e.g. collect requests before starting to work on them) or also completely *reject* incoming calls. Moreover, the entity may perform additional *side-effects* as a result of the call (e.g. logging), of which the caller might not be aware of. Considering agents, this interpretation of method call autonomy can be further extended towards how the agent comes to a decision on how to react to a call. Depending on the type of agent as being a goal-oriented, situated, and/or social entity, different approaches should be supported to fully reflect the bandwidth of the agent paradigm. E.g. as a result of receiving a method call, a goal-oriented agent might perform complex reasoning processes based on its current beliefs before deciding to execute the requested action, while a social agent acting in a group might coordinate its behavior with other agents using sophisticated interaction protocols.

With regard to consistency, it is well known in object-oriented programming that writing correct concurrent code is nearly impossible in practice using low level concepts like threads and locks [12]. The reason for this is the exponential growth of possible interleavings with regard to the number of algorithm steps even when considering only two threads. Due to race-conditions, concurrency situations are difficult to reproduce, making testing and debugging of concurrent code very hard and error prone. On the other hand, concurrency is a desired and also required property of distributed systems, because it is needed for utilizing the available computational resources (e.g. multi-core processors) and is also an inherent property of all distributed systems that are composed of independent (and thus concurrently executing) nodes. Unlike other paradigms such as object orientation, the agent paradigm offers a conceptual abstraction for concurrency [13]: an agent is an independently acting entity that encapsulates its own (logical) thread of control. As interaction between agents is based on asynchronous message passing, no locks are required and thus many of the problems of concurrent programming are alleviated. This conceptual advantage of the agent paradigm needs to be preserved when introducing method calls as a new interaction scheme.

To illustrate possible solutions to the above problems, we consider implementations of the calculator entity from the Mandelbrot example application (see Fig. 2). As we want to support method-call style interaction, we define an object-oriented interface of the calculator entity called *ICalculateService* (lines 1-3), which offers to calculate an area of a fractal image as specified in the *AreaData* parameter. At first, we consider the first option with a direct return value (line 2a). A naive implementation of the method (lines 8-12) represents the typical way of object-oriented programming, but offers no autonomy or consistency at all, because the code of the method is executed directly and unconditionally on the caller's thread. While this implementation therefore does not provide a solution to the autonomy and consistency problem, it represents an ideal that we would like to come close to in the sense of keeping the programming as simple as possible. In the follwing, three independent improvements of the naive implementation are explained, each of which addresses different requirements. To isolate the state of caller and callee, supplied parameters and the return value need to be cloned, such that caller and calle always operate on local copies of these values. This can be achieved by providing a *clone()* operation and applying the operation to the corresponding values (lines 15-20). The next implementation (lines 23-31) incorporates a decision, if a call should be accepted. The decision itself is moved to a new internal method *acceptCall()* (line 16), such that it can be implemented independently of the business logic in the *calculateArea()* method. This solution allows rejecting calls in

[2]Dictionary.com: Autonomy of the individual (`http://dictionary.reference.com/browse/autonomy`)

```
01: public interface ICalculateService {
02a:  public AreaResult calculateArea(AreaData area); // Option a: direct return value
02b:  public IFuture<AreaResult> calculateArea(AreaData area); // Option b: Future return value
03: }
04:
05: public class CalculatorAgent implements ICalculateService {
06:
07:   // No autonomy or consistency
08:   public AreaResult calculateArea(AreaData area) {
09:     AreaResult result;
10:     // Perform calculation...
11:     return result;
12:   }
13:
14:   // Isolated state
15:   public AreaResult calculateArea(AreaData area) {
16:     area = clone(area);
17:     AreaResult result;
18:     // Perform calculation...
19:     return clone(result);
20:   }
21:
22:   // Decided execution
23:   public AreaResult calculateArea(AreaData area) {
24:     if(acceptCall()) {
25:       AreaResult result;
26:       // Perform calculation...
27:       return result;
28:     } else {
29:       throw new RuntimeException(...);
30:     }
31:   }
32:
33:   // Decoupled execution
34:   public IFuture<AreaResult> calculateArea(AreaData area) {
35:     Future<AreaResult> result = new Future<AreaResult>();
36:     scheduleTask(new CalculateAreaTask(area, result));
37:     return result;
38:   }
39:
40:   // Combined solution
41:   public IFuture<AreaResult> calculateArea(AreaData area) {
42:     area = clone(area);
43:     Future<AreaResult> result = new Future<AreaResult>();
44:     scheduleTask(new DecideCallAcceptance(result,
        new CalculateAreaTask(area, result)));
45:     return new CloneFuture(result);
46:   }
47: }
```

Figure 2. Autonomy and consistency in OO method calls

which case the caller will be informed about the call rejection in terms of an exception. To achieve state consistency and also allow delaying or reordering of calls, decoupling needs to be introduced (lines 34-38). Here, the calculation is moved to a separate class called *CalculateAreaTask*. This task is scheduled to be executed on the thread of the agent using a *scheduleTask()* method (line 36), which might be provided by the underlying agent platform or framework. Now the execution is performed asynchronously, i.e. independent of the caller thread, which means that the method call returns after the task is scheduled, but maybe before it has been executed. Therefore the result of the calculation cannot be directly provided, but needs to be wrapped in a *Future*[3] object [14] using option b of the interface (c.f. line 2b). The future is passed to the calculation task as well as to the caller. After the calculation is finished, the result will be stored in the future, where the caller can retrieve it. When the future supports the observer pattern, the caller can be notified when the result is available, such that it does not have to periodically poll the future for the result. The last implementation (lines 41-46) combines the previous three solutions. The decision has been implemented as a separate *Decide-*

[3] *A future represents a placeholder object for the result of an asynchronous call. It helps decoupling the caller from the callee by freeing the latter from immediately processing the request. Instead the callee returns the future and processes the call when it deems it appropriate. A future typically offers a notification scheme for the caller to become aware of the result, which is stored in the future. On the one hand the future can be probed in order to test if the result has been delivered and on the other hand a blocking call on the future can be used to let the caller explicitly wait until the result is available (wait-by-necessity). To avoid such blocking waits a future can also offer a listener interface. In case the callee has produced the result and saved it in the future, it will automatically notify all installed listeners and in this way continue processing in the caller.*

CallAcceptance task, which is scheduled on the agent. When the call is accepted, the actual *CalculateAreaTask* will be scheduled by the *DecideCallAcceptance* task. Otherwise an exception will be set as the result of the future, such that the caller will know, that the call has been rejected. Instead of the original result future, a *CloneFuture* is returned, which takes care of cloning the result value before making it available to the caller.

The nesting of asynchronous tasks as shown in the last implementation is a powerful pattern, that meets the requirements for autonomy and consistency with respect to agent method calls. Calls can be rejected, reordered and delayed. The actual business logic code is always executed on the agent thread and thus does not cause consistency issues. Moreover, side-effects of calls can easily be introduced as an additional task. The developer may choose to add the side-effect before or after the *DecideCallAcceptance* task, if she wants the side-effect to happen for all or only for accepted calls.

Although autonomy and consistency are preserved by the introduced pattern, some drawbacks can be noted. The solution imposes a considerable code overhead (e.g. compared to the naive implementation several additional classes are necessary for describing the tasks) and thus the initial goal of capturing the simplicity of method calls is not achieved. The additional code also introduces new error sources, e.g. a developer may accidentally call methods, which are not considered to be called from other agents and are thus not decoupled. Moreover, in all methods which apply the decoupling pattern, the reasoning logic (call acceptance, side-effects) has to be hard coded. When an agent wants to expose many methods, all corresponding method implementations need to be changed when the developer chooses to alter the reasoning performed by the agent.

3.3. Summary

The above discussion has shown that autonomy and consistency are important properties that need to be taken into account, when thinking about method call style agent interactions. Technically, existing object-oriented programming techniques can be used to meet the identified requirements. Yet, the considered programming patterns introduce a considerable overhead and have limited flexibility. Therefore, in the next section a generic solution is presented, which provides integrated support for method call style agent interactions as part of an agent programming framework and thus relieves the developer from the burden of implementing these details by hand.

4. Approach

Method calls can be integrated into the agent concept by following the active object design pattern [15] and enriching it with agent reasoning capabilities. The changed method invocation scheme for agents is shown in Figure 3. It can be seen that method calls are routed into an interceptor chain for the purpose of transparently (for the caller and partially also for the agent programmer) pre- and postprocessing a call. The interceptor chain has the purpose to induce actions before and after a call in order to ensure the agent's autonomy and its state consistency. In order to achieve both specific decoupling and reasoning interceptors are needed, which are explained in the following.

- *Autonomy is safeguarded:* The reasoning interceptor has the purpose of initiating agent behavior specifically designed for reasoning about method calls. This reasoning behavior is in charge of deciding about all important aspects how the call should be handled, e.g. if the call should be rejected or delayed. Depending on the type of the agent other forms of reasoning interceptors can be used, e.g. in case of a BDI (belief desire intention) [16] agent a specific service goal can be created. The call semantics for agents is defined as follows. From a programmer's point of view a method invocation should always lead to method body execution, because it is a fundamental principle of the object-oriented programming model. In case of action requests for agents this is not the case due to their increased behavior autonomy. The semantics for service invocations lies in between the object-oriented call and the message-

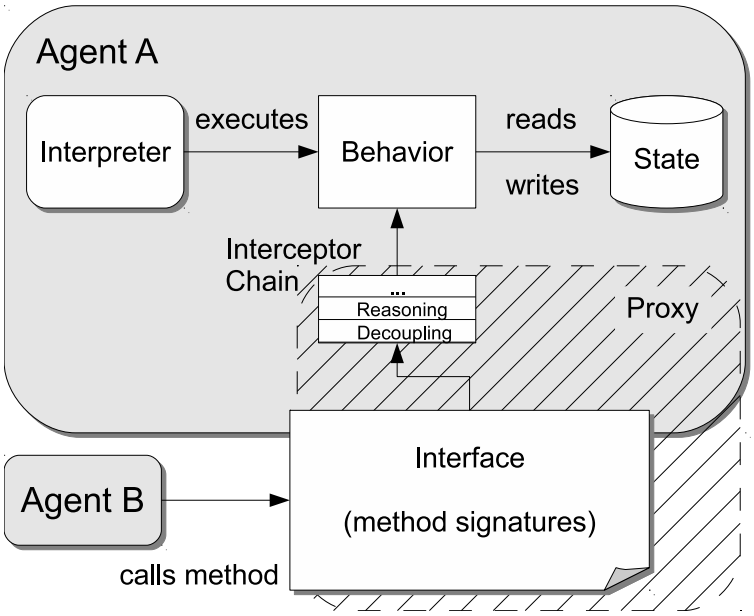

Figure 3. Solution for method calls

based action request semantics. On the one hand, it makes sense to assume a reasonable and simple default behavior for service invocations that is near to the object and service-oriented world and on the other hand services should not be understood as a means for reducing the possible autonomy of the entity. Thus, we define that the invocation of a method *m* of service *s* on agent *c1* from agent *c2* with the intention of getting an action *a* executed must not lead to action execution (1). Instead, it must only lead to a result or failure response *r* (within some timeout) to let the caller know about the method call consequences (2).

$$\neg\Box(invoked(c_1, m, s, c_2) \wedge request(m, a) \rightarrow \Diamond done(c_1, a)) \quad (1)$$

$$\Box(invoked(c_1, m, s, c_2) \rightarrow \Diamond result(c_2, m, r)) \quad (2)$$

- *Consistency is safeguarded:* To ensure consistency it is a mandatory prerequisite to protect the agent state from concurrent access. Thus, the decoupling interceptor has the task of routing a call from the callers execution thread to the execution thread of the callee, i.e. from agent B to agent A in the scenario. This is achieved in the same way as in active objects. Instead of directly executing the call, a call request is appended to an action queue of the called agent. The interpreter monitors the queue and fetches entries for processing whenever available. In this way the call is made part of the agent behavior and the state is not accessed from outside but only from the agent itself. The decoupling interceptor also assures that method parameters and results are passed using a call-by-copy semantics.

Compared to the preliminary solution pattern of Section 3.2, the integrated approach presented above offers several important advantages. Firstly, the agent platform manages the method call interaction by routing all calls through the interceptor chain. Therefore, the caller does not have access to the agent object directly, but only to a proxy of the agent, which is automatically generated by the platform. This proxy object exposes only the methods of those interfaces that the agent programmer has explicitly chosen to be visible to the outside. Therefore no accidental calls of unprotected methods can occur. Secondly, as a default the interceptor chain includes a decoupling interceptor that schedules calls on the agent thread. Therefore consistency is automatically assured and as a result method implementations inside the agent can follow the naive scheme without any code overhead. Thirdly,

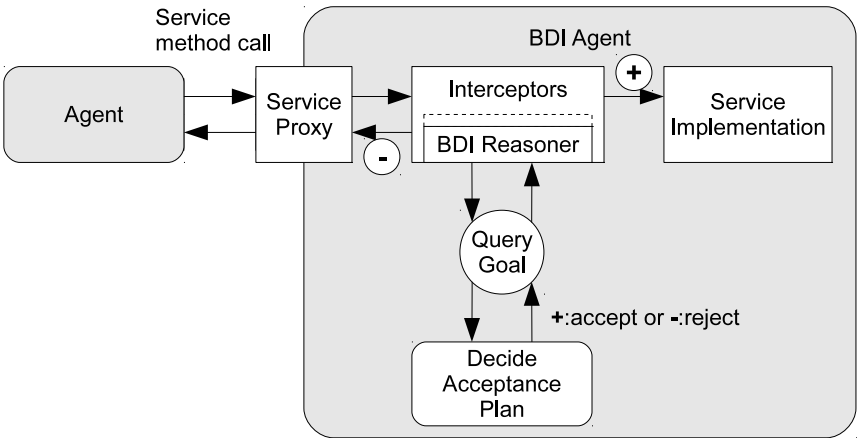

Figure 4. BDI interceptor call visualization

reasoning concerning the autonomy of the agent is separated from the business logic. The agent developer can implement various types of interceptors with different approaches for reasoning, e.g., about call acceptance, and add these interceptors to the interceptor chain where needed. Finally, this separation of concerns further facilitates runtime flexibility, because the interceptor chain can be easily changed during the lifetime of an agent. E.g. an agent might decide temporally enabling logging or access control by adding/removing an appropriate interceptor.

4.1. BDI Reasoning Interceptor

The interceptor concept will be further illustrated using the mandelbrot example. In order for the generator agent to be able to distribute the work pieces efficiently to available calculator agents it is assumed that the calculator agents can refuse service calls if they are already working on another task. Such a refusal will immediately be propagated back to the generator agent using a service invocation exception. The intended call processing is also shown in 4. An arriving service call arrives at the service proxy and passes the interceptor chain. After decoupling and state isolation has been performed a user defined BDI reasoning interceptor is called. This interceptor will asynchronously create a goal in the agent for deciding about call acceptance. After some plan has been performed reasoning logic the control flow branches according to the result (either + for acceptance or - for rejectance). In case of acceptance the service implementation is invoked and otherwise the interceptor directly returns an exception to the calling agent. The BDI calculator agent implementation with a reasoning interceptor is depicted in Figure 5).

The BDI calculator agent (lines 1-24) consists of the provided calculate service, a goal for reasoning about the call acceptance and a plan for handling this goal. It can be seen in Figure 5 that the calculate service (lines 18-23) is defined using its service interface ICalculateService (line 19), its implementation class CalculateService (line 20) containing only functional code for rendering the area and the additional BDI reasoning interceptor class BDIReasoningInterceptor (line 21). The processing logic of the agent is as follows. Whenever a method call on the provided service arrives, it will go through the interceptor chain eventually arriving at the BDIReasoningInterceptor. The interceptor (lines 31-50) only delegates the acceptance call reasoning to the agent via dispatching a decide_call_acceptance query goal.[4]

The reasoner consists of two methods. The first method isApplicable() is called to determine if the interceptor is allowed to handle the call at all. If not, processing continues with the next interceptor

[4]A query goal is a goal type that aims at retrieving information. In the example it is considered as succeeded when the parameter 'execute' is supplied with a value. More details about goal types can be found in [17].

from the chain. If it is applicable, the execute method of the interceptor will be invoked. The BDI-ReasoningInterceptor intercepts only calls to the calculateArea() method (line 33) and handles calls by creating and dispatching a decide_call_acceptance goal (lines 37, 47). As result a future is returned, which will store the result of the reasoning. For retrieving the result a goal listener is added on the goal (line 38), which subsequentially uses the parameter of the goal to decide if the method will be called (lines 40-45). Otherwise a service exception is generated and set as result of the future (line 44).

The goal (lines 3-5) possesses one out parameter called execute, which denotes the execution decision. In order to determine if the call should be accepted a plan will be executed for the query goal. The plan definition (lines 8-16) consists of a parameter mapping (lines 9-11), a plan body declaration (line 12) and a trigger definition (line 13-15). The body is meant to define the Java class (here DecideCallAcceptancePlan), which is instantiated and executed whenever the trigger is activated. The trigger states that it reacts to decide_call_acceptance goals. Finally, the parameter mapping is used to connect goal and plan parameters. Here, it means that the goal execute parameter can be accessed under the same name also in the plan. The implementation of the plan is kept very simple in this example (lines 26-29). It just sets the boolean execute parameter to true whenever it is idle, and to false otherwise (using the isBusy() function).

```
01: <agent ... name="Calculator" package="...">
02:   <goals>
03:     <querygoal name="decide_call_acceptance">
04:       <parametername="execute" class="Boolean" direction="out"/>
05:     </querygoal>
06:   </goals>
07:   <plans>
08:     <plan name="decide_call_acceptance_plan">
09:       <parameter name="execute" class="Boolean" direction="out">
10:         <goalmapping ref="reasoncall.execute"/>
11:       </parameter>
12:       <body class="DecideCallAcceptancePlan"/>
13:       <trigger>
14:         <goal ref="decide_call_acceptance"/>
15:       </trigger>
16:     </plan>
17:   </plans>
18:   <services>
19:     <providedservice class="ICalculateService">
20:       <implementation class="CalculateService"/>
21:       <interceptor class="BDIReasoningInterceptor"/>
22:     </providedservice>
23:   </services>
24: </agent>
25:
26: public class DecideCallAcceptancePlan extends Plan {
27:   public void body() {
28:     getParameter("execute").setValue(isBusy());
29: }
30:
31: public class BDIReasoningInterceptor implements IInterceptor {
32:   public boolean isApplicable(ServiceInvocationContext sic) {
33:     return sic.getMethod().getName().equals("calculateArea");
34:   }
35:   public IFuture<Void> execute(ServiceInvocationContext sic) {
36:     Future<Void> ret = new Future<Void>();
37:     final IGoal g = createGoal("decide_call_acceptance");
38:     g.addGoalListener(new IGoalListener() {
39:       public void goalFinished(AgentEvent ae) {
40:         if((Boolean)g.getParameter("execute").getValue())
41:           sic.invoke().addResultListener(
42:             new DelegationResultListener<Void>(ret));
43:         else
44:           ret.setException(new ServiceException(sic.getMethod()));
45:       }
46:     }
47:     dispatchTopLevelGoal(g);
48:     return ret;
49:   }
50: }
```

Figure 5. BDI calculator agent with reasoning interceptor

```
01: public interface ICalculateService {
02:   public IFuture<AreaResult> calculateArea(AreaData area)
03: }
04:
05: // Part of agent
06: public class CalculatorAgent extends MicroAgent
      implements ICalculateService {
07:   public IFuture<AreaResult> calculateArea(AreaData area) {
08:     AreaResult result;
09:     // Perform calculation...
10:     return new Future<AreaResult>(result);
11:   }
12: }
13:
14: // Separate class
15: public class CalculateService implements ICalculateService {
16:   @ServiceComponent
17:   private IBDIInternalAccess agent;
18:
19:   public IFuture<AreaResult> calculateArea(AreaData area) {
20:     AreaResult result;
21:     // Perform calculation...
22:     return new Future<AreaResult>(result);
23:   }
24: }
```

Figure 6. Implementation choices for the calculator service

5. Implementation

The interceptor chain approach for method call interactions has been implemented in the open source Jadex agent framework.[5] In Jadex, agents are realized as so called active components [18,13], that may expose services defined as Java interfaces to other agents. The internal behavior of an agent can be specified according to one of several available internal architectures. E.g. so called *Micro* agents allow to easily realize simple (e.g. ant like) agent behavior in pure Java classes while *BDI* agents allow describing complex reasoning behavior based of the belief-desire-intention model [16].

5.1. Services and Service Proxies

The connection between a service interface and the corresponding agent behavior is realized in a class, that implements the corresponding service interfaces (see Figure 6). For Java-based (e.g. Micro) agents the developer can choose to implement services directly as part of the agent class (lines 6-12) or in a separate Java class. Other agent types such as the XML-based BDI agents always have to provide separate classes for service implementations (lines 15-24). It can be seen that the only difference between the naive implementation of Section 3.2 and the implementation as a Jadex agent is the introduction of a future as result (line 10 and line 22). This is required for the method signature being conform to the interface, which provides the *calculateArea()* operation in an asynchronous way (line 2).

The Jadex platform automatically creates proxies for the service implementations, which include a corresponding interceptor chain. Per default, this interceptor chain includes interceptors for automatic service validation, execution decoupling and state isolation. In this context, service validation ensures a weak form of call acceptance by rejecting all invocations to services that are not in running state, i.e. have not been initialized or already been terminated.

The proxy approach allows implementing services as pure Java objects, e.g. neither the *ICalculateService* interface nor the *CalculateService* implementation need to extend or implement any ex-

[5] `http://jadex.sourceforge.net/`

```
+ <methodname>(<param>[0..*]): void
+ <methodname>(<param>[0..*]): IFuture<type>
+ <methodname>(): <type>
```

Figure 7. Allowed service method signature types

isting API of the Jadex platform. For such plain Java services, the Jadex platform offers annotations for dependency injection [19], e.g. of the agent that owns the service (lines 16, 17), such that the service can access its agent (*@ServiceComponent*), or to define specific startup or shutdown methods of the service (*@ServiceStart*, *@ServiceShutdown*).

Finally, proxies allow relaxing the restriction that service methods must always return future objects. As a common case, besides methods that represent executable service operations, services often have informational methods that allow accessing fixed properties of a service (e.g. name or some quality attributes). When these properties do not change during the lifetime of a service, they can be precached, i.e. when a proxy for a service is created, all properties are retrieved and stored in the proxy. When later another agent wants to access a property of this service through the proxy, the value can be synchronously returned, because it is contained in the proxy and no access to the agent which contains the service is required. As a result, service methods can follow one of the three signature types shown in Fig. 7, the first case being a one shot call (no result or error is available), the second case being the normal future-based asynchronous call and the third case being the described synchronous property access.

5.2. Programming Model

Figure 8 illustrates the resulting asynchronous object oriented programming model for agent interactions. The general invocation scheme is shown in the upper half of the figure using UML sequence diagram notation. In the lower half, the code of the caller (left) and the receiver (right) are given. First, on the caller side, the *calculateArea()* method of the calculate service is invoked (1). This call is handled by the service proxy, which applies the interceptor chain to the call and afterwards returns a future to the caller (2). As part of the interceptor chain, the decoupling interceptor has scheduled the call on the receiving component (agent). When the call is accepted, i.e. not rejected by a reasoning interceptor, the agent interpreter will invoke the *calculateArea()* method on the service implementation (3). For illustration purposes, we assume that the calculator agent needs to lookup the algorithm in a database, which is again realized using an asynchronous call with a future result. The agent adds a result listener to the future (4), thus letting the agent interpreter know that the implementation should be called again, when the algorithm is available. The result listener extends a framework base class *ExceptionDelegationResultListener*, which assures that any exception during algorithm lookup automatically gets propagated to the result future *ret* of the *calculateArea* method implementation. When the database lookup succeeds, the interpreter calls the *customResultAvailable* method (5) and the service implementation continues by calculating the pixels for the requested area. Finally, the result future *ret* is provided with the area result object (6). The service proxy is informed about the result availability and again applies corresponding interceptors, e.g. for cloning the result value. Afterwards, the result of the call is made available to the caller (7) by invoking the *resultAvailable* method of the declared result listener. If any problems occur during the processing of the call, e.g. the call gets rejected by the reasoning interceptor, the algorithm lookup fails, or a network timeout occurs during the transfer of the result, the proxy instead calls the exceptionOccured method (7), thus informing the caller that the *calculateArea* call has failed.

5.3. Remote Communication

Agent applications are typically realized as distributed systems, such that the question arises how method call interaction fits to distributed computing. The asynchronous, proxy-based interaction

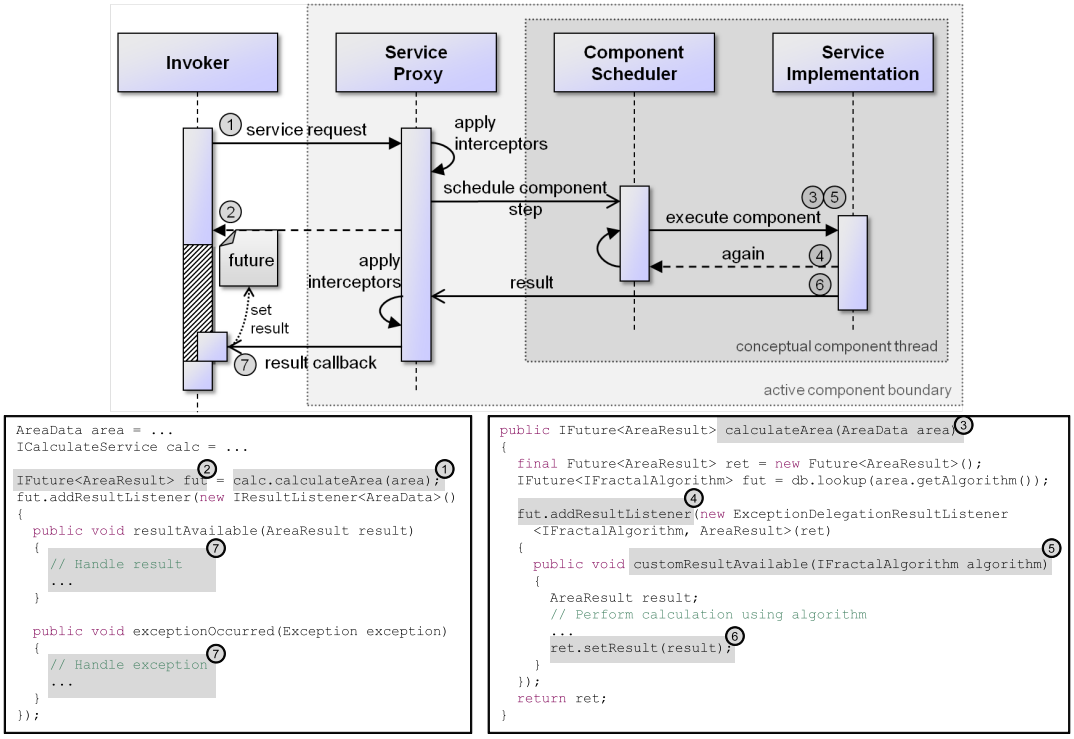

Figure 8. Invocation scheme (top) with example caller code (left) and service implementation (right)

scheme proposed in this paper already addresses many of the issues of distributed communication as it allows incorporating, e.g. failure-handling and timeout mechanisms. In the Jadex platform this is realized by a so called remote management service (RMS), which handles all service calls to agents on another platform.

The communication scheme is essentially the same as in remote method invocation (RMI) systems. First the data is marshaled. In Jadex, the codec for marshaling the data can be freely configured. As a default, an XML codec is provided, which is able to serialize arbitrary Java objects that follow the Java Beans specification [20]. After marshaling, the data and method call information is sent as a message (e.g. using the TCP-based message transport) to the RMS agent of the remote platform. The remote RMS unmarshals the data, retrieves the proxy for the service to be called and calls the appropriate method, which leads to the interceptor chain being executed. The result of the call is transferred back to the caller using the same (un)marshaling procedures.

In addition to dealing with the communication issues, the RMS also implements a configurable timeout mechanism. Besides a global timeout that as a default applies to all remote service calls, the developer can also define custom timeouts for a specific service or some of its methods by using the *@Timeout* annotation in the service interface. This allows realizing robust distributed applications that can recover from network failures or broken nodes.

For discovery of the available agents and their services, the Jadex platform includes a peer-to-peer awareness mechanism. When awareness is enabled, a Jadex platform will signal its existence to the environment and will detect other awareness-enabled Jadex platforms automatically. An agent that wishes to use a service can issue a search based on the required service interface. The distributed Jadex infrastructure will then query the available agents for their services. The found service proxies are then returned to the searching agent, such that it can select and invoke an appropriate service. In this scheme it does not matter, if the service providing agent is a local agent or resides in a remote platform, because the required communication is automatically and transparently handled by the RMS.

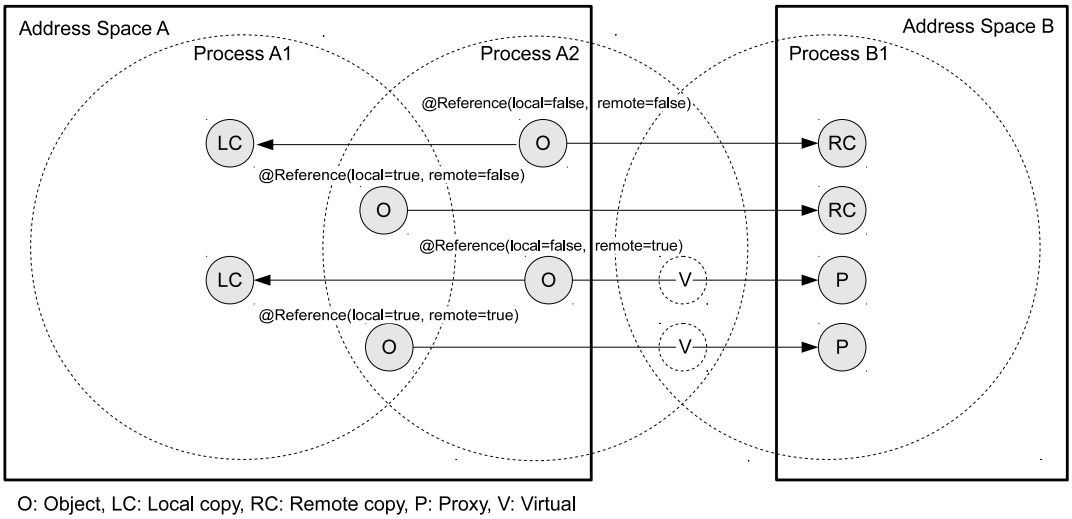

Figure 9. State object semantics

As already mentioned, method calls on agents involve two logically separated entities, the caller and the callee. To ensure that no state inconsistencies occur the states of both entities have to be protected. Besides avoiding concurrent access to the state of agent, it also has to be guaranteed that the states do not contain references to the same object. The general principle follows the actor idea [21] by separating completely the address spaces of different entities. This means that method calls have to apply call-by-value semantics for parameters and return value as default. In contrast to the general principle there are several cases in practice, in which copying values is unwanted. Examples are immutable objects which need not to be copied per definition and listener objects, which always have reference semantics. Moreover, copying parameters has a significant performance penality which should be avoidable in certain cases. Therefore, in Jadex programmers can control parameter passing semantics explicitly and in a fine granular way.

In Fig. 9 the different available parameter types are shown. To facilitate a fine-grained specification, local and remote reference semantics can be defined separately per data type (class) using the @Reference(local, remote) annotation. Furthermore, the data type specific properties can be overridden for parameters of specific methods by using the same annotation as part of the method signature. In the figure the different combinations of reference settings for local and remote cases and their meanings are depicted in detail. The first case shows the default, i.e. local and remote calls are treated with call-by-copy semantics. The object O in the middle belongs to process A2. A call on a method of process A1 in the same physical address space A leads to creating a copy of O. The same is true for a remote call to Process B1 in another address space B1. This default semantics is useful for all kinds of data objects that can be modified independently by the different processes (e.g. arrays, lists or maps). The copy semantics assures that e.g. changes to a list object in process A1 do not affect process A2, because each process owns a local copy of the list. The second case shows an object with local reference and remote copy semantics. In this case processes A1 and A2 share the object O, but a copy is created for a remote call from process B1. This semantics fits for immutable objects (e.g. Java basic types such as numbers and strings as well as custom data objects that do not provide any operations for modifying the objects state). Using local reference semantics for immutable objects does not affect the functionality of an application, but improves perfomance and reduces memory consumption, because the system does not need to manage multiple instances of identical objects.

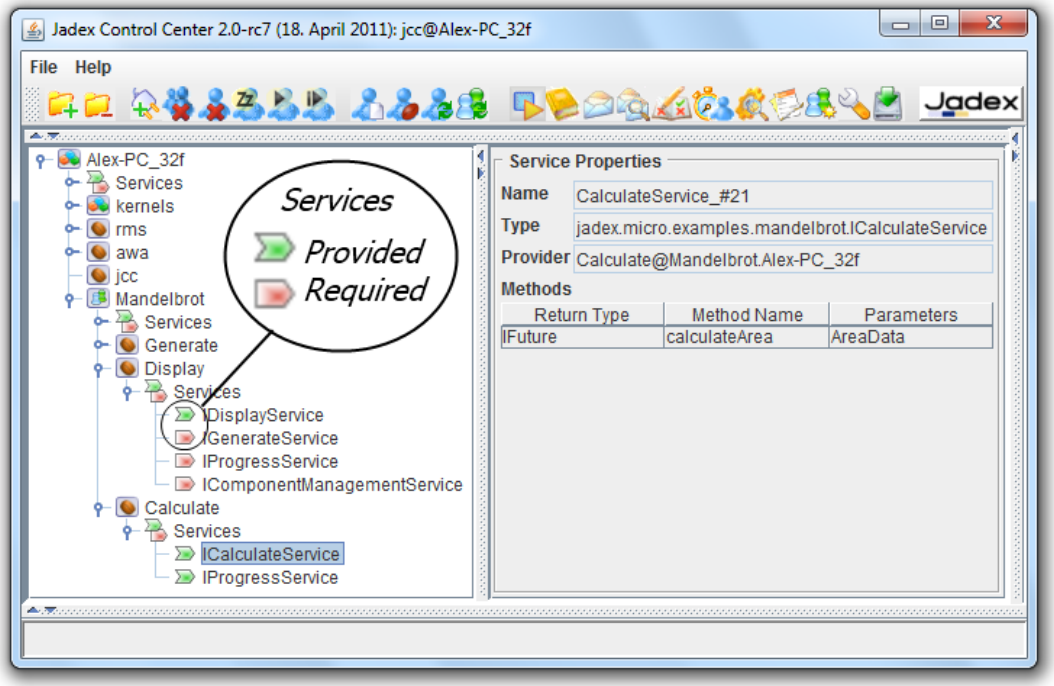

Figure 10. Agent services and methods in the Jadex Control Center

The last two cases have remote refences semantics, which means that a proxy of the parameter object is created on the remote site, so that method calls to the proxy can automatically be routed to the original object in another address space. The proxy realizes the concept of a virtually shared object, because it provides the impression that process A2 and process B can operate on a shared object, although these processes are in different address spaces. The third case with local copy semantics and remote reference semantics is included for completeness in the figure, but seems to have no obvious use in practice. The fourth case is local and remote reference semantics and fits to listeners that are applicable for local as well remote calls.

5.5. Tool Support

To support the developer in building service and method-based agent interactions, the administrative tools of the Jadex platform have been extended to incorporate this view. Figure 10 shows the Jadex Control Center (JCC) displaying the currently running agents (left). The agents are organized in a hierarchy starting from the platform (*Alex-PC_32f*) to the *Mandelbrot* application including the *Generate*, *Display* and *Calculate* agents. Besides these application agents, also system agents, e.g. *rms* and *awa* (for awareness) can be seen. The display and generate agents have been expanded in the tree to further inspect their services. Jadex distinguishes between provided services and required services. Provided services are those services, which are implemented in the agent as described in this paper. Required services allow specifying, which services of other agents an agent may want to access during runtime and thus enable some form of static checking, but agents are free to search and use additional services not specified as required. The *ICalculateService* of the calculate agent has been selected and the corresponding details are shown on the right. The *name*, *type* (Java interface), and *provider* (owning agent) are given as well as the method signatures, which are included in the service interface.

6. Related Work

In the multi-agent research community several extensions have been developed in order to cope with problems caused by message passing. One approach is the agents and artifacts (A&A) paradigm [22], which adds artifacts as new first-level concept to the conceptual toolset of an agent developer. The underlying idea is that artifacts can be used to represent tools and items agents work with. Agents can use artifacts in an object-oriented way by exploiting their usage interface, i.e. the paradigm faciliates constructing systems by separating message-based inter agent and method-based artifact communication.

Another way of managing complexities consists in abstracting from low level message passing by introducing new or using alternative programming concepts. In [10] goal-oriented interaction protocols have been used to shield a developer from communication details. For this purpose at the application level (protocol) goals are used for specifying the domain objective that should be achieved, e.g. the lowest acceptable price for a good and deadline. The goal also returns only domain relevant information, although a message-based communication between agents is carried out behind the scenes. Similarily, the idea of commitment-based interactions [23] also tries to abolish message details in favor of commitments of the interaction partners. These commitments can be used to dynamically and flexible send messages according to the mental states of the participating agents. Finally, the service and agent integration has been proposed in order to facilitate interaction between both kinds of entities [24]. For example, in the agent framework JIAC [25], agents communicate using services but under the hood a message-based metaprotocol is used for service negotiations between service provider and consumer.

Closely related to our approach is the active object concept [15]. Despite this similarity, active objects are not agents as they are completely controlled by externally requested tasks and thus do not exhibit flexible proactive behavior [3]. Active objects cleanly solve the problem of how to decouple service requester and provider using sound object-oriented techniques like futures and task schedulers. They do not consider how behavior interference can be resolved. One typical framework based on active objects combined with component ideas is Proactive [26].

Many actor approaches also claim being conceptually near to active objects as the share the foundations of independently executing entities. In spite of this similarity, the actor model is also based on message passing as communication scheme so that these approaches do not put foward new solutions how interaction can be simplified. One exception is AmbientTalk [27], which is a framework and programming language for ambient intelligence based on actors. AmbientTalk introduces the concept of far references for method calls on other actors. For these method calls similar consistency assurances concerning state and parameter passing are guaranteed as in our model. AmbientTalk does not tackle the issue of behavior interference.

Summarizing, the problems of solely message-passing-based communication are well known in the agent research community and different approaches have been developed to remedy the complexities by hiding them from the agent programmer. In addition, active object and actor framework have partially addressed method-based communication without tackling the problems stemming from internal and external behavior interferences.

7. Discussion

The goal of the work presented in this paper is to simplify the development of multi-agent systems by incorporating method calls as a well understood interaction mechnism that is easy to use for unexperienced as well as advanced programmers. It could be argued that using method calls instead of message passing is a step backwards, because the agent paradigm is considered a successor of the object-oriented paradigm. It is often claimed that agents communicate on a semantic level with messages that represent intentional speech acts [4] and message contents following an ontology that

formalizes the agent's world knowledge. On the other hand, method invocations are considered a tight, low-level coupling, not suitable for adaptive and autonomous agents.

On a closer look, these differences are not as fundamental as it may seem. In a clean object or service-oriented design, the interfaces between callers are defined in a way to support loose coupling and reusability. The operations (methods) of a service definition represent the intentions understood by the service provider and the data types used in the method parameters correspond to the ontology. Therefore, a method-call-based interaction may as well achieve the same level of semantic abstraction typically found in message-based agent systems. This only requires the interfaces to be designed accordingly for abstracting away from irrelevant implementation details and focusing on the intentions behind the interaction. It should be noted that this aim towards open and standards-based interfaces is a general scheme in distributed systems not unique to agents.

This does not mean that every complex kind of semantic agent interaction can be realized as well using method calls. It rather shows an alternative that is easier to use, yet sufficient for many kinds of agent interactions. Therefore method calls should not be seen as a replacement, but as a supplement to messages. Instead of starting each interaction design by thinking in terms of messages, the developer should focus on the intended purpose of the interaction and choose between methods or more complex message-based protocols consciously.

8. Conclusions

This paper has argued that method call semantics is an important concept for the design and implementation of modular systems. Method call interaction is supported by many contemporary programming paradigms such as components and services. In the agent area, method calls are usually prohibited as they are considered to violate the autonomy of agents.

The implications of method call interactions for agents have been analyzed and it has been identified that besides autonomy, also state consistency has to be taken into account. It has been shown that careful implementation based on existing object-oriented programming techniques such as futures allows method call interactions between agents without violating state consistency and autonomy, yet that the resulting solutions induce considerable overhead and new sources for errors. Therefore an integrated approach has been conceived that incorporates method call interactions into an agent programming framework. The approach is based on the active object concept and allows agents to expose services defined as object-oriented interfaces that include method signatures. Autonomy and state consistency is automatically preserved by so called interceptors that enable call decoupling for state consistency and reasoning about, e.g., call acceptance for autonomy. The interceptor chain approach is very flexible, as it allows separating reasoning for autonomy from the concrete business logic and also allows adding and removing interceptors at runtime and therefore dynamically changing the behavior how an agent reacts to service method calls.

The usefulness of the approach has been illustrated by the running example of calculating Mandelbrot images. Other applications that have been realized with the approach e.g. include an agent-based coordination system for disaster management, where ambulances and fire brigades are represented as agents that coordinate with each other using method calls. Future work will be targeted in two directions. On the one hand, the software engineering aspect of the approach will be further strengthened by developing methodical processes for designing method-call-based agent applications and realizing additional tools for e.g. monitoring and visualizing the method calls that happen between running agents. On the other hand, the technical implementation will be extended towards existing standards, e.g. from the web services area. Therefore, the RMS will be extended to support existing protocols such as SOAP for achieving interoperability with other web service enabled systems.

References

[1] S. Franklin and A. C. Graesser, "Is it an agent, or just a program?: A taxonomy for autonomous agents," in *Proceedings of the 3rd Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages (ATAL 1996)*, J. Müller, M. Wooldridge, and N. Jennings, Eds. Springer, 1997, pp. 21–35.

[2] M. Wooldridge and N. Jennings, "Intelligent Agents: Theory and Practice," *The Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, 1995.

[3] M. Wooldridge, *An Introduction to Multiagent Systems*, 2nd ed. Chichester, UK: Wiley, 2009.

[4] *FIPA ACL Message Structure Specification*, Foundation for Intelligent Physical Agents (FIPA), Dec. 2002, document no. FIPA00061. [Online]. Available: http://www.fipa.org

[5] T. Finin, J. Weber, G. Wiederhold, M. Genesereth, D. McKay, R. Fritzson, S. Shapiro, R. Pelavin, and J. McGuire, "Specification of the KQML agent-communication language – plus example agent policies and architectures," Tech. Rep. EIT TR 92-04, 1993.

[6] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi, "JADE - A Java Agent Development Framework," in *Multi-Agent Programming: Languages, Platforms and Applications*, R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, Eds. Springer, 2005, pp. 125–147.

[7] G. Rimassa, D. Greenwood, and M. E. Kernland, "The Living Systems Technology Suite: An Autonomous Middleware for Autonomic Computing," in *In Proceedings of the International Conference on Autonomic and Autonomous Systems (ICAS 2006)*, 2006.

[8] *Web Services Description Language (WSDL)*, World Wide Web Consortium (W3C), 2007. [Online]. Available: http://www.w3.org/TR/wsdl20/

[9] M. Winikoff, "Implementing commitment-based interaction," in *In Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, 2007, pp. 868–875.

[10] L. Braubach and A. Pokahr, "Goal-oriented interaction protocols," in *5th German conference on Multi-Agent System Technologies (MATES 2007)*. Springer, 2007.

[11] A. Pokahr and L. Braubach, "From a research to an industrial-strength agent platform: Jadex V2," in *Business Services: Konzepte, Technologien, Anwendungen - 9. Internationale Tagung Wirtschaftsinformatik (WI 2009)*, H.-G. F. Hans Robert Hansen, Dimitris Karagiannis, Ed. ÃŰsterreichische Computer Gesellschaft, 2 2009, pp. 769–778.

[12] H. Sutter and J. Larus, "Software and the concurrency revolution," *ACM Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[13] L. Braubach and A. Pokahr, "Addressing challenges of distributed systems using active components," in *Intelligent Distributed Computing V - Proceedings of the 5th International Symposium on Intelligent Distributed Computing (IDC 2011)*, F. Brazier, K. Nieuwenhuis, G. Pavlin, M. Warnier, and C. Badica, Eds. Springer, 2011, pp. 141–151.

[14] H. Baker and C. Hewitt, "The incremental garbage collection of processes," in *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*. New York, NY, USA: ACM, 1977, pp. 55–59.

[15] G. Lavender and D. Schmidt, "Active object - an object behavioral pattern for concurrent programming," in *Pattern Languages of Program Design 2*, J. Vlissides, J. Coplien, and N. Kerth, Eds. Addison-Wesley, 1996.

[16] A. Rao and M. Georgeff, "BDI Agents: from theory to practice," in *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS 1995)*, V. Lesser, Ed. MIT Press, 1995, pp. 312–319.

[17] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf, "Goal Representation for BDI Agent Systems," in *Proc. of (ProMAS 2004)*. Springer, 2005, pp. 44–65.

[18] A. Pokahr and L. Braubach, "Active Components: A Software Paradigm for Distributed Systems," in *Proceedings of the 2011 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2011)*. IEEE Computer Society, 2011.

[19] M. Fowler, "Inversion of control containers and the dependency injection pattern," 2004, http://martinfowler.com/articles/injection.html.

[20] G. Hamilton, *JavaBeans, Specification Version 1.01*, Sun Mircosystems, 1997.

[21] R. Karmani, A. Shali, and G. Agha, "Actor frameworks for the jvm platform: a comparative analysis," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '09. New York, NY, USA: ACM, 2009, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/1596655.1596658

[22] A. Ricci, M. Viroli, and A. Omicini, "The A&A programming model and technology for developing agent environments in MAS," in *Programming Multi-Agent Systems, 5th International Workshop (ProMAS 2007)*, M. Dastani, A. E. F. Seghrouchni, A. Ricci, and M. Winikoff, Eds. Springer Berlin / Heidelberg, 2007, pp. 89–106.

[23] J. Xing and M. Singh, "Formalization of commitment-based agent interaction," in *Proceedings of the 2001 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2001, pp. 115–120.

[24] M. Singh and M. Huhns, *Service-Oriented Computing. Semantics, Processes, Agents*. John Wiley & Sons, 2005.

[25] S. Albayrak and D. Wieczorek, "Jiac - a toolkit for telecommunication applications." in *Proceedings of the 3rd International Workshop on Intelligent Agents for Telecommunication Applications (IATA 1999)*, S. Albayrak, Ed. Springer, 1999, pp. 1–18.

[26] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. PÃl'rez, "Gcm: a grid extension to fractal for autonomous distributed components," *Annals of Telecommunications*, vol. 64, no. (1-2), pp. 5–24, 2009.

[27] T. Van Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. De Meuter, "Ambienttalk: Object-oriented event-driven

programming in mobile ad hoc networks," *Chilean Computer Science Society, International Conference of the*, vol. 0, pp. 3–12, 2007.