On the Encapsulation and Reuse of Decentralized Coordination Mechanisms: A Layered Architecture and Design Implications

Jan Sudeikat[1,2] and Wolfgang Renz[1]

[1]Multimedia Systems Laboratory, Hamburg University of Applied Sciences, Berliner Tor 7, 20099 Hamburg, Germany
Email: {sudeikat; wr}@informatik.haw-hamburg.de
[2] Distributed Systems and Information Systems, Computer Science Department, University of Hamburg, Vogt–Kölln–Str. 30, 22527 Hamburg, Germany
Email: 4sudeika@informatik.haw-hamburg.de

Abstract: The effective and reliable coordination of agent activities is a momentous problem for Multi–Agent System (MAS) developers. Particularly challenging is the *decentralized* coordination of agents that enables systems to exhibit self–organization. Natural phenomena typically serve as design metaphors and developers apply *Decentralized Coordination Mechanisms* (DCMs) that have been inferred from biological, physical or social systems. This paper addresses the utilization of DCMs as reusable software components.Current development practices give little guidance for DCM selection and force developers to manually design, implement and tune mechanism parameters *ad hoc*, leading to highly specialized algorithms. Here, we propose a layered software architecture that encapsulates DCMs in (multiple) coordination spaces. A generic, annotation–based interface allows to separate agent coordination from agent functionality, therefore enabling DCM reuse and facilitating application (re)designs, i.e. mechanisms exchange and parameter adjustments. Implications for development procedures are discussed and the application of the layered architecture is exemplified in a resource allocation case study.

Keywords: decentralized coordination, coordination mechanism reuse, feedback, multi-agent system, self–organization.

1. Introduction

Conceiving distributed software systems as Multi–Agent Systems (MAS) demands the effective coordination of agent activities. Within Agent–Oriented Software Engineering (AOSE), this is typically approached at design time by explicitly modeling organizational structures (commonly in terms of *roles* and *groups* (e.g. reviewed in [15]). The derived models describe the procedural agent activities as well as agent activity adjustments, e.g. role changes [17]. The coordination of agents is decided in early development phases and changing coordination schemes enforces considerable effort.

A virtue of agent–based software development, which is drawing increasing attention in MAS research, is its ability to support the creation of *decentralized* as well as *adaptive* application designs that adjust their organizational structure at run–time [2, 23]. These systems *self–organize*, i.e. agents interact locally but their concurrent, autonomous activities enable the rise of macroscopic system phenomena [23]. Prominent examples are macroscopic observable *artifacts*, e.g. the (shortest path) routes that are formed by ant–algorithms, or *phase changes* lead to sudden behavioral changes [18].

These macroscopic observable phenomena ca be traced back to feedback loops that are established between agents [27, 29] (cf. section 2). Due to the inherent difficulty to anticipate the dynamics of non–linear feedbacks, development teams typically take inspiration from well–known natural scenarios, i.e. biological, physical and social phenomena that serve as design metaphors (e.g. discussed in [12, 28]). Examinations of the internal workings of these sources of inspiration have lead to the definition of *decentralized coordination mechanisms* (DCMs) that serve as reusable design pattern [6].

The utilization of these means to the construction of feedback loops within MAS is largely unexplored. Developers are forced to decide for a coordination mechanism at early stages of development, and then either utilize a agent platform that natively support selected mechanism or implement them themselves.

In this paper, we discuss DCM mechanics and trace their functionality back to the publication and perception of agent local information. Based on this observation, we facilitate the software engineering usage of DCMs, namely the reuse and exchangeability of coordination schemes. A unifying framework is presented that allows to provide a library of predefined DCMs. These are accessed via a generic interface that allows the annotation of agent models with coordination meta–data, i.e. declarations of agent internals to be communicated or modified by of DCM–based communications. Upon declaration, the reactive functionality of DCMs is processed in the background, therefore facilitating the separation of concerns by detaching procedural agent functionalities from role/behavior changing dynamics. Software producing organizations can maintain libraries of coordination mechanisms to be used interchangeably.

This paper is structured as follows. In the next section software engineering approaches to the utilization of decentralized coordination mechanisms within self–organizing MAS development are reviewed. A layered architecture for mechanism encapsulation and reuse is presented (section 3), followed by an outline of a tailored development procedure (section 4). Finally, the reuse and exchange of coordination mechanisms is exemplified (section 5), before we give concluding remarks.

2. Decentralized Agent Coordination in Self–Organizing MAS Development

Decentralized, self–organizing processes are powerful means to guide MAS adaptivity by the establishment and maintenance of macroscopic observable, nonlinear phenomena, e.g. *phase changes* [23]. However, these phenomena challenge traditional *top–down* and *divide–and–conquer* design strategies [7, 8]. Therefore, simulation–based development procedures have been proposed [4, 7]. These approach the transfer of nature–inspired self–organizing phenomena to software applications by bottom–up development procedures. These comprise the (1) selection of an appropriate design metaphor (e.g. insect societies or market dynamics) and (2) maping their constituent parts to the actual application domain. Based on these early design choices, the actual coordination mechanisms are (3) selected and (4) implemented in a target agent architecture. Implementation comprises the tuning of system parameters to ensure the intended behavioral regimes.

Considerable effort has been spend to compare and classify DCMs, i.e. the computational means to the self–organizing coordination of agents [5, 6, 12, 23]. These rely on the decentralized establishment of feedback loops within agent populations [29]. Feedbacks require that agents perceive the MAS state (*interdependency level*) and adjust their behavior (*behavior adaption level*) according to the gained perceptions [28, 29]. Mechanism dynamics typically comprise a locality of interactions and allow perturbations, which are necessary ingredients for self–organizing processes. Interdependency-level components can be based on *direct interactions* of agents and indirect interactions that are *mediated* by MAS environments. Direct interactions between agents are based on public markers (e.g. tags associated to agents) or message / token exchange (mechanisms reviewed in [5]). Mediated mechanisms have been classified in [12]. A prominent example are computational fields (co-fields) [14]. Agents emit these fields in a virtual environment to coordinate agent activities. Agent architectures steer the *behavior adaption* and range from purely *reactive* mechanisms, to *adaptive*, *cooperative* and *generic* architectures (cf. [23, 29]).

2.1 Systemic Modeling of Coordination Mechanics

Systemic modeling notions [24], i.e. the concepts of causal links and feedback loops, have been applied to model requirements on self–organizing MAS dynamics [27] and design metaphor behaviors [28]. An adjustment of system dynamics Causal Loop Diagrams (CLDs) [24] was given in [20] and is utilized in figure 1 to exemplify how positive links can be refined by two DCMs, namely the exchange of *token* and the propagation of *computational fields* [5]. In this graph–based notation, nodes denote the accumulative values of system state variables, e.g. *agent role occupations* and *system property variables*. Edges between these nodes describe additive (+) and subtractive (-) causalities between their accumulative values. I.e. increases in system states cause increases of positively connected system states. Negative links indicate inverse contributions. Circular structures form feedback loops that are either balancing (B; odd number of negative links) or reinforcing (R; even number of negative links).

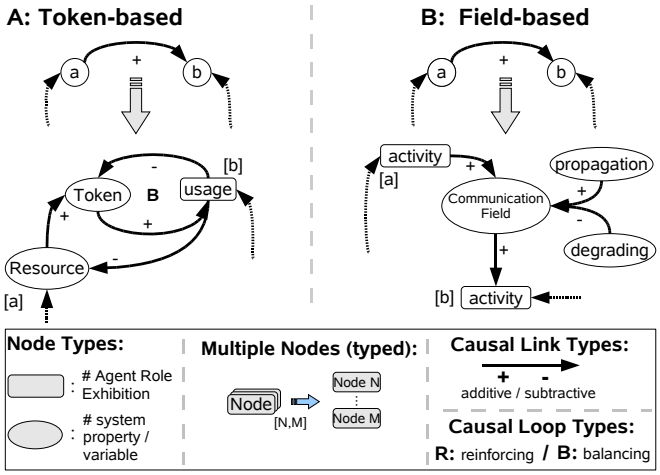

Fig. 1. Decentralized Coordination Mechanism dynamics.

Positive links between system states can be realized by distributing *tokens* (figure 1; A) that represent the availability of resources or access permissions (cf. [5]). The CLD–based notation denotes that the availability of resources positively influences the availability of tokens. These in turn are used by agents, i.e. transferred to agents (positive link) and consumed (negative link) by agent instances, therefore leading to a balancing feedback loop that limits the amount of tokens.

Positive links can also be realized by *computational fields* (cf. [5]), where sources are located in a virtual 2D space and sense the gradients of the spatially distributed fields which communicate application dependent information. The CLD–based notation (figure 1; B), certain activities lead to (positive link) the distribution of specific fields. The larger the field force is, the more agent notice the field and adjust their local behavior (positive link). Two mechanism internal properties influence field distributions as fields are (positively) propagated by the environment and (negatively) degraded to limit field sizes.

2.2 Coordination Environments

The coordination of agents within multi–agent systems is an essential topic for agent–based development approaches that lead to the development of MAS environment frameworks to facilitate agent coordination ([31]). These take inspiration from specific coordination metaphors / mechanisms and provide reusable, metaphor specific infrastructures as well as usage interfaces. In [9], it has been proposed to separate coordination contexts by providing multiple independent coordination environments.

3. A Layered Agent Coordination Architecture

In the following, we present a layered approach to the encapsulation, reuse and exchange of DCMs that provides a unifying implementation architecture and coordination interface. Coordination mechanisms are provided as services for inter–agent coordination to encourages the separation of agent coordination and agent functionality. Accessible via a generic interface, DCM mechanics are realized within dedicated coordination spaces.

3.1 Decentralized Coordination Mechanism Layers

Figure 2 gives a conceptual overview on the here proposed approach to the decentralized coordination of agents. In order to encapsulate specific coordination mechanisms, these are provided in a layer that is implemented on top of agent platforms. Agents that are to be coordinated have to include a module that mediates between the coordination layer and the agent internal reasoning. Agent architecture specific elements can be annotated with coordination information (cf. section 3.2) that control how modules send coordination information and modify agent due to coordination perceptions.

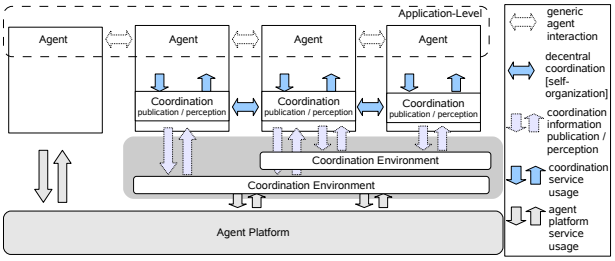

Fig. 2. A Layered approach to the encapsulation of decentralized coordination mechanisms. Each coordination mechanism instance provides a unique coordination space.

3.2 A Publish/Subscribe–based Coordination Interface

Enabling the reuse and exchange of DCMs demands the separation of the agent functionality from the coordination activities. This separation is realized by *annotating* agent models with the information which agent internal activities are to be published / triggered by perceptions. Mechanisms implementations provide two coordination operations, namely to *publish* and agent internal event and to *perceive* these information. *Publications* are triggered by agent internal events / activities. I.e. agents are enabled to communicate belief value changes or agent activities via the shared coordination spaces. Perceptions can be used to trigger agent internals, i.e. change belief values or trigger agent internal goals / activities. The intended separation is enabled by annotating what agents should automatically published (*register_publication*) or vice versa which activities should be automatically triggered by perceptions (*register_subscription*). Annotations can be read in at agent startup or registered at run–time.

When publications/perceptions are triggered is agent architecture dependent. While the decentralized communication of MAS environment properties is in principle possible, developers will typically be concerned with coordinating agent population members. I. e. to communicate *agent state changes* (e.g. belief updates, etc.) or *agent activities* (e.g. role occupations, plan executions, etc.).

DCMs communicate application dependent information via application independent distribution mechanisms (cf. section 2). The configuration of a DCM comprises the configuration of both the application dependent publications / perceptions as well as the configuration of the DCM internals (cf. figure 3). Application dependent *agent coordination configuration* describes the participation of agent instances in the shared coordination space that realizes DCM behavior. It consists of annotations to agent models that describe which agent internals are to be communicated (*publication configuration*) and which perceptions should trigger agent internal processing (*perception configuration*). DCM mechanics are application independent and are defined by a *communication configuration*. However, agent modules communicate application dependent information. The declaration of what is to be communicated is defined in so–called *parameter mappings* as part of the perception and publication configurations. In addition, developers may want to steer DCM properties by application dependent parameters. Therefore, developers can define the strength of *publication enforcement*, by mapping this DCM dependent parameter to either parameter values that are part of the communication (*communication parameter mapping*), agent belief values (*belief mapping*) or set it to *fixed* values.

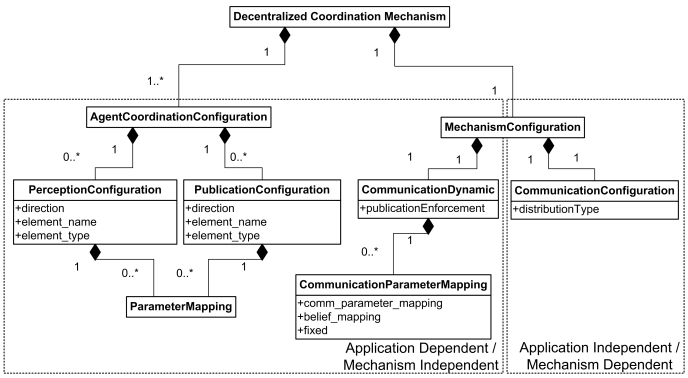

Fig. 3. The conceptual model of DCM configuration.

3.3 Realization

Figure 4 denotes the different architectural alternatives to the realization of coordination environments. First, fully *decentralized* designs can be conceived (figure 4; A). These emulate a commonly available coordination space solely by the exchange of messages between the participating agents and can either be directly implemented in communicating agents or can be separated from agent implementations by a network middleware (e.g. in the *TOTA* system [13]). Secondly, *centralized* architectures have been applied to provide coordination spaces (figure 4; B). These can either be based on (1) *dedicated agents* that are responsible to manage the access to coordination spaces and ensure their consistency, or on (2) specific *coordination platforms* that are deployed on networked servers. Prominent examples for coordination platforms are Linda style [32] *tuple spaces*. In addition specific agent–oriented infrastructures have been revised, e.g. *Cartago* [21, 22]. We coin these approaches centralized since they provide commonly accessed components. Actual realizations may duplicate servers, e.g. to increase robustness and fault tolerance. Thirdly, the coordination of agents may be *delegated* to another MAS (figure 4; C) where those agents have the sole purpose to function as proxies to provide coordination information. E.g. in [10] manufacturing control system elements are enabled to ”send out” mobile agents that distribute information in a common environment.

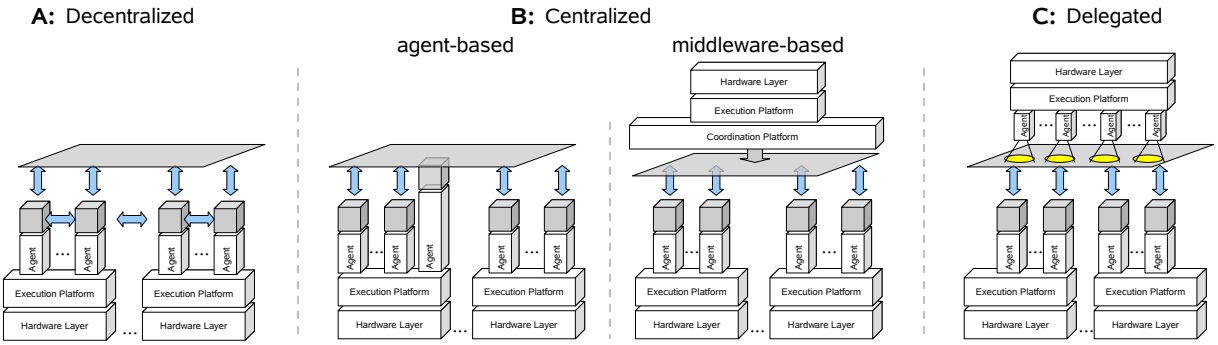

Fig. 4. Alternative implementations of decentralized coordination mechanisms. A: completely decentralized due to agent interaction; B: centralized by distinguished agents, or dedicated middleware services; C: delegation to another MAS instance.

In order to validate the practicability of the here proposed scheme for the separation of coordination and agent functionality, a prototype implementation has been revised that allows agents which are programmed and executed within the *Jadex* system[1] to use the previously described coordination interface. Jadex provides an execution environment and development tools that facilitate the development of *Belief–Desire–Intention* (BDI) style agents [19]. The BDI reasoning engine is conceived as an extension to arbitrary agent middleware and is freely available. Following the BDI model [19], beliefs describe the agent knowledge about their own state and its surrounding environment. Goals describe the states which agents try to pro–actively achieve and are typically defined in terms of belief values. Plans provide the executable means to achieve goals. Reactive reasoning is responsible to *deliberate* on the goals to pursue and to select appropriate plans for goal achievement via *means–end reasoning*.

Jadex agent implementations consist of two parts. So–called *Agent Definition Files* (ADFs) describe the structure of agent types. These comprise definitions of agent beliefs, goals and plans as well as events that occur during agent execution. Events indicate agent internal incidents or agent external influences, e.g. sending and reception of Agent Communication Language (ACL) messages. These declarations are made in XML[2] syntax. The executable agent activities, i.e. plans, are provided by ordinary Java[3] classes. The *capability* concept [3] has been proposed to modularize BDI agents. Jadex facilitates the encapsulation of clusters of beliefs, goals and plans supplemented by visibility rules [1]. Capabilities are defined in ADFs as well and can be included to add functionalities to agents.

Figure 5 gives an overview on the embedded realization of coordination interactions. In order to allow the supplement of agent models with coordination mechanisms, developers can reference a capability in agent models that encapsulate DCM implementations (subcapabilities), provide the previously described usage interface (section 3.2) and handle both DCM perceptions as well as DCM publications.

On agent start–up, the capability processes the meta–data (agent coordination configuration; cf. figure 3) that were annotated to agent models. Additional perception and publication configurations can be added at run–time (register publication / subscribe primitives; cf. table **??**). The annotations control agent activity that is *contributively* executed during agent operation. These contributive activities are executed via a so–called *co–efficient* extension to capabilities that have been introduced in [26]. According to this mechanism (cf. figure 5) the capability provides additional functionality by (1) registering and event–listener for its surrounding agent. This listener is (2) notified when publishable events occur and subsequently looks–up the type of publication that is to be communicated. The encapsulated DCM implementation is responsible to perform the actual communication. When DCM implementations receive publications, they look–up the type of agent internal reasoning event that has been associated and dispatch it. The associations of publications and perceptions (cf. section 3.2) allow for the declaration of parameter values that are mapped to (DCM–internal) communication event parameters to be transferred between agents.

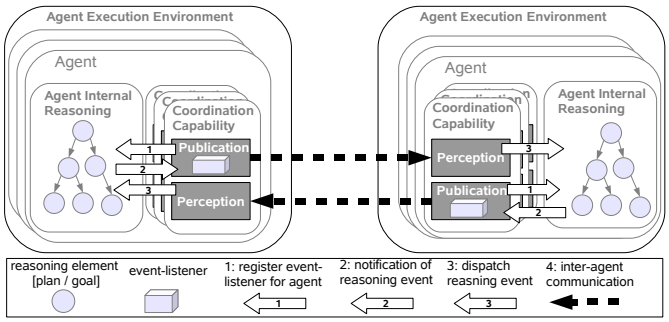

Fig. 5. Realization of agent interactions.

4. Designing Decentral Coordination

The strict separation of agent activity and agent coordination has has to be supported by MAS design and development practices. We propose two modeling levels to coexist and be subject to synchronized incremental refinement. Design models of the *agent coordination* address the dynamics with which agents collectively adjust their activities. These describe the causal links and feedback structures (cf. section 2) that drive the self–organizing dynamics. *Agent design* models are agent architecture specific and define the agent internal structure.

[1] http://vsis-www.informatik.uni-hamburg.de/projects/jadex/

[2] http://www.w3.org/XML/

[3] http://java.sun.com/

4.1 Modeling Agent Coordination and Agent Activity

MAS developers face the challenge to ensure both the correct functionality of individual agents as well as that agent coaction lead to the intended system functionality. When the subject of agent coactions are self–organizing dynamics, CLD–based models of MAS dynamics are appropriate [27]. The system states of these models are system properties as well as agent role / group occupation counts. It allows to explicitly model the causalities between agent activities as well as the resulting feedbacks that steer self–organizing behaviors [20]. Therefore, feedback structures become design artifacts themselves and serve as subjects to refinement procedures [28] in models of the *agent coordination*.

Since AOSE methodologies are typically tailored toward specific agent architectures [25] their modeling notions and procedures should be applied when a methodology matches the target agent platform. AOSE modeling notions then provide *agent design* models and guide agent implementations.

Both models are inherently synchronized since a subset of the nodes of the CLD–based coordination models denote agent role / behavior occupations. These behaviors that agents exhibit are defined in the agent architecture dependent AOSE models. The incremental refinement of these models is driven by the introduction of fine–grained (sub–) roles, (sub–)goals or (sub–)behaviors that detail system states. While the agent designs describe all agent activities, reference agent coordination models only the roles / activities that are caused by other agents. These causes may be direct inter–agent communication or mediated by environment perceptions. I.e. agent design models are concerned with the procedural internal workings of agents, while the coordination models provide a dynamic view–point on the observable agent activities.

4.2 Designing MAS by Causality Refinement

The development of self–organizing MAS is typically addressed by simulation–oriented, bottum–up development procedures (cf. section 2). however, development projects typically start from well–defined macroscopic system requirements. When the subjects of the MAS development are self–organizing behaviors, the intended feedback loops can be described by the above discussed agent coordination models. Since the here proposed architecture and usage interface (cf. section 3) provides the means to establish causal links between agent implementations, causal links in CLD–based models do not only serve as abstractions but can directly be implemented. The subjects that are to be coordinated are agent models that may be derived from top–down AOSE methodologies [11] or bottum–up development strategies [7].

In order to introduce top–down design techniques to self–organizing MAS development, we propose a procedure that is based on the incremental refinement of CLD–based MAS models. The requirements on a self–organizing application can be defined in CLDs that describe the causalities of macroscopic system states [27]. Based on the macroscopic observable feedback loops, designers can derive sets of design metaphors that map to the intended system behavior as discussed in [28]. These models are the subject to further refinements by detailing agent roles and system properties. Finally, the specific agent types are identified and the causal links between different agent types are identified. DCMs provide the means for establishing causalities between the roles of different agent types, while the different agent roles within a specific agent type are controlled with agent architecture internal techniques (e.g. goal/subgoal relationships or state machines). In section 5, a similar CLD refinement is exemplified that originates from the initial system requirements, introduces a nature-inspired design metaphor and finally identifies a causal link between agent types that is to be realized by applying DCMs. Their selection is guided by systemic models that allow to infer dynamic properties of DCMs (cf. section 2.1) Finally, the initial requirements – given in CLD notation – can be validated by comparing the behavior of (prototype) MAS implementations with CLD animations, using statistical methods [27, 30].

5. Case Study

Here, we consider how a *dynamic allocation* strategy that is inspired by the foraging behavior in *honey bee* societies can be added to agent implementations in a simplified resource allocation scenario. Following the application setting that has been examined in [16], we consider a cluster of *servers* that host web services. Since the external usage of services (as websites, etc.) is expected to vary considerably, it is intended to allow servers to adjust their allocations on the fly to met the changing demands on the fly. In addition, a decentralized coordination is required to facilitate scalability and robustness.

Initially, the basic agent functionalities are conceived. Servers are able to (1) *handle requests* and (2) *(re–)configure*, i.e. can alternate between different services offers. Jadex agents represent servers and incoming requests are mimicked by ACL messages. Servers register their availability as the agent platform *domain facilitator* and (re–) configurations modifying these registrations.

Figure 6 (A) shows the expected application behavior in a CLD–based notation (cf. section 2). System external *requesters* make requests (*jobs*) that are answered by *serving teams*. These teams are composed of individual servers that are members of a team or are *unbound*, i.e. do not answer requests. The dynamic allocation strategy to server teams relies on two balancing feedback loops. The amount of jobs is to be balanced by corresponding team sizes (α) as well as the amount of bound / unbound servers (β).

Insect–inspired application designs metaphors, e.g. the utilization of *honey bee foraging* strategies [16], can guide the realization of these feedbacks. Bee societies rely on so–called *scouts* that wander an environment and report encounters of *resources* that may be foraged. The availability of resources, along with locally conceivable quality measures (distance to nest, quality of nectar, etc.) is communicated via *waggle dances* which cause *foraging* bees to exploit resources. Individual foragers are free to switch to more beneficial resources when communicated. A corresponding refinement (figure 6; B) identifies *scouting-servers* (scouts) and *associated servers* (foragers) as members of serving teams. Scouting servers *recruit* servers to be associated to certain request types by communicating resources.

Following this design metaphor, developers have to decide how to realize these recruitments. In order to show the reuse and exchangeability of DCMs, we discuss how *token–based* as well as *field–based* mechanisms can be applied to realize the same causal link.

Using a Token–based DCM, tokens represent the availability of resources and are distributed by the scouting servers. Coordination annotations control when tokens are to be distributed, separating the mechanism-based interactions from the mechanisms dependent implementation details. These details can therefore be adjusted without changing the actual agent implementation. In our simulations, tokens have been represented by native Java objects that are randomly distributed via ACL messages.

Field–based mechanisms can be similarly mapped to the application domain. I.e. agents are arranged in a (virtual) grid and scouting servers emission computational fields represent the availability of resources. These field have been mimicked by a centralized Cartago[4] workspace [21]. The Cartago system implements the *Agents and Artifacts* (A&A) coordination model that is inspired by human cooperative working environments [22]. So–called *artifacts* serve resemble objects, resources and tools that agents can manipulate. Each computational field is represented by an artifact that coordination capabilities constantly observe and examine whether they are within field range.

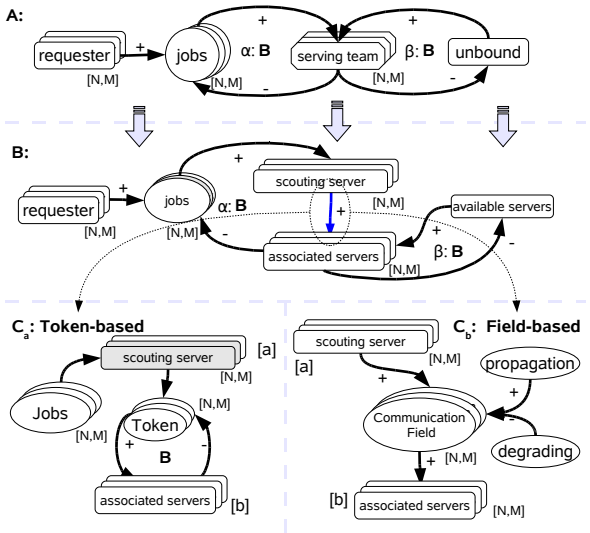

Fig. 6. Allocation dynamics. System requirements (A) are refined to a insect–inspired design methaphor (B). Different DCMs can be utilized to realize the causal link between scouting and associated servers (C).

Agent implementations can utilize these DCMs by declaring the publication and perception of DCM–based interactions. These annotations are DCM independent. They provide an interface between the application–dependent agent properties to the published / perceived and the applied DCM. Listing 1 shows the annotation that defines that the reception of a (typed) perception causes the dispatchment of a specific goal type (”change_allocation”). An optional parameter mapping allows to forward parameters of the perceived interaction, i.e. here the type of request that is advertised by the publisher. The attribute name indicates the name of the parameter of the referenced BDI element and the ref attribute is a reference to a DCM internal, unique parameter identifier. Similarly publications can be indicated by changing the direction attribute.

The behavior of the DCMs is also prescribed by XML declarations. Following the previously given meta–model (MechanismConfiguration; cf. figure 3) the DCM specific internal workings (CommunicationConfiguration) as well as (application dependent) dynamic properties of the interactions (CommunicationDynamic) can be defined. I.e. the specification of communication dynamics has been used to map the strength of communication to a publication value. E.g. the confidence (application dependent integer value) that scouts have about the need for reinforcement for a specific requests type is mapped to the ranges of corresponding force fields.

Listing 1. Coordination Annotation: Denoting the publication of request types. An extract from a Jadex ADF shows a goal declaration. A coordination annotation declares that the goal is to be dispatched when perceptions are received.

```
<agent>
        ...
        <goals>
                ...
                <!-- Agent Behavior: Change server allocation . -->
                <achievegoal name="change_allocation">
                        <parameter class="String" name="service_type"/>
                </achievegoal>
                ...
        </goals>
        ...
</agent>

< agent_coordination_configuration >
    <perceptions>
        <perception type="request_type" element_type="GOAL"
                element_name="change_allocation" direction ="PERCEPTION">
            <parametermapping>
                <parametermapping ref="service_type" name="service_type"/>
            </parametermapping>
        </perception>
    </perceptions>
</ agent_coordination_configuration >
```

6. Conclusions

In this paper, we proposed the encapsulation and reuse of decentralized coordination mechanisms (DCMs) that provide field–tested means to the construction of self–organizing dynamics within MAS. While being inspired from interdisciplinary biological, physical and social systems can their utilization in MAS development be attributed to the ability to locally communicate application dependent information. A layered architecture has been outlined that provides coordination mechanism as services for agent coordination. A corresponding usage interface allows to separate agent functionality from the agent coordination and therefore facilitates software engineering practices, i.e. mechanism reuse, exchange and redesign. I.e. agent implementations can be annotated with coordination information. The implications of the strict separation of agent models from coordination mechanisms has been discussed and an incremental development procedure has been presented.

[4] http://www.alice.unibo.it/xwiki/bin/view/CARTAGO/

Future work will examine the utilization of causal links and feedback structures as design and development artifacts. The utilization of a dedicated coordination language promises to automate the annotation of agent models and guide the validations of DCMs usage via system simulations [27, 30]

Acknowledgments

One of us (J.S.) would like to thank the *Distributed Systems and information Systems* (VSIS) group at Hamburg University, particularly Winfried Lamersdorf, Lars Braubach and Alexander Pokahr for inspiring discussion and encouragement.

References

[1] L Braubach, A Pokahr and W Lamersdorf, Extending the capability concept for flexible BDI agent modularization. Proc. of PROMAS-2005 2005.

[2] S Brueckner and H Czap, Organization, self-organization, autonomy and emergence: Status and challenges. International Transactions on Systems Science and Applications, Vol. 2, No. 1, 2006, pp. 1–9.

[3] P Busetta, N Howden, R Rönnquist and A Hodgson, Structuring BDI agents in functional clusters. ATAL '99 2000, Springer, pp. 277–289.

[4] T DeWolf and T Holvoet, Towards a methodolgy for engineering self-organising emergent systems. Proceedings of the International Conference on Self-Organization and Adaptation of Multi-agent and Grid Systems 2005.

[5] T DeWolf and T Holvoet, A catalogue of decentralised coordination mechanisms for designing self-organising emergent applications, Tech. Rep. Report CW 458, Department of Computer Science, K.U. Leuven, 2006.

[6] T DeWolf and T Holvoet, Decentralised coordination mechanisms as design patterns for self-organising emergent applications. Proceedings of the Fourth International Workshop on Engineering Self-Organising Applications 2006, pp. 40–61.

[7] B Edmonds, Using the experimental method to produce reliable self-organised systems. Engineering Self Organising Sytems: Methodologies and Applications 2004, no. 3464 in LNAI, pp. 84–99.

[8] B Edmonds and J J Bryson, The insufficiency of formal design methods - the necessity of an experimental approach for the understanding and control of complex mas. AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems 2004.

[9] A Gouaich and F Michel, Towards a unified view of the environment(s) within multi-agent systems. Informatica (Slovenia), Vol. 29, No. 4, 2005, pp. 423–432.

[10] K Hadeli, P Valckenaers, C Zamfirescu, H V Brussel, B S Germain, T Hoelvoet and E Steegmans, Self–organising in multi–agent coordination and control using stigmergy. Engineering Self-Organising Systems 2004, LNCS, Springer Berlin / Heidelberg, pp. 105–123.

[11] B Henderson-Sellers and P Giorgini, Eds., Agent-oriented Methodologies, Idea Group Publishing, 2005, ISBN: 1591405815.

[12] M Mamei, R Menezes, R Tolksdorf and F Zambonelli, Case studies for self-organization in computer science. J. Syst. Archit., Vol. 52, No. 8, 2006, pp. 443–460.

[13] M Mamei and F Zambonelli, Programming stigmergic coordination with the tota middleware. AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems New York, NY, USA, 2005, ACM, pp. 415–422.

[14] M Mamei, F Zambonelli and L Leonardi, Co–fields: A physically inspired approach to motion coordination. IEEE Pervasive Computing, Vol. 03, No. 2, 2004, pp. 52–61.

[15] X Mao and E Yu, Organizational and social concepts in agent oriented software engineering. Agent-Oriented Software Engineering V, 5th International Workshop, AOSE 2004, Revised Selected Papers 2004, Vol. 3382 of LNCS, Springer, pp. 1–15.

[16] S Nakrani and C Tovey, On honey bees and dynamic server allocation in internet hosting centers. Adaptive Behavior, Vol. 12, No. 3-4, 2004, pp. 223–240.

[17] J J Odell, H V D Parunak, S Brueckner and J Sauter, Temporal aspects of dynamic role assignment. Agent-Oriented Software Engineering IV 2004, Vol. 2935/2003 of LNCS, Springer, pp. 185–214.

[18] H V D Parunak, S Brueckner and R Savit, Universality in multi-agent systems. AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems 2004, IEEE Computer Society, pp. 930–937.

[19] A S Rao and M P Georgeff, BDI-agents: from theory to practice. Proceedings of the First Int. Conference on Multiagent Systems 1995.

[20] W Renz and J Sudeikat, Modeling feedback within mas: A systemic approach to organizational dynamics. Proceedings of the International Workshop on Organised Adaptation in Multi–Agent Systems 2008, to appear.

[21] A Ricci, M Viroli and A Omicini, Cartago: A framework for prototyping artifact-based environments in mas. Environments for Multi-Agent Systems III 2007, D Weyns, H V D Parunak, and F Michel, Eds., Vol. 4389 of LNCS, pp. 67–86.

[22] A Ricci, M Viroli and A Omicini, Give agents their artifacts: the a & a approach for engineering working environments in mas. AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems New York, NY, USA, 2007, ACM, pp. 1–3.

[23] G D M Serugendo, M P Gleizes and A Karageorgos, Self–organisation and emergence in mas: An overview. Informatica 2006, Vol. 30, pp. 45–54.

[24] J D Sterman, Business Dynamics - Systems Thinking an Modeling for a Complex World, McGraw–Hill, 2000.

[25] J Sudeikat, L Braubach, A Pokahr and W Lamersdorf, Evaluation of agent–oriented software methodologies - examination of the gap between modeling and platform. Agent-Oriented Software Engineering V, Fifth International Workshop AOSE 2004 2004, pp. 126–141.

[26] J Sudeikat and W Renz, Monitoring group behavior in goal–directed agents using co–efficient plan observation. Agent-Oriented Software Engineering VII, 7th International Workshop, AOSE 2006, Hakodate, Japan, May 8, 2006, Revised and Invited Papers 2006.

[27] J Sudeikat and W Renz, On expressing and validating requirements for the adaptivity of self–organizing multi–agent systems. System and Information Sciences Notes, Vol. 2, No. 1, 2007, pp. 14–19.

[28] J Sudeikat and W Renz, Toward systemic mas development: Enforcing decentralized self–organization by composition and refinement of archetype dynamics. Proceedings of Engineering Environment–Mediated Multiagent Systems 2007, LNCS, Springer.

[29] J Sudeikat and W Renz, Applications of Complex Adaptive Systems, IGI Global, 2008, ch. Building Complex Adaptive Systems: On Engineering Self–Organizing Multi–Agent Systems, pp. 229–256.

[30] J Sudeikat and W Renz, A systemic approach to the validation of selforganizing dynamics within mas. 9th International Workshop on Agent Oriented Software Engineering 2008, to appear.

[31] M Viroli, T Holvoet, A Ricci, K Schelfthout and F Zambonelli, Infrastructures for the environment of multiagent systems. Autonomous Agents and Multi-Agent Systems, Vol. 14, No. 1, 2007, pp. 49–60.

[32] G C Wells, A G Chalmers and P G Clayton, Linda implementations in java for concurrent systems: Research articles. Concurr. Comput. : Pract. Exper., Vol. 16, No. 10, 2004, pp. 1005–1022.