Grounding Social Interactions in the Environment *

Florian Klein** and Holger Giese

Software Engineering Group, University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany
fklein@upb.de, hg@upb.de

Abstract. While agents and environments are two intimately connected concepts, most approaches for multi-agent development focus on the agent-specific part of the system, whereas the handling of concerns related to the environment is often neglected or delegated to implementation level constructs. In this paper we demonstrate that building on an environment specification with expressive semantics is instrumental in designing agents that are capable of flexible and complex interactions. We propose a modeling approach that allows describing the concrete aspects of a multi-agent system as well as its conceptual and cognitive aspects within a single coherent conceptual framework by grounding all aspects in the environment. This framework enables an efficient development process built around the rapid prototyping and iterative refinement of multi-agent system specifications by applying model-driven design techniques to the system in its entirety.

1 Introduction

Agents and environments are two intimately connected concepts. Most widely accepted definitions juxtapose agents with the environment they sense and act on. Wooldridge and Jennings, for example, propose the following as the most basic definition of an agent: 'An agent is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives.' (emphasis theirs) [1]. However, little attention has traditionally been paid to the environment per se, which many approaches dealing with cognitive agents see as a mere stage on which the agents' intelligent behavior unfolds, in essence a necessary evil that should be treated as abstractly as possible.

Consequently, most approaches focus the analysis and design process on the agent-specific part of the system, whereas the handling of concerns related to the environment is delegated to agent frameworks, middleware, and other implementation level constructs. The environment's impact on the cognitive aspects of the system is thus not directly addressed in the model, but mostly implied in the available interfaces.

Research in the area of reactive agents emphasizes the situatedness of agents and generally pays closer attention to the environment and the way agents perceive and affect it. Despite the central role of the environment, however, the philosophy that 'the

* This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

** Supported by the International Graduate School of Dynamic Intelligent Systems.

world is its own best model' [2] basically places it outside the scope of explicit modeling. Again, the implications of the environment for the cognitive level are not explored on a theoretical, more abstract level.

Only recently, the idea to view the environment as a first-order abstraction [3] has begun to gain acceptance in the agent community. Especially through work on stigmergy [4–6], the idea that using the environment offers great potential for the efficient coordination and control of multi-agent systems has been established.

We believe that building on an environment specification with expressive semantics is instrumental in designing situated cognitive agents that are capable of flexible and complex interactions. Only an adequate exposition of the environment at the modeling level provides a generic mechanism allowing cognitive agents to make effective use of the environment.

Towards this goal, we make two important contributions, extending our previous work [7, 8] towards a complete theory and comprehensive methodology:

1. At the conceptual level, we propose a modeling approach that allows describing the concrete aspects of a multi-agent system, i.e. sensing and acting in physical and virtual environments, and its conceptual aspects, i.e. communication, coordination and social structure, within a single coherent conceptual framework by grounding all aspects in the environment.
2. As a practical result, our approach allows applying model-driven design techniques such as automatic code generation and formal model analysis techniques to the system in its entirety, enabling an efficient development process built around the rapid prototyping and iterative refinement of multi-agent system specifications.

At the heart of our conceptual contribution is the set of abstract principles that shape the proposed conceptual framework. While these are independent of any specific notation and formalism, we chose to extensively build on established software engineering techniques for our practical work. We use UML-based diagrams as an accessible graphical formalism to specify both structure and behavior. The formal, operative semantics required for rapid prototyping and verification of our system are provided by the theory of graph transformation systems.

Returning to the basic idea of an agent interacting with an environment through sensors and effectors, our modeling approach starts from an – essentially object-oriented – model of the entities an agent could potentially interact with and the sensors and effectors available for interaction, comprising both structural and dynamic aspects. As a second layer of abstraction, we add services that the environment provides to the agents to the specification. Services are described in terms of and provided through entities, and thus transparently integrated into the environment. Reusable templates allow the quick incorporation of common services or agent frameworks. As the last level, we add a conceptual layer defining social structures and coordination mechanisms. Inspired by the way human interaction works, social rules act on properties and behavior that are observable, as they would otherwise be neither realizable nor enforceable. Nonetheless, the social context may provide conventions for the interpretation of such observations. These allow the derivation of high level concepts such as commitments or group membership, which are useful for more sophisticated reasoning and yet grounded in the observable environment.

Due to the principle of grounding, the environment model plays an important role at any level. As a side effect, exposing the environment to the agents at the model level enables flexible behavior and makes the agents' interactions with and *through* the environment amenable to formal analysis and verification. Finally, grounding combined with the operative semantics of the employed specification technique ensures that all aspects of the system can be operationalized.

The proposed way of modeling does not impose a specific process model, even though the hierarchic layering implies certain dependencies. In order to explore the approach's potential, we present our vision for a model-driven design process built around prototyping and iteration. This design process comprises four phases (see Fig. 1):

Fig. 1. Overview of the proposed process

1. In the *analysis phase*, the environment and the overall requirements are modeled.
2. In the *social design phase*, the requirements are assigned to social structures; roles and norms fulfilling them are defined; and the required services are added to the environment. Formal verification and experimental validation using rapid prototyping techniques allow the evaluation and step-wise improvement of the design at this early stage.
3. In the *agent design phase*, the actual agents are implemented, respecting the constraints of the social specification. The agents can then be evaluated and optimized for performance, again using generated prototypes and a simulated environment.
4. In the *deployment phase*, the agents are tested in their production environment. This requires replacing the implementations of those services, sensors and effectors directly interfacing with the physical environment, but leaves all other aspects of the specification unchanged.

We are currently implementing the tools and frameworks required for bringing this vision to life within the scope of a student project. As our example for validation purposes, we are using an automated warehouse.

Section 2 provides a short introduction to the notation. The proposed conceptual framework is discussed in Sect. 3. The process is treated in more detail in Sect. 4. We present our intermediate results in Sect. 5, followed by a review of related work and an outlook on future work.

2 Foundations

For specifying the structural and behavioral aspects of the system, our approach employs UML-based notations. We chose them because visual specification languages provide an accessible and intuitive way of modeling and because this allows us to build on existing tools and practices from object-oriented software engineering.

We basically use UML class and object diagrams for the structure and employ story patterns, an extended type of UML object diagrams based on the theory of graph transformation systems (cf. [9]), for expressing structural changes and properties.

We provide a formal semantics for the employed concepts that are typically missing from UML-based notations by mapping them to a formal model based on the theory of graph transformations (cf. [10, 11]), which serves as the basis for code generation and formal verification.

2.1 Class and Object Diagrams

Class diagrams are the most fundamental UML concept employed in our approach. They allow describing the underlying structure of the problem and solution domain using classes and their relations. Class diagrams have been successfully employed to model complex structures and relationships in the context of software systems. Furthermore, we employ them to describe the physical environment, using attributes to capture physical characteristics such as mass, position, or velocity.

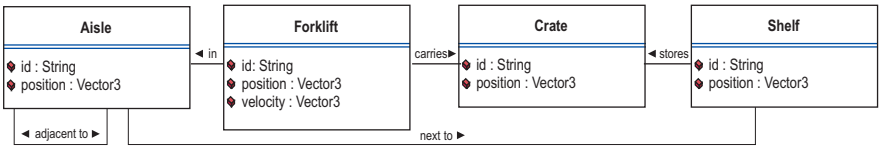


Fig. 2. Elements of a simple warehouse

In Fig. 2, an example consisting of a forklift, crates, aisles, and shelves is represented as a class diagram.

Object diagrams can be used to depict specific configurations of objects which are valid instances of a given class diagram. This can be employed for describing the topology or the initial configuration of a system, e.g., laying out a warehouse floor and populating the shelves with crates (see Fig. 3).

2.2 Story Patterns

Behavioral aspects and invariants of the system under consideration can then be modeled using *story patterns*. In general, story patterns specify two instance situations, a precondition (left hand side) that needs to be fulfilled before applying the pattern and a postcondition (right hand side) that is fulfilled after the pattern is applied. We first

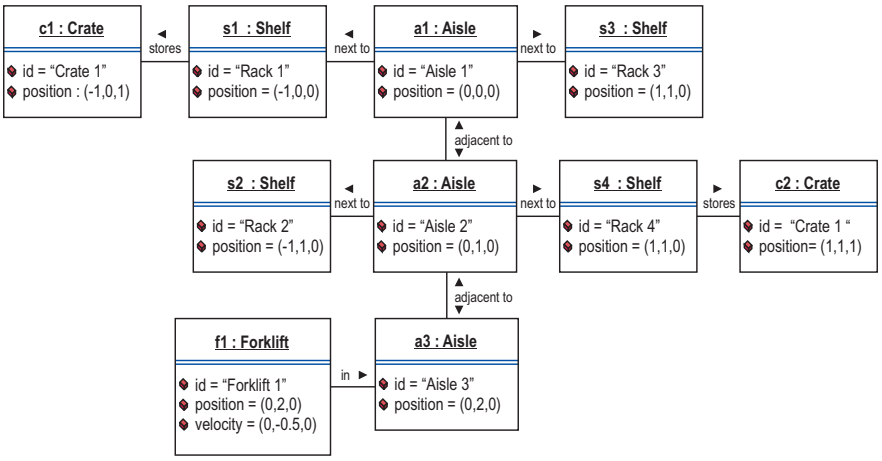
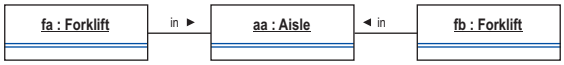
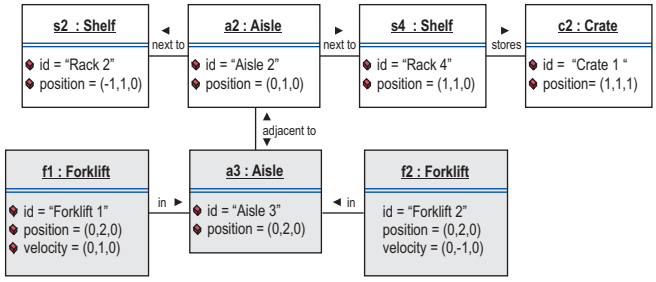


Fig. 3. A simple warehouse layout

look into *invariant story patterns*, where only the left hand side is present and the pattern simply states that the situation should always (positive invariant) or never (negative invariant) hold.



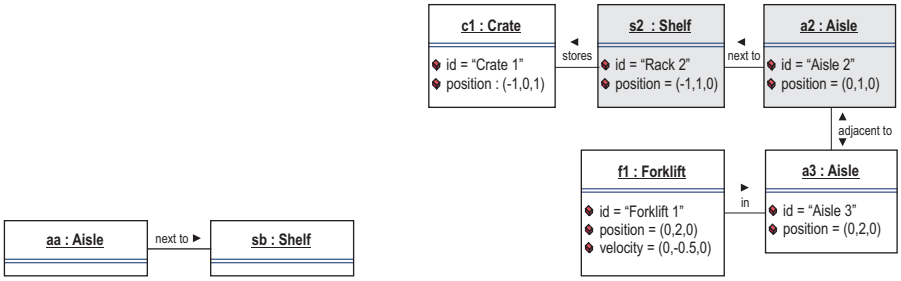
(a) Specification of a negative invariant



(b) Forbidden situation: Match for the invariant (shaded)

Fig. 4. Invariant: Two forklifts may not occupy the same space

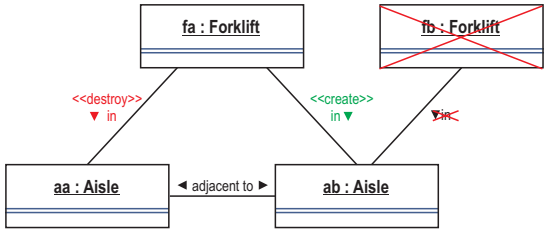
In Fig. 4, we model the constraint that two forklifts may not occupy the same space in an aisle as a negative invariant, an elementary guarantee upheld by the laws of physics. The positive invariant that every shelf is accessible, i.e. next to, an aisle is specified in Fig. 5.



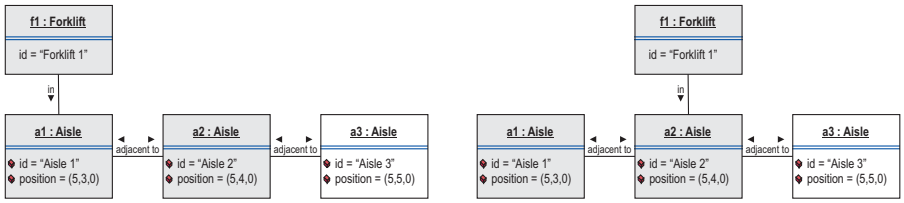
(a) Positive invariant (b) Valid situation: Invariant matches (shaded)

Fig. 5. Invariant: Every shelf is accessible from an aisle.

Forbidden elements may be indicated in story patterns by crossing them out. They can be employed to describe rules that only match when no match for any one of their forbidden elements is found, which allows specifying more complex rules.



(a) Pattern specification



(b) Matching the precondition (c) Applying the postcondition

Fig. 6. A forklift moves to an aisle, provided it is unoccupied.

If we also provide a right hand side for a rule, we can use it to describe behavior. The right hand side may be integrated into the left hand side by using the stereotypes ≪create≫ and ≪delete≫ for denoting elements which should be created resp. deleted as a side-effect of the rule, which yields a compact representation. An example of such a rule is depicted in Fig. 6, where a forklift moves to another aisle.

2.3 Tool Support

The open source UML CASE tool Fujaba [1] offers state of the art support for the modeling of class diagrams, object diagrams, and story patterns and thus enables model-driven development based on the outlined set of UML concepts.

The UML concepts introduced above, notably class diagrams and object diagrams, and the proprietary extensions such as story patterns (cf. Fig. 2, 3, 4(a), and 6(a)) have been given a formal semantics based on the theory of *graph transformation systems* (GTS).

This formal semantics enables us to provide sound code generation for these models, which is both useful for deriving simulation prototypes from early models (cf. [9]) and for generating a correct implementation of the production system from the final model that does not introduce implementation errors. Currently, the code generation of Java and C++ source code is supported for all employed diagram types.

Given the formal semantics, we can further employ formal techniques to validate and verify a given model, which is supported from inside the Fujaba CASE tool.

A first option is the GTS model checker GROOVE [12]. GROOVE imports GTS specifications and computes all reachable states of the transformation system, optionally bounded by the occurrence of a forbidden graph. With the help of a converter plug-in, we are able to export models from Fujaba to GROOVE's input format, check them, and visualize identified counterexamples in Fujaba. However, this approach can only be employed to verify finite state models with known initial configurations.

In order to be able to check invariant properties for the more general case of infinite state systems as well, we developed an invariant checker [11, 13] which exploits the local character of the graph transformation rules to perform a fully automatic check whether a given set of properties represent an inductive invariant of the system.

3 Approach

We now introduce the elements of the conceptual framework underlying our approach in more detail.

The notion of agents interacting with an environment, be it digital or physical, through sensors and effectors is central to our approach. We therefore make a clear distinction between concrete entities that agents can perceive and/or manipulate directly and conceptual entities that only exist virtually. Conceptual entities are explicitly derived from concrete entities by convention.

The concrete part of the model is predominantly descriptive in nature. Of course, design decisions do have a profound impact on the model, as the choice of sensors and effectors provided to the agents constrains what can be expressed. However, agents in the implemented system can immediately interact with concrete entities, even in heterogeneous open systems. The conceptual part of the model, on the other hand, is engineered deliberately, with the system's design objectives in mind. The way that conceptual entities are grounded in the concrete entities is *not* immediately visible to the agents. In order to allow an agent to interact with the system, this knowledge needs

[1] http://www.fujaba.de

to be made explicitly available to the agents or implied in their implementation. This problem is also touched on by [3], who distinguish between *natural* and *arbitrary* protocols and observe that the more natural protocols are, the easier ensuring interoperability becomes.

In order to structure the design and treat these different concerns separately, we divide the model into submodels that are layered on top of each other.

The environment model describes the environment, the concrete entities it contains, and their behavior. It also specifies the agents (which are themselves concrete entities) with their sensors and effectors.

The service model describes the infrastructure and protocols provided to the agents by the environment. As services are provided and accessed through concrete entities, the service model is predominantly concerned with the concrete parts of the system.

The social model introduces social structures, roles and norms along with the conventions required to connect them to elements of the more concrete layers.

3.1 Environment Model

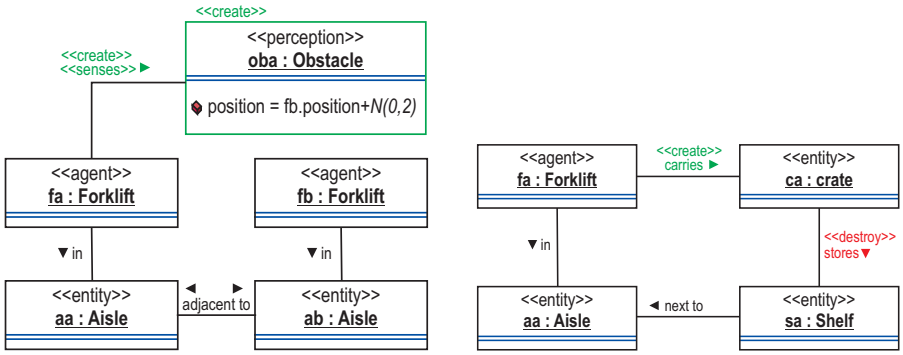
In the *environment model*, we want to describe all concrete *entities* and environment processes as they are relevant to the agents. We try to model the entities as 'objectively' as possible, i.e. as they are, not as the different agents perceive them, however. Concrete entities can be physical – these entities need to be simulated while prototyping and are later provided by the physical environment – or digital – which means they need to be implemented in software both in the prototypes and the production system. The entities and their attributes are modeled using class diagrams (as in Fig. 2).

We also model environment *processes*, using story patterns. They describe laws of nature (e.g. gravity), the behavior of simple machines (e.g. a conveyor belt), and non-deterministic external influences on the system (e.g. an entity arriving in the environment). They are useful both for simulating the system and reasoning about its expected behavior at the agent level.

The *agents* and their sensors and effectors can now be added to the environment model. Both sensors and effectors can only be applied to a specific *context*, i.e. the subset of all entities that is, e.g., of the right type and physically close enough to the agent. The story patterns that specify the effects of the sensors and effectors limit them to this context.

Sensors transform concrete entities into *perceptions*. When generating perceptions, the sensor usually only retains a subset of an entity's attributes, may transform and aggregate them, may introduce random errors with a specific probability distribution, or may even fail to produce a perception with a given failure probability (see Fig. 7(a)).

Effectors create, manipulate and destroy entities, their attributes and associations. Unlike typical AI-centric agent specifications that usually provide an agent with a set of named actions or performatives, the semantics of the effector actions we specify are fully transparent both for the agent and any formal method we would like to employ at the agent level; i.e. we can seamlessly integrate the environment into our analysis of an agent's behavior. The formalism allows specifying any conceivable state transition of the specified environment and is thus capable of expressing the effects of any effector, now matter how complex (see Fig. 7(a)).



(a) Sensing a nearby forklift using sonar (b) Picking up a crate from a shelf

Fig. 7. Sensor and Effector specifications.

Obviously, in order to obtain a valid model, the specified effects need to stay within the limits of what is reasonable and physically possible. Generally, the validity of any results obtained by means of simulation and formal verification of the model largely depends on the quality of the environment model, i.e., whether it is correct and appropriate. This is less of a problem for digital entities, as – due to the reliance on proven object oriented formalisms – they can be represented by their actual design. It is somewhat more problematic for physical entities, where we can only strive to provide as good an approximation as possible. Our approach is better suited to describing structural adaptation than continuous change. We currently only support modeling continuous processes through difference equations, which we consider sufficient for most application areas, though. If an in-depth treatment of the mechanical engineering aspects of the system is essential, it is necessary to additionally apply our techniques for the design of hybrid systems [14].

3.2 Service Model

As we have already suggested with the introduction of environment processes, entities are not limited to being inert, monolithic objects, even if entities in the environment model were limited to rather simple behavior. In principle, entities may be complex and have extensive internal machinery that performs complex actions. The essential distinction is that entities are never autonomous and do not possess internal motivation, i.e. they are passive unless activated by an agent or an environment process.

The *service model* basically describes the infrastructure used by the agents. This infrastructure is implemented as a set of services that are provided through dedicated entities called *facilities*. *Services* can fall into various classes, e.g. life cycle management, resource allocation, scheduling, communication, directory services, persistency, access control, authentication, or application-specific functions. They can reach a high level of sophistication, e.g. a distributed blackboard with consistency management.

Services mostly represent functionality that is normally provided by middleware. Indeed, services will often be implemented using some type of middleware. We can

differentiate between production middleware that will be present in the final system, usually providing lookup, messaging and other higher level functions, and prototyping middleware that is mainly concerned with emulating the production environment, providing services that will later be implicitly performed by the physical environment (e.g. computation of the available physical context) or the production hardware (e.g. scheduling).

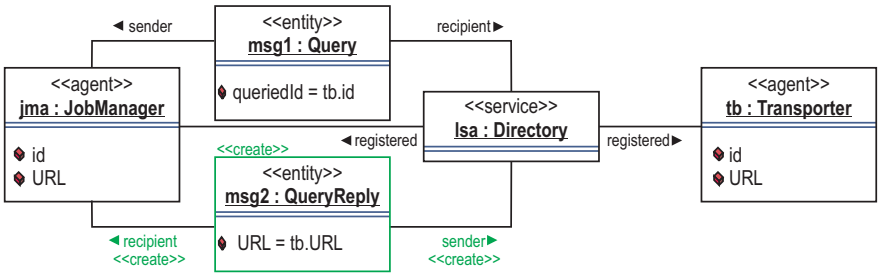


Fig. 8. A directory service replies to a query

As services are specified in terms of entities, we can apply the same object-oriented modeling techniques using story patterns as for entity behavior and effector use (see Fig. 8). However, as services can to some degree be standardized, they offer obvious potential for reuse. Specifying the same services from scratch over and over again would be tedious and inefficient. Templates encoding recurring desing patterns for reuse offer a solution to this problem. Templates may range from simple patterns describing the functionality of a single facility to complex systems of connected facilities representing a whole agent platform, component framework or distributed computing library. This means that after a service description for a particular solution has been modeled once, it can be reused, adapted and combined with other building blocks in a modular manner.

3.3 Social Model

Now, where are the agents, organizations, roles, and communication languages in all this? Frequently, agent-oriented methodologies that closely build on object-oriented software engineering techniques are criticized for focusing on the technical aspects of multi-agent systems and neglecting advanced agent-oriented abstractions, thus providing poor support for the coordination of multi-agent systems and essentially limiting their scope to simple reactive agents. We, however, believe that such abstractions can in fact be supported based on an object-oriented design.

Mentalistic concepts have proven useful for reasoning about autonomous, cognitive agents. It is mainstream in agent-oriented research to assume that agents have intentional stance, assigning beliefs, goals and intentions to them [15]. Despite its unquestionable appeal, formalisms based on intentional stance face some well-documented problems, notably when used in the context of agent communication (languages). Such formalisms often assume a specific implementation of the agents' internals, which

severely limits their applicability to real-world scenarios. As the semantics of messages depend on the state of an agent's mind, they may not be decidable from an outside perspective [16]. Besides, the resulting specifications are notoriously complex, and proving the conformance of an implementation may be impossible [17]. One solution that was proposed to solve these problem is to model agents as *observable sources* that expose a well-defined part of their internals in order to allow other agents to reason about their beliefs and intentions [18].

Legal Stance We propose using the environment to a similar effect, thus providing a generic mechanism that is completely independent of the agents' implementations. The basic principle is to not reason about what an agent actually intends or believes, but what an external observer, or more specifically other agents in the system, can know or reasonably assume the other agent to believe or intend. It is inspired by the way human interaction, or more specifically human laws, work. Courts frequently infer beliefs and intentions from situations, acts, and speech. Legal codes (in the continental tradition) devote significant effort to fixing the exact modalities of how and when a person can profess an intention. In criminal codes, intent is a defining characteristic of various crimes, and the punishment of attempted crimes hinges on establishing the intention (e.g., an unauthorized person breaking into and hotwiring a car could clearly be supposed to intend to steal it). In civil law, what a person should have known (e.g. caveat emptor) and seems to have intended based on the given evidence is a common question. We therefore call this view that is concerned with the *professed intentions* (and professed beliefs) that can be deduced from the environment *legal stance*.

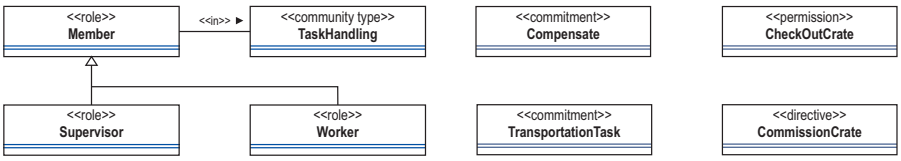
Conventions for interpreting the environment can be attached to any effector or entity type. This specifically includes messages, allowing the specification of agent communication languages, the predominant kind of social convention in current multi-agent system. The implied professed intentions can be used to reason about the system at a higher level of abstraction (see Fig. 12–14). Concepts such as assertions for communicating beliefs, directives as a means of soliciting specific behavior, or commitments for making behavioral guarantees (cf. [19]) help to structure and guide agent behavior.

Just like laws, professed intentions are artificial constructs that are only valid in a specific social context. A group of agents needs to agree on a set of conventions before it can become useful for governing their interaction.

Social Specification This social context is provided by *communities*, which are – possibly overlapping – groups of agents sharing the same conventions. Research into agent organizations has shown that social structure is essential for designing complex, heterogeneous systems [20]. While our ideas are conceptually close to established work on organizations, we chose the term 'community' in order to avoid confusion because we felt that 'organization' suggests a greater degree of institutionalization, persistence, and complexity than exhibited by many of the communities we have in mind, and, on the other hand, we did not want to try to change established concepts by making additions that are specific to our modeling approach to them.

The conventions used by a community are set down in the corresponding *community type*. The specification of a community type mostly consists of various types of norms, expressed in terms of observable entities by means of story patterns. In detail, a community type defines the following:

– a set of *roles* that can be assumed by agents (Fig. 9(a)),
– a set of *professed intentions* that can be attributed to agents (Fig. 9(b)),
– *instantiation norms* for instantiating communities, as there may be multiple instances (Fig. 10),
– *binding norms* for joining and leaving the community, and for assuming and relinquishing roles (Fig. 11),
– *conventional norms* that specify social conventions, i.e. generate professed intentions from observations (Fig. 12,13(a)),
– *behavioral norms* specifying allowed or required behavior, which need to be compatible to the agents' effector specifications (Fig. 13(b)), and
– a set of *community types* that can be used to form subcommunities contained in the community.



(a) Roles defined by the community type (b) The set of professed intentions

Fig. 9. Community Type for completing transportation tasks.

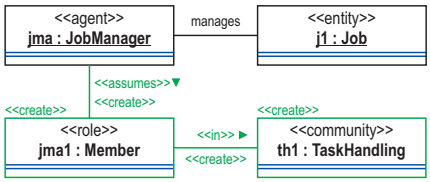
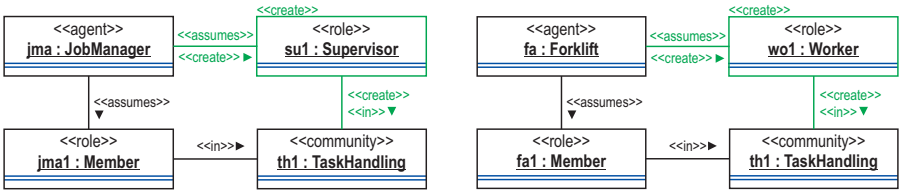
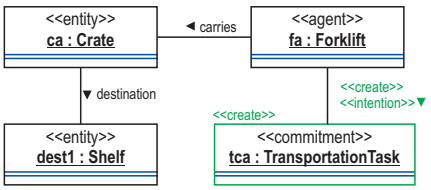


Fig. 10. Instantiation norm instantiating a community

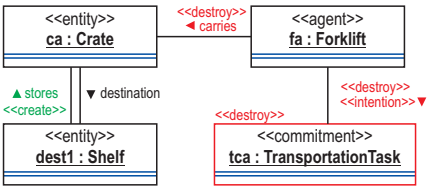


(a) Binding norm for JobManager (b) Binding norm for Forklift

Fig. 11. Binding member agents to more specific roles by type.

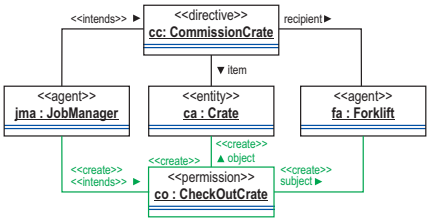


(a) Carrying a crate entails the commitment to transport it to its destination

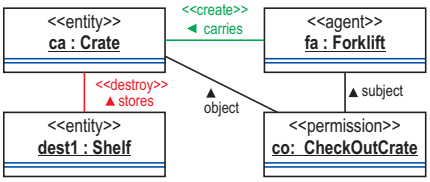


(b) The commitment is fulfilled by delivering the crate

Fig. 12. Conventional norms for making and fulfilling commitments.



(a) Conventional norm: the directive implies a permission to actually pick up a crate



(b) Behavioral norm requiring this permission as a precondition

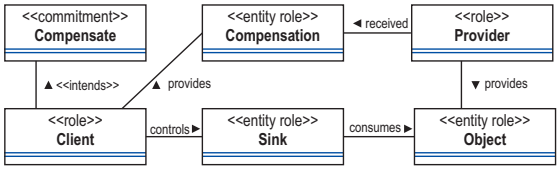
Fig. 13. Conventional and Behavioral norms interact to control behavior.

The specification is expressed entirely in terms of classes and objects, which are marked up with stereotypes in order to indicate their specific semantics. Therefore, it is possible describe community types as graph transformation system, which can be seamlessly integrated with the graph transformation system of the environment model to yield a comprehensive specification of the system's physical and social behavior.

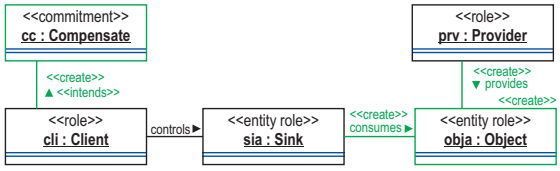
Community types can specify complex organizations, but may as well describe the ad-hoc interaction between a pair of agents. In general, a community type deals with a particular problem, which usually grows in complexity in proportion to the community type's position in the hierarchy.

As there may be commonly recurring subproblems (e.g., collision avoidance, job assignment, coordinating distributed problem solving), we once again propose the use of templates or design patterns. We call these patterns *cultures*. Cultures extract the essence of a community type by abstracting from the concrete environment. This is done by replacing the concrete agent and entity types used in norms (e.g. 'car salesman', 'motorist', 'car') with more generic agent ('buyer', 'seller') and entity ('merchandise') roles (see Fig. 14). A culture can be formally verified in order to prove that it correctly solves a problem, or, more specifically, satisfies a set of requirements using our work on the verification of coordination patterns [21]. It may then be reused in future systems to derive a community type inheriting these properties by binding appropriate concrete types or sets of types from the environment model to these abstract roles.

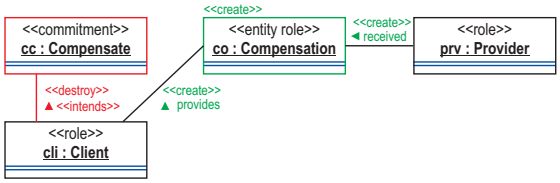
Even though community types impose requirements and limitations on the capabilities of agents, they do not restrict the actual implementation of agents in any way, mak-



(a) The culture is modelled in terms of roles



(b) Norm: Providing the service



(c) Norm: Compensating the provider

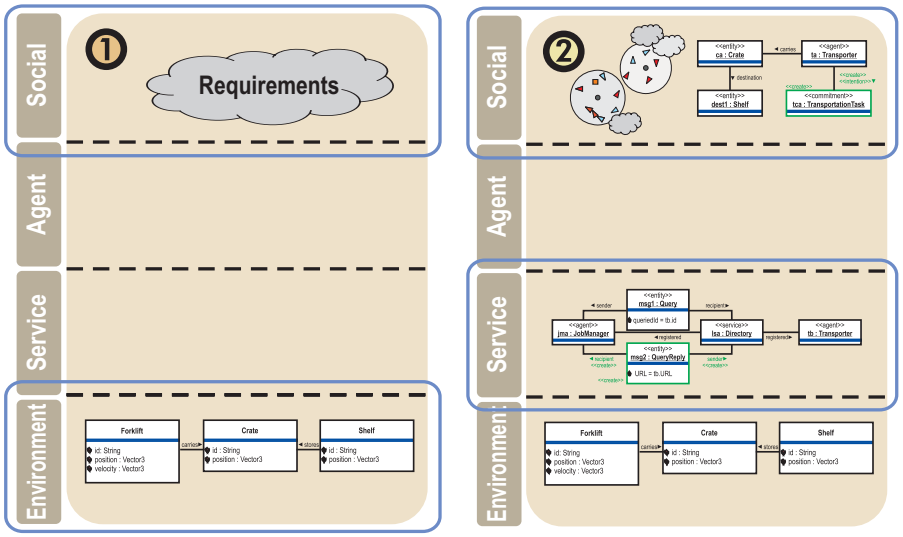
Fig. 14. Culture regulating payment for goods or a service in a generic manner. A concretization is used as a sub-community type inside the above TaskHandling community type by JobManagers (Client) for paying Forklifts (Provider) after delivering a crate.

ing the approach mostly agnostic with respect to their internal architecture. As the specification is only concerned with observable behavior, correctly implementing it comes down to behaving correctly in the environment. The agents have complete 'freedom of thought', even though certain mentalistic notions are certainly implied by particular behavior through conventions, which may require some sort of compatible internal model in order to be able to conform to more intricate behavioral requirements.

4 Process

In order to show how the proposed concepts can benefit the development of situated multi-agent systems, we now describe the process from requirements engineering to deployment as we envision it. As mentioned in the introduction, we divide this process into analysis, social design, agent design and deployment. Figure 15 gives an overview of the phases and their main objectives.

Even though each phase deals with clearly defined aspects of the system, a linear progression through the phases should be seen as an idealization. Each phase builds upon the output of the previous phase. Within the later phases, prototyping is used to enable iterative improvement of the specification. In practice, it will be necessary to revisit previous phases and make adjustments in this context.



(a) Analysis: Modeling Requirements and the Environment

(b) Social Design: Designing Communities and Services

(c) Agent Design: Designing agents' behavior and internals

(d) Deployment: Replacing the simulated environment

Fig. 15. Phases of the proposed process in more detail. The UML diagrams merely serve as iconic representatives for the respective models (see Fig. 12(b), 7(b), 8, and 2 for full-scale versions)

4.1 Analysis Phase

The analysis phase is mainly concerned with the specification of the environment model. The structure and behavior of the environment are considered as fixed at this stage. Using

methods for the identification of classes from traditional object-oriented analysis, the relevant entities from the system's prospective environment can therefore be identified and modeled. Likewise, the behavior of these entities may be observed and modeled through environment processes. Such services as are already provided by the environment are also recorded at this time. The result is a domain model of the environment which forms the core of the ontology used in later phases.

As agents (as physical entities), sensors, and effectors are part of the environment model, they are included in the analysis phase. This is only logical, as any model is, by a common definition, driven by a specific purpose, which in our case is to represent the environment as relevant to the agents. Without at least a basic knowledge of their capabilities, the environment model could not fulfill this purpose. Nonetheless, it could be argued that the agent types and the sensors and effectors available to them are design decisions that have no place in analysis. While it is true that the addition of new agent types may become necessary in the subsequent social design phase, and the exact capabilities of the sensors and effectors may not be fixed before the agent design phase, we do, however, consider it an important part of analysis to identify prospective classes of agents and establish a general idea of their (potential) capabilities with respect to the environment. In practice, especially when working with (mechanical) agents in phyisical environments, the agents and their capabilities are usually given to a large extent before the design of the multi-agent system even starts.

The second concern of the analysis phase are the system *requirements*. Again, established requirements analysis can be used to identify functional and non-functional requirements, as there is nothing inherently agent-specific in the requirements – after all, agent-orientation is supposed to be a solution, not part of the problem. The resulting requirements do not need to be expressed using a specific notation, they can even be informally documented in textual form. If a requirement is to be the subject of formal verification later on, it is however preferable to specify it directly as a story pattern over the environment model. It is furthermore desirable to structure and rank the requirements.

4.2 Social Design Phase

The social design phase begins by taking these requirements, breaking them down into suitable subsets, and assigning them to agent communities. These communities are then responsible for ensuring that the systems meets the requirements in question by defining appropriate norms.

For each set of requirements, a community type which is capable of dealing with this responsibility is then designed. This will usually include the definition of subcommunities concerned with even more specific tasks, which ultimately leads to a hierarchy of community types whose bottom elements are basic interaction patterns dealing with simple, manageable problems. At this time, cultures that address a specific requirement can be applied to the system. As cultures may themselves contain more specific subcultures, instantiating a culture may create a whole hierarchy of community types.

When applying cultures or devising new solutions to problems, the designer will need to take the agents' capabilities (as expressed by their sensors and effectors) into account. It is not helpful to specify behavioral norms that agents cannot enact, or conventional norms that depend on something agents cannot sense. If the physical design of

the agent is under the designer's control, it is possible at this point to add new capabilities through new sensors and effectors. The most common way to provide agents with additional capabilities, however, is to specify services that provide them. Also, the idea behind communities *is* using interaction in order to achieve goals beyond the reach of the individual agents. Thus, communities can themselves provide new capabilities that can be used as a bootstrap by other communities. For example, in order to allow every agent in a system to communicate with any other agent, one could introduce a network of access points capable of relaying messages and a community type that requires agents to register with a distributed directory, instead of upgrading the agents' antennas.

Together, the community types need to result in a consistent specification. If the requirements are not orthogonal, i.e., the norms of the community types constrain the same effector or concern the same entities, different community types might be in conflict. E.g., if a community type responsible At this point, we try to spot cases where conforming with all norms is *theoretically* impossible. Other than that, we merely strive to keep dependencies between community types as weak as possible and defer the task of actually reconciling conflicts to the agent design phase.

Once all relevant community types have been specified, the model can be validated. Individual community types can be formally verified: As both their norms and the behavior of the environment can be modeled as a graph transformation system, we can apply the above-mentioned invariant checking techniques [11] in order to prove certain required properties, e.g. the absence of accidents or safety hazards. We can also export the model into GROOVE, which allows us to systematically explore the state space for specific initial configurations up to a certain size. Other aspects such as requirements concerning efficiency and performance or emergent properties of complex systems that cannot be assessed through formal methods require empirical validation by means of a prototype.

As the complete specification, from entities and environment processes to roles, norms, and professed intentions, is expressed by class diagrams and story patterns, it is possible to create an executable prototype using code generation. The environment model simply needs to be run in order to simulate the environment, even though it is usually advisable to make use of some kind of prototyping middleware in the implementation of the model in order to achieve more efficient simulation. As for the social model, the generated implementation tries to apply all applicable norms to the system and keeps track of the result by instantiating explicit representations of communities and professed intentions. For each agent, the system non-deterministically choses an action from the set of enabled sensor and effector applications. This makes it possible to assess whether the system's behavior stays within the intended limits. When a constraint specified by a community type is violated or an agent gets stuck in a state with no valid course of action left, this is detected and reported by the system. It may indicate a conflict within or between communities. As corrections in the model can immediately be tested in a new prototype, iterative, step-wise improvements can be applied to the design quickly and conveniently. Starting from the earliest stages of the design, it is possible to generate prototypes from specifications in order to evaluate them.

4.3 Agent Design Phase

Once the social specification meets all pertinent formal requirements and appears to address all other requirements, the actual agents can be implemented, respectively specified. Even though any architecture producing the appropriate output is acceptable, it is convenient to start from the generated implementation of the social model. For behavioral norms that specify concrete, reactive behavior, adding strategies for intelligently chosing between the available options is a quick way to obtain a reasonable implementation. Behavioral norms that are of a more declarative nature, e.g., concerning the commitment to travel to a specific location by a given deadline, require more elaborate strategies and algorithms that cannot simply be deduced, but need to be designed.

In practice, devising an implementation that respects the constraints of all pertinent communities may not be a trivial task. It is facilitated by the ability to test prototypes of the agents in the simulated environment. The generated social model is reused for monitoring conformance with the specification. This time around, the model does not randomly choose a course of action for the agents, but merely checks whether the exhibited behavior was enabled, flagging violations. Agents can also be benchmarked extensively in order to optimize their performance.

4.4 Deployment Phase

Once the agent design conforms to the social specification and has proven itself in the simulated environment, it can be moved to its production environment for testing. An appealing feature of the proposed approach is that this mainly means replacing the simulated parts of the system with their physical counterparts. Provided that it was modeled correctly, the environment model can simply be dropped. At the service level, the prototyping middleware is replaced by the production middleware. The overall complexity of the software system decreases, as physics, processes and physical constraints (e.g. context) no longer need to be replicated in software; however, the middleware that processes sensor input and interprets effector commands becomes significantly more complex internally.

The model of the actual agents remains unchanged, as their interfaces are unchanged. Using a specific real-time runtime environment supported by Fujaba's code generator, we can actually reuse the exact same code for simulation and hardware tests [22]. Of course, it is nonetheless necessary to perform a sufficient number of tests in order to ensure that there were no errors or oversimplifications in the environment model that lead to significant discrepancies between simulated and actual behavior (cf.[23]).

5 Experiences

In order to evaluate our approach, we intend to apply it to a large, realistic scenario. Implementing a non-trivial example and testing the iterative development process furthermore requires appropriate tool support. Within the scope of the project group *intrapid*, consisting of 18 students and currently in its second and final semester, we are working towards these objectives.

We are basing our work on Fujaba for Eclipse, a port that integrates the Fujaba CASE tool into the Eclipse platform as plugins. We are implementing our extensions as a UML Profile, i.e., we add the appropriate stereotypes from our conceptual framework to the metamodel and define the different variants of story patterns and class diagrams we use with their respective constraints and semantics. We are also adding a configurable translation layer that transforms these diagrams into input for the existing code generation mechanisms by joining them together in the correct manner.

Another important part of the project is the work on 'prototyping' middleware that supports the efficient simulation, visualization and run-time manipulation of environments. At the agent level, a component-based 'production' middleware is providing the internal infrastructure.

The ultimate goal is to use the developed tools and middleware for the design of a large and complex logistics management application. The scenario comprises warehouses and various types of robotic agents moving goods inside and between them. It was chosen because logistics are a common application area of agent-based approaches that offers potential for optimization and exploiting synergies between agents. It also requires relatively simple agents, which, however, need to flexibly interface in different ways.

In order to give the students the opportunity to familiarize themselves with the approach and get a better grasp of the underlying concepts, we implemented a very simple scenario as a prototype. The idea was to keep the application specific part as basic as possible and focus on a sound implementation of the meta model and the infrastructure as a preparation for the work on tools and middleware in the second phase. The prototype was therefore not designed to validate the approach in general, but show the feasibility of some of our ideas and identify challenges.

We use the phases of the development process to structure the presentation of the prototype and our experiences.

5.1 Analysis Phase

The scenario consists of an aquarium containing a swarm of fish hunted by sharks. The environment is extremely simple: there are walls, cylindrical obstacles that may be placed dynamically by the user, sharks, and prey. Sharks and prey have sensors for sensing other fish (sharks have longer range, but a more limited field of vision, prey have a greater field of vision but limited range) and an effector (tail fin) for propulsion (sharks are faster, but prey is more agile). Sharks also have an effector for eating prey fish. The context of this latter effector is limited to fish in close proximity that are, literally, right under a shark's nose. There are no more advanced sensors and effectors, specifically none enabling direct communication.

The requirements were straight-forward: sharks were supposed to eat as many fish as possible, prey was to keep together and try to stay alive.

The main challenge at this level was designing and implementing the modular prototyping middleware needed to simulate the physical behavior of the system, compute the sensors' contexts, visualize the simulation and allow users to place obstacles into the aquarium. See Fig. 16 for a screenshot.

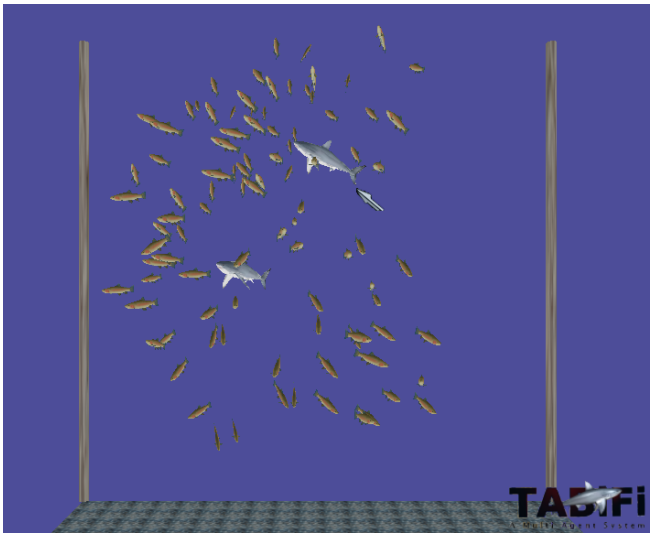


Fig. 16. Screenshot from the prototype

5.2 Social Design Phase

The requirements easily lead to the two community types shoal (for the prey) and pack (for the sharks).

The shoal type is quite basic. The behavioral norms are reactive in nature, describing how fish should react to seeing other fish or shark. They are modeled on the well-known boid paradigm [24], i.e., fish move towards the center of mass of the fish they see, try to keep their distance from neighboring fish and obstacles and match their velocity with other fish. Besides these basic behaviors, they exhibit a strong repulsion from sharks and places where sharks have recently been seen and a weak tendency to keep away from the outer parts of the aquarium. The behavioral norms require only simple processing of the fish and almost directly map sensor input into effector output.

Perhaps contrary to intuition, a large shoal of fish is not represented by a single community. It is rather composed of many smaller communities consisting of a fish playing the 'active' role and every fish it can see, playing the 'passive' role. As visibility between fish is not a symmetric relation, each of the communities from the same 'shoal' may have slightly different members. The behavioral norms of the community only have implications for the behavior of the 'active' fish at the center. While this may seem like a degenerate case of a community, it illustrates one important principle of community type design: A design that operates by governing behavior based on observable actions and states will only work as intended if it can reasonably be assumed that an agent expected to exhibit a certain behavior can actually make the observation that is supposed to trigger it. More specifically, there needs to be a conventional norm that can generate a professed intention to that effect (in this case the assertion that 'an agent knows what it sees', combined with the sensor context specification). In the example, as all prey fish

implement the same community type, behavior is still consistent – 'as if' all fish were in the same community.

The sharks' pack community type is based on similar behavioral norms. Sharks tend to keep greater distances, but match velocities in order to attack the prey in a coordinated manner. Obviously, sharks are strongly attracted by prey. There are simple conventional norms for creating professed intentions for coordinating shark behavior, e.g. determining which fish a shark is currently hunting so that sharks may cooperate in their attacks and do not go after the same fish.

Substantial effort was spent on implementing the shoal community type in the way it will later be automatically generated from the social model. This required creating explicit representations of the relevant meta model elements, namely community types, communities, behavioral norms (for specifying expected behavior), and binding norms (for determining membership in communities). As the behavioral norms are deterministic (unlike in typical future systems, where norms permit non-deterministic choice or may even only state goals that need to be achieved), a conforming implementation of agent behavior could be achieved by explicitly checking and applying behavioral norms.

5.3 Agent Design Phase

The sharks were implemented as simple intelligent agents. They are deliberately not implementing an explicit 'community' concept, but simply conform to the community type specification by implementing the required behavior. This demonstrates both how the approach can be used as a fairly architecture-agnostic specification technique and that the essence of the specification is in the observable behavior.

The prey fish were not implemented at all, as executing the community type specification was already sufficient for obtaining the desired behavior. All that was required was a stub for relaying the commands from the social model to the effector. The community type specification simply applies the behavioral norms to every fish in a fish's community, aggregates the results into a single impulse and passes the result down to the fin effector of the agent stub.

The ability to run a 'prototype' of one type of agent alongside an actual agent implementation only made a small difference in the current context, but should be interesting for the development of more complex systems.

The prototype allowed experimenting on the emergent properties of the system. By changing sensor and effector parameters and varying the thresholds and intensities for the behavioral norms, the balance of the system could be shifted and diverse behavioral patterns be induced.

For obvious reasons, there was now deployment phase.

In all, the prototype has already hinted at the potential of the approach, but has also helped to identify challenges. The efficient evaluation of social norms will be essential for prototyping and monitoring large systems. For truly assessing the added value of the approach on the conceptual level, we will have to wait for the full scale logistics scenario incorporating multiple community types and communication – the current state is already promising, though.

6 Related Work

Even though our methodology is clearly rooted in object-oriented software engineering traditions and techniques, the conceptual framework for grounding social interactions in the environment is based on agent-oriented abstractions and influenced by existing research from the domain.

The idea of the legal stance is both related to work on intentional stance (cf. [1]), the social level [25] and social order [26]. It was inspired by Viroli and Omicini's idea of agents as an *observable source* [18], but goes beyond it by basing the interaction on the environment. This provides a more flexible, general mechanism, at the cost of diminishing the ability to formally reason about the observations from an AI perspective. The categories of professed intentions we use stem from Singh's work on agent communication languages [16].

The concept of social structure was established by Ferber's organizational models [20]. As discussed, Communities combine an intentionally broad interpretation of what constitutes social structure with pattern-based software engineering techniques.

Environments still play a minor role in current multi-agent systems research, even if this is beginning to change. However, there are several papers putting forward similar ideas concerning the role of the environment:

Weyns et al. [3] discuss several functions of environments that are also important to our approach, namely structuring the system (for defining localized communities), providing a shared state (by storing the evidence of professed intentions), providing service support (through facilities), enabling coordination (this, again, is the key idea of legal stance) and acting as a regulating entity (by means of laws implemented as environment processes). The practical work on virtual environments [27, 28] could be seen as a real world application of the idea of bootstrapping by means of services. Our main contribution is offering systematic support of such solutions at the level of a model-driven development process.

Recent work by Omicini et al. [29] proposes 'artifacts' as a general way to structure the interaction between agents and the environment. In a way, artifacts and facilities represent similar concepts. However, facilities are rooted in the software engineering perspective and offer concrete, transparent behavioral specifications, whereas artifacts come from an AI background and provide more abstract interfaces based on messages, which facilitates standardization but makes analysis of the services they actually provide harder.

7 Conclusion and Future Work

We have presented a conceptual framework that grounds social interactions in the environment. We have moreover proposed an iterative design process that makes use of the ability to generate executable prototypes from high-level specifications and conduct formal analysis, starting with the early phases.

While the results obtained from our implementation work are encouraging, it is too early to actually draw any conclusions concerning the applicability and validity of our approach. We did, however, establish that our ideas for modeling the concrete parts

of the system centered around entities, sensors and effectors, are feasible and work as intended.

We hope to complete our work on the development tools in the foreseeable future and be able to test them on a large multi-agent system using sophisticated coordination mechanisms.

At the same time, we will continue our work on the theoretical underpinnings of our approach. We hope to be able to extend the scope of formal verification techniques in the social model and assist in the identification and reconciliation of conflicts between norms.

References

1. Wooldridge, M., Jennings, N.: Intelligent agents: Theory and practice. Knowledge Engineering Review **10** (1995) 115–152
2. Brooks, R.A.: Intelligence Without Reason. In Myopoulos, J., Reiter, R., eds.: Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91), Sydney, Australia, Morgan Kaufmann publishers Inc.: San Mateo, CA, USA (1991) 569–595
3. Weyns, D., Parunak, H.V.D., Michel, F., Holvoet, T., Ferber, J.: Environments for multiagent systems state-of-the-art and research challenges. In Weyns, D., Parunak, H.V.D., Michel, F., eds.: Environment for multi-agent systems: first international workshop, 2004, New York, NY. Volume 3374 of Lecture Notes in Computer Science. (2004) 1–47
4. Bonabeau, E.: Editor's introduction: Stigmergy. Artificial Life **5** (1999) 95–96
5. Fenster, M., Kraus, S., Rosenschein, J.S.: Coordination without communication: Experimental validation of focal point techniques. In: Proc. of the 1st Int. Conf. on Multiagent Systems (ICMAS), San Francisco, CA, USA, The MIT Press (1995) 102–108
6. Parunak, H.V.D., Brueckner, S., Sauter, J.A.: Digital pheromones for coordination of unmanned vehicles. In Weyns, D., Parunak, H.V.D., Michel, F., eds.: Environments for Multi-Agent Systems, First International Workshop, New York, NY, USA, 2004. Volume 3374 of Lecture Notes in Computer Science., Springer (2005) 246–263
7. Klein, F., Giese, H.: Separation of concerns for mechatronic multi-agent systems through dynamic communities. In Choren, R., Garcia, A., Lucena, C., Romanovsky, A., eds.: Software Engineering for Multi-Agent Systems III. Volume 3390 of Lecture Notes in Computer Science (LNCS). Springer Verlag (2005) 272–289
8. Klein, F., Giese, H.: Analysis and Design of Physical and Social Contexts in MultiAgent Systems using UML. In et al., R.C., ed.: Proc. of the 4th Workshop on Software Engineering for Large-Scale Multi-Agent Systems at ISCE, St. Louis, MO, USA, IEEE (2005) 1–7
9. Köhler, H., Nickel, U., Niere, J., Zündorf, A.: Integrating UML Diagrams for Production Control Systems. In: Proc. of the 22^{nd} International Conference on Software Engineering (ICSE), Limerick, Irland, ACM Press (2000) 241–251
10. Zündorf, A.: Rigorous Object Oriented Software Development. Habilitation, University of Paderborn (2001) Available online: http://wwwcs.upb.de/cs/ag-schaefer/Personen/Ehemalige/Zuendorf/AZRigSoftDraft_0_2.pdf.
11. Giese, H., Schilling, D.: Towards the Automatic Verification of Inductive Invariants for Invinite State UML Models. Technical Report tr-ri-04-252, University of Paderborn, Paderborn, Germany (2004)
12. Rensink, A.: Towards model checking graph grammars. In Leuschel, M., Gruner, S., Presti, S.L., eds.: Workshop on Automated Verification of Critical Systems (AVoCS). Technical Report DSSE–TR–2003–2, University of Southampton (2003) 150–160

13. Becker, B., Giese, H., Schilling, D.: A plugin for checking inductive invariants when modeling with class diagrams and story patterns. In: Proc. of the 3rd International Fujaba Days 2005, Paderborn, Germany. (2005)
14. Burmester, S., Giese, H., Oberschelp, O.: Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In: Informatics in Control, Automation and Robotics. Kluwer Academic Publishers (2005) to appear.
15. Rao, A., Georgeff, M.: BDI Agents: From Theory to Practice. In: Proceedings of the 1st International Conference On Multi Agent Systems, San Francisco, USA (1995)
16. Singh, M.P.: On Competitive On-Line Algorithms for the Dynamic Priority-Ordering Problem. IEEE Computer **31** (1998) 40–47
17. Wooldridge, M.: Verifiable semantics for agent communication languages. In: Proceedings of the 3rd International Conference on Multi Agent Systems (ICMAS98), Paris , France. (1998) 349–356
18. Viroli, M., Omicini, A.: A specification language for agents observable behavior. In: Proceedings of the International Conference on Artificial Intelligence (ICAI) 2002 (Las Vegas, US), CSREA Press (2002) 321–327
19. Singh, M.P.: The intentions of teams: Team structure, endodeixis, and exodeixis. In: ECAI. (1998) 303–307
20. Ferber, J., Gutknecht, O.: A meta-model for the analysis and design of organizations in multi-agent systems. In: Proceedings of the 3rd International Conference on Multi Agent Systems (ICMAS98), Paris , France. (1998) 128–135
21. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the Compositional Verification of Real-Time UML Designs. In: Proc. of the European Software Engineering Conference (ESEC/FSE), Helsinki, Finland, ACM Press (2003) 38–47
22. Burmester, S., Giese, H., Klein, F.: Design and Simulation of Self-Optimizing Mechatronic Systems with Fujaba and CAMeL. In Schürr, A., Zündorf, A., eds.: Proc. of the 2nd International Fujaba Days 2004, Darmstadt, Germany. Volume tr-ri-04-253 of Technical Report., University of Paderborn (2004) 19–22
23. Broeckman, B., Notenboom, E.: Testing Embedded Software. Addison Wesley (2003)
24. Reynolds, C.: Flocks, herds, and schools: A distributed behavioral model. Computer Graphics **21** (1987)
25. Jennings, N.R., Campos, J.R.: Towards a social level characterisation of socially responsible agents. IEE Proceedings on Software Engineering **144** (1997) 11–25
26. Castelfranchi, C.: Engineering social order. In: Engineering Societies in the Agent World, First International Workshop, ESAW 2000, Berlin, Germany, August 21, 2000, Revised Papers. Volume 1972 of Lecture Notes in Computer Science., Springer (2000) 1–18
27. Weyns, D., Schelfthout, K., Holvoet, T., Lefever, T.: Decentralized control of E'GV transportation systems. In Pechoucek, M., Steiner, D., Thompson, S., eds.: 4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands, ACM (2005) 67–74
28. Schelfthout, K., Holvoet, T.: Objectplaces: An environment for situated multi-agent systems. In: 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), 19-23 August 2004, New York, NY, USA, IEEE Computer Society (2004) 1500–1501
29. Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., Tummolini, L.: Coordination artifacts: Environment-based coordination for intelligent agents. In: 3rd International Joint Conference on Autonomous Agents and Multiagent Systems, 19-23 August 2004, New York, NY, USA. (2004) 286–293