Chapter 1
The Jadex Project: Simulation

Lars Braubach and Alexander Pokahr

Abstract

Simulation is on the one hand an important application area for multi-agent systems, but on the other hand also a useful tool for building agent applications. This chapter investigates constructs and techniques that foster both usages of simulation in the context of agent technology. The vision for integrating simulation support consists in establishing simulation transparency, i.e. it should be ensured that applications can be built to a large extent without simulation specific parts. First, approaches for dealing with time in simulated and non-simulated agent execution are discussed. Afterwards the role of virtual environments in agent applications is tackled. Both technical topics are illustrated using concrete applications that further represent the different usages of simulation.

1.1 Introduction

The combination of agents and simulation forms a mutual benefit. Multi-agent-based simulation (MABS) is an approach, that uses the concept of an agent for supporting social simulation. Agents are well suited for e.g. representing realistic human behavior in simulation models, such as pedestrian traffic in a to be constructed train station. Therefore, agents are an accepted technology in the area of simulation. Viewed from the opposite direction, simulation is also a useful technology for supporting the construction of agent applications. In many agent applications, the interaction between the agents is considered to be an important part of the computational algorithm, e.g. in negotiations or decentralized coordination. Building such agent applications

Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
{pokahr | braubach}@informatik.uni-hamburg.de

often requires fine tuning of parameters and making sure that the application produces suitable results, which can both be achieved by simulating the application behavior for testing and evaluation purposes.

This is one of two chapters describing practical applications built with the Jadex agent framework. This chapter describes techniques and constructs of Jadex, that particularly focus on supporting simulation while establishing *simulation transparency* to a large extent. Simulation transparency means that the functional code of an application must not contain any simulation specific aspects. In this way an easy transistion between simulations and applications can be achieved, e.g. code from an upstream simulation can be directly used to implement an application. It has to be noted that the environment and its connection to the application cannot be made transparent as in many cases a virtual one needs to be replaced by a real one.

Each section starts with a short historical background about why a certain topic was considered important for Jadex, followed by a more general motivation about the relevance of the concept itself. A related work section is presented for each concept, trying to give an overview of the field with pointers to other relevant works in the area. Afterwards the approach as implemented in Jadex is covered in detail and further illustrated by example applications that have been built. Each section closes with a short summary.

In this chapter, the following topics are tackled. The simulation of agent systems for supporting analysis and testing of applications is examined in Section 1.2. A useful supplement for simulation, but also a relevant topic in itself is the concept of virtual environments as described in Section 1.3. Finally, in Section 1.4 a conclusion and critical reflection on the described topics is given.

1.2 Simulation Clocks

One important aspect of simulation is how time passes during simulation runs. E.g. event-driven simulation allows executing scenarios "as fast as possible", because computation only happens for the relevant time points. Using timed execution, simulation helps comprehending system activities and interrelationships from a global perspective and in a timely condensed fashion. Analysis of system behavior can be done in different ways. On the one hand hypotheses about the system behavior can be tested using simulation experiments and on the other hand conclusions can be drawn from experimentation under different setups e.g. comparing alternative strategies. Finally, verification of system behavior is related to hypotheses testing but more concerned with ensuring that the simulation model complies in its behavior to some system design specification.

Yet, the connection between simulation and application construction is often not well established. In many cases simulation is considered on its own

as technique for experimentation. If a simulation is part of a real world application development project, in many cases simulation is used to analyze and verify the expected real world system behavior. Hence, the simulation model determines the design and implementation of the real world application, which is typically built from scratch, after simulation model validation has been taken place. This kind of throw away system construction is problematic not only for resource wastage reasons and the increased effort due to double development but also with respect to the preservation of validated system properties. These cannot be easily guaranteed for the newly built application if some of the model assumptions are implicit, e.g. hidden as implementation detail.

1.2.1 Related Work

Corresponding to the two viewpoints, agents for simulation vs. simulation for agents, solutions can be broadly categorized according to agent-based simulation toolkits and agent platforms with support for simulations.

The first group includes approaches systems like Repast [2], NetLogo[1] and SeSAm [6]. Most of these kinds of systems use a simple time-driven simulation clock advancement mechanism, which assumes the time passes in fixed steps. All agents are notified when a new round begins, typically in a sequential one by one manner, by invoking a behavior method. The processing of an agent is finished in a round when its behavior method is done. As communication between agents is handled in an indirect way by using the environment there are no negotiation based interrelationships between agents that need to be considered for end of processing determination. The whole round ends when all agents have finished their executions. In addition, most simulation frameworks assume a very simple agent architecture so that simulation scenarios with many simple agents are supported best. The time-driven clock mode has the advantage of being easy to understand and but disadvantage of being inefficient when activities of agents are not equally distributed over time.

Typical representatives in the area of agent platforms with simulation support are Cybele[2], Brahms[3] and PlaSMA [12]. PlaSMA is an extension for the JADE agent platform allowing it to be used as simulation runtime environment. In order to control the JADE agents PlaSMA uses a conservative time scheduling that resembles the time service protocol introduced in Section 1.2.2.1. The protocol is hidden from the user by using a simulation agent base class, which automatically notifies the clock when the agent's pro-

[1] `http://ccl.northwestern.edu/netlogo/`

[2] `http://www.i-a-i.com/cybelepro/`

[3] `http://www.agentisolutions.com/`

cessing is done. On the other hand, OpenCybele as well as Brahms employ infrastructure support based on clocks similar to the approach described in Section 1.2.2.2.

1.2.2 Approach

An important aspect of the solution consists in understanding that time advancement control is the key concept of simulation environments. The time advancement mechanism determines the temporal way the application is executed, e.g. in real time or in an event driven mode. The general idea is to use a clock abstraction for encapsulating the logic of time advancement separated from the rest of the infrastructure. The only simulation related activity in application code, which is valid for real-time applications as well, consists in issuing wait actions that interrupt processing until the specified time point has been reached. This allows executing the same application in different modes just by switching the underlying clock type. In general there are two different ways for realizing such a clock as a generic reusable component. It can either be intimate part of underlying runtime infrastructure or it can be added at the application layer. The first option has the advantage that the programming model for applications needs not to be touched at all as the clock interaction can be integrated in the API (application programming interface). On the other hand, it is challenging because it has to be anchored at the heart of the runtime platform. In contrast, the second option is non-invasive with respect to the infrastructure but requires the programmer to explicitly handle a time protocol for clock interaction in addition to the existing platform API. In the following a closer look will be taken at both solution paths, starting with the time service as incarnation of an application layer solution.

1.2.2.1 Time Service

The time service concept [1] assumes that a global coordinator is responsible for time management. Time clients have to contact the coordinator in case they have finished their activities or want to wait for a specific point in time. Adapting the notion of synchronous process-oriented simulation, the participants send to the coordinator a *Passivate* message to indicate that they have no scheduled activities, or a *Hold(t)* message with the time of the next activity that needs to be scheduled. The coordinator acts in a round based fashion by waiting for messages of each notified participant. It uses the messages to update its global list of announced time points. A round is finished when all participants have sent their decision. The coordinator then iteratively removes the next entry from this list, advances the clock, and

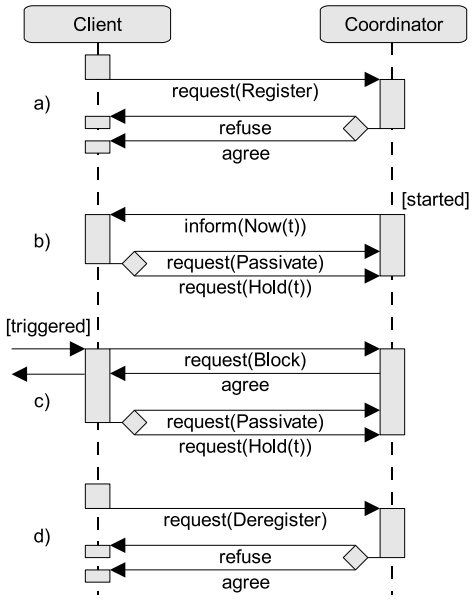

Fig. 1.1 Time Service Protocol (from [1])

informs the corresponding participants that the time point is reached. Then the coordinator will wait until all processes have answered again.

Fig. 1.1 shows an UML interaction protocol of the time service protocol. The four different interaction cases are denoted by the characters *a* to *d*. The initialization phase (*a*) is used by participants to register at the coordinator using a *Register* message and will receive an *agree* message, if they are not already registered. When a participant terminates it requests a *Deregister* (*d*), receiving a *refuse* if the participant was not registered. The other two interactions (*b* and *c*) may happen repeatedly during simulation runs.

The initial time (*Now*) is sent to all participants after the simulation has been started (*b*). New participants that register while the simulation is running will immediately receive the *Now* message with the current simulation time. While the simulation is running a participant will continuously receive so called wake-up calls whenever its registered point in time is reached. After receiving the wakeup the participant is supposed to execute its current activity and sends back a message afterwards. Either it announces the time point (t) for its next activity by submitting a *Hold(t)* request, or it currently has no activities to be scheduled and therefore submits a *Passivate* request. If the participant does not answer in a pre-defined timeout period the simulation continues with the next event excluding the non-answering candidate.

The participant that has been woken up may interact freely with all available other participants. Thus, waiting participants can react to messages received from other participants. In addition, a waiting participant may re-

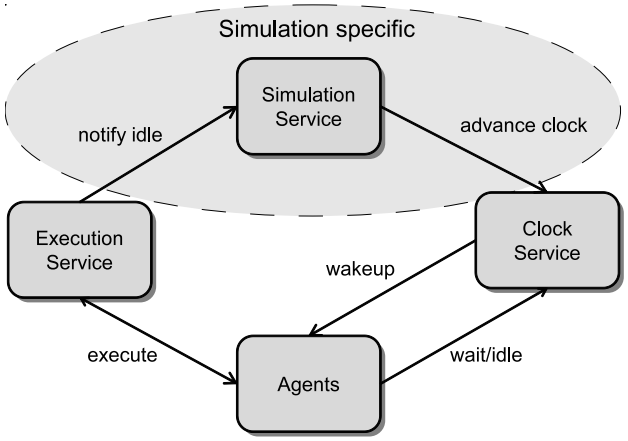

Fig. 1.2 Infrastructure components

ceive new information leading him to reconsider its activation time point (*c*). It can decide to activate itself at the current point in time by sending a *Block* request to the time service. The service removes the original time entry and acknowledges this by sending an *agree* message. It has to wait until all active participants declare that they are finished by sending a *Passivate* or *Hold(t)* request.

1.2.2.2 Infrastructure Clock Support

Infrastructure clock support allows completely hiding the timing mechanism that is needed to control application execution. For this purpose the clock has to be coupled with the execution machinery of the infrastructure. The general setup of such an infrastructure is depicted in Fig. 1.2. It consists of three interacting services, namely the *execution*, *simulation* and *clock service*. These services control the execution of the application by activating agents at specific time points. Concretely, the clock service manages an ordered list of time points which have been announced by the agents calling *wait* or *idle* (if they do not wait). Whenever the simulation clock is instructed by the simulation service to *advance* the time, it removes the next time point from the list and calls *wakeup* on the corresponding agents. The agents are subsequently *executed* by the execution service as long as they wish to perform tasks at the current point in time. The execution service monitors overall agent execution activities until no agent wants to be executed any more. If this is the case, it *notifies* the simulation service about the reached quiescence. The simulation service acts as connecting link between execution and clock service. It allows controlling exactly when the clock is advanced and normally instructs the clock to advance the time whenever it receives

notifications of the execution services. In some situations it can also defer the clock notification e.g. if the system is running in stepped mode and requires a human user to trigger the next clock step.

In addition to simulations, Figure 1.2 also shows how normal applications are executed. In this case the simulation service does not exist and there is no connection between the execution and clock service. This works because normal clocks are active by themselves, i.e. in contrast to simulations time advances automatically and the clock does not need an external trigger. From the agent's perspective the execution works in the same way as before by announcing wait and idle commands so that they can be built agnostic with respect to the execution mode.

Infrastructure clock support currently is limited to simulations running on the same platform. In order to support also distributed simulations on infrastructure level, the clocks of the participating platforms would have to adhere to a protocol that ensures that a virtual global simulation time is used. Following a conservative approach, one could employ a master slave approach, in which the participating clocks first use an election algorithm to decide about the master role, and afterwards use the master to annouce timing events in a similar way as in the time service protocol from the last section. Such a solution is transparent for the agents on the platform as they need not to be aware of how the clock service derives its current time.

1.2.3 Applications

In the following two example applications will be presented, for which simulation is a necessary prerequisite. The first, called MedPAge, deals with appointment scheduling in hospitals and the second, named SodekoVS, tackles software engineering with self organization.

1.2.3.1 MedPAge

Within the "MedPAge" hospital logistics project (cf. the other Jadex chapter in this book), an important requirement was the ability to benchmark different kinds of hospital appointment scheduling algorithms against each other. This was needed to better understand the quantitative advantages and problems of the new decentralized, agent-based approaches with respect to the established mechanism within hospitals. In order to efficiently execute the MedPAge application many time with varying parameters and underlying scheduling mechanisms simulation techniques are required. Using event-driven simulation instead of real-time execute allows conducting the experiments as fast as possible, i.e. computing resources are the only determining factor for experiment execution time.

The second crucial requirement within MedPAge was the ability to use the system to perform benchmarks and to run it as application prototype within a real hospital environment. Typically, this would require a huge effort as simulation and execution platforms rely on a different set of concepts and it is not easily possibly to adapt code written for one type of platform to the other. Furthermore, in most cases agent simulation toolkits focus on large number of simple agents and do not support negotiation protocols and intentional agent concepts. Hence, from those MedPAge project requirements a runtime infrastructure should naturally include general simulation support and allow programming of simulations and applications with a consistent programming model for both.

The infrastructure clock support described above has been integrated into the Jadex platform in order to simplify the development of simulation applications and especially enable the creation of simulations whose code can be kept for subsequent application development. This capability proved useful in the context of the MedPAge project. In an earlier project phase the simulation capabilities of Jadex could be exploited to isolate the most promising appointment scheduling algorithm among several approaches in the hospital domain. In a later project phase the application was extended in the direction of an assistance system that concrete helps with deciding which patient should be called next to a functional unit for treatment. For this purpose the appointment scheduling mechanism was kept as is and an additional user interface was added to the system. Using the system clock instead of a simulation clock the application could directly be tested within its target environment [14].

The time service approach from Section 1.2.2.1 was an integral part for simulation control in the so called Agent.Enterprise [3] and Agent.Hospital [5] initiatives. They were part of the German priority research programme SPP 1083 and served as integration approaches for the numerous subprojects within the SPP, including MedPAge. The general idea was to create a complex enterprise or hospital application scenario, in which the projects work cooperatively together to fulfill higher level objectives. Concretely, the Agent.Enterprise scenario is an inter-enterprise multi-level supply-chain scenario, including process planning and SCM scheduling as well as tracking and tracing of supply chains. Agent.Hospital builds on a model with numerous different healthcare actors and consists of detailed partial models of the healthcare domain. It enables the examination of modeling methods, configuration problems as well as agent-based negotiation strategies and coordination algorithms. The Agent.Enterprise and Agent.Hospital applications have been realized as so called multi-multi-agent systems, i.e. a multi-agent system that is composed of further multi-agent subsystems that bring about specific functionalities. The role of the time service was to timely coordinate the simulation within the overall multi-multi-agent system by managing the execution order of the different subsystems.

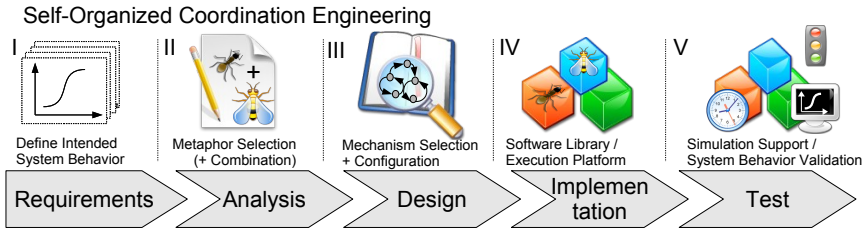

Fig. 1.3 SodekoVS development process (from [10])

1.2.3.2 SodekoVS

"SodekoVS" [10] is a DFG-funded[4] project that aims at making self-organization techniques usable as part of the normal software engineering process. In many application areas non-functional requirements like fault tolerance and adaptability play an important role. Examples include urban transport systems consisting of many small vehicles, low cost satellites that are able to perform a mission together as well as monitoring and automatic reconfiguration of server farms in case of changing customer demands. From these examples it becomes apparent that it is a key requirement that single entities may fail at any point in time, e.g. a hardware error occurs in a server, and these error must not disturb the overall system functionality. Furthermore, the examples highlight that a completely decentralized infrastructure is assumed in which a multitude of autonomous entities act and interact to bring about the system objectives. No superordinated entity exists, which on the one hand avoid a single point of failure but on the other hand demands novel software concepts to realized coordination as a function of peers.

Methodology

The SodekoVS project aims at providing a development process as well as a middleware for constructing applications with self organization features. The SodekoVS middleware is based on the Jadex agent framework and especially depends on the integrated simulation support. In Fig. 1.3 the proposed development process is shown. It can be seen that it shares the typical development phases with traditional processes and adds additional self organization tasks in each phase. At the heart of the approach distributed control loops are used to describe the expected macroscopic behavior in terms of role interactions. The control loops describe coordination behavior in terms of system state variables and causal relationships between them denoting the rates of change. Starting from the requirements phase the intended system behavior

[4] Deutsche Forschungsgemeinschaft (German Research Council): `http://www.dfg.de`

is elicitated. In the following analysis phase it has to be decided which coordination metaphor fits best the application needs, examples include pheromone approaches inspired by ant colonies and waggle dances of bee societies (cf. [11]). Afterwards a catalogue of ready-to-use self organization mechanisms, in the spirit of software engineering patterns, can be inspected in the design phase to find a suitable coordination mechanism. The patterns represent implemented coordination strategies for common use cases and can be directly integrated into an application. A developer has to configure the mechanism according to the system variables to be used and their update rates. In the implementation phase the binding between these variables and the agent states has to be defined. In this respect it has to be concretized in which situations agents play specific roles according to the macroscopic model and how transitions between such roles occur, i.e. it has to be defined how single agent behavior causes coordination actions. After the system has been implemented its behavior especially with regard to the coordination behavior has to be tested and validated. For this purpose the system is run in simulation mode in various different scenarios. Often self-organization mechanisms require parameter adjustment to function as expected. Using simulation these tasks become manageable and executable in a comparably short amount of time (compared to real-time application configuration). In case validation is completed successfully, the system can be deployed in the target environment and operates in real time.

Middleware

The architecture blueprint of the SodekoVS middleware is depicted in Fig. 1.4. The figure shows a layer model with three layers. At the bottom the execution infrastructure layer is located. In this layer the agent platforms are placed, which are responsible for providing fundamental services to the upper layers, e.g. agent execution and management on possible different network nodes. The topmost application layer realizes the application functionality by a set of agents. Between these layers, SodekoVS adds a coordination layer, which has the task to rather transparently realize the modeled self organization. For this purpose the concepts of coordination endpoints and coordination media are introduced. Each coordination endpoint registers itself with a coordination medium and is this way a network of endpoints is created. A coordination endpoint is part of an agent that independent of its original behavior and is used for two purposes. On the one hand, the endpoint observes the agent's state and forwards relevant changes to its associated coordination medium. On the other hand, the endpoint receives information updates from the coordination medium and influences the agent behavior if those updates are relevant to the entity. The coordination medium itself realizes the dynamics of information processing and distribution by relying on specific decentralized coordination mechanisms. The explicit distinction be-

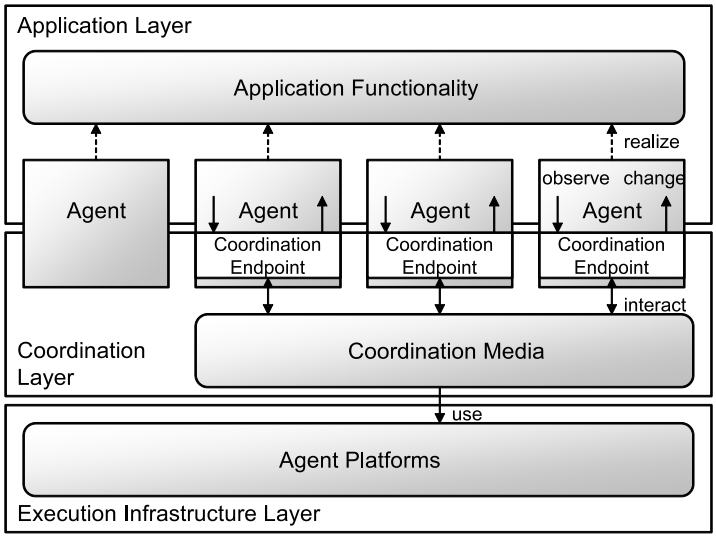

Fig. 1.4 SodekoVS middleware (from [10])

tween endpoints and media reflects the conceptual separation of local entity adaptations and coordination based information exchanges.

The SodekoVS middleware has itself been used for the development of several self-organized simulations and applications. Most notably, the approach was employed in the logistics domain to optimize parcel routing via heavy goods vehicles (HGV) between redistribution centers. In this scenario a market-based negotiation strategy was applied that allowed parcels to bid for transportation by an HGV with a virtual currency. The HGVs transport parcels between redistribution centers and try to optimize their own profit by serving different routes and negotiating prices with goods. More details about the approach can be found in [7].

1.2.4 Summary

In this section the usefulness of simulation itself and simulation techniques as part of application development have been discussed. It has been highlighted that time advancement is of crucial importance for simulation infrastructures and that it is possible to factor out time management and provide solutions that operate independently of the programming model. This leads to a consistent agent programming for simulations and applications using the same concepts. From the programming perspective one is unaware of the time mode the application is run with and can use a clock type that fits to the scenario needs.

Two conceptually different approaches have been introduced for externally and internally controlling time advancement. The first introduces an infrastructure service called *time service*, which represents a global clock that is used by the agents to coordinate their execution according to the simulation time. Concretely, the service manages a list of time points announced by the agents. In case a time point is due the corresponding agent is awakened and starts processing, which may involve communication with arbitrary other agents that may also start processing. After the activated agent has finishing processing it needs to notify the time service so that the clock can advance and the next agent is activated. All agents may decide to change their registered time point at any time if new information becomes available.

The second approach is based is tightly integrated with the runtime infrastructure and uses the interplay of three services for bringing about simulation in a completely transparent way. The clock service manages again the list of registered time points. In this case the execution service, which monitors the agent activities, triggers the advance of the clock (indirectly via the simulation service) whenever the agents have finished their processing. In contrast to the time service solution also the interaction with the clock is completely hidden. This is achieved by making the agent to clock interaction part of the normal platform level application programming interface.

It has further been shown that both approaches can be used within different context beneficially. The time service allows creating simulations within heterogeneous and possibly distributed environments, in which no direct control about the execution infrastructure can be exerted. On the other hand, simulation clocks allow constructing simulations and applications only within one platform but with much less effort because the agent programmer does not have to care about simulations and can build its application as if it were a normal application.

1.3 Virtual Environments

Virtual environments have a number of typical and less common use cases. Obviously, virtual worlds form an integral part of many computer games, and similar technologies can as well be found in training applications. Also for the teaching of agent concepts, virtual worlds are often employed, as the idealized settings simplify the understanding of the complex concepts. But even for the development of more conventional (e.g. business) applications, virtual environments can be a helpful tool. During implementation it can be helpful to execute parts of the later system in a controlled environment. In this case, the virtual environment would represent external systems or subsystems. Explicitly modeling this environment allows observing the behavior of implemented components in certain situations. This approach can be regarded as similar to mocking techniques as found in software testing, where

e.g. special mock objects are built for replacing parts of a real software environment during testing. Especially for agent systems, that exhibit pro-active, autonomous and adaptive behavior, setups based on virtual environments are helpful for testing and debugging the complete application during the implementation phase. A virtual representation of the external environment is also paramount when using simulation as described in Section 1.2. During application development, simulation can fulfill a number of different purposes. On the one hand, it allows intensive testing of an application prior to putting in into productive use. On the other hand, one can benchmark different implementations in the same environment or test implemented system behavior in changing environments. Finally, one may consider the virtual environment as part of a deployed application in the sense of augmented reality. For example, a virtual environment could be used for representing digital pheromones as part of an ant-like path-finding algorithm for a transport logistics application.

1.3.1 Related Work

In line with the use cases for virtual environments mentioned in the previous section, at least two different strands of research related to agents and virtual environments can be identified. The first concerns specialized simulation toolkits that often include simple agent frameworks for easy definition of the behavior of simulated entities. Typical examples are NetLogo[5] and Repast Simphony[6]. These toolkits are well-suited for agent-based simulation, e.g. for teaching or analysis purposes. In this respect, simulation toolkits usually offer rich facilities for statistical evaluation. Also visualization, e.g. as 2D virtual worlds are a typical strength of these systems. Some simulation toolkits, such as SeSAm [6], even offer graphical tools for specifying simulation behavior making them usable even by non-programmers. On the other hand, simulation models developed in these systems are not meant to be part of deployed applications. Thus unlike middleware platforms like JADE, simulation toolkits do not provide a communication infrastructure or interfaces to external systems.

Another strand of research investigates the explicit modeling of environments for agent-based applications. E.g. Weyns et al. argue in [13] that an explicit representation of an environment in agent applications can be useful as a part of the application itself, e.g. for a coordination layer. The idea is that a clean separation between agent and environment implementation simplifies the development and leads to better maintainable code. On specific model for such an explicit environment is the A&A model, which considers an application to be composed of agents and artifacts [9].

[5] `http://ccl.northwestern.edu/netlogo/`

[6] `http://repast.sourceforge.net/`

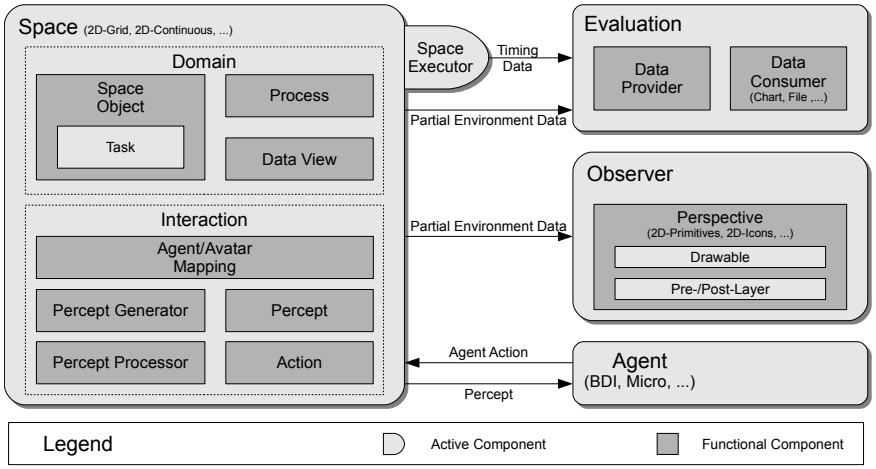

Fig. 1.5 EnvSupport structure (from [4])

1.3.2 Approach

The approach chosen for supporting the development of virtual environments in Jadex was separating the different aspects and allowing a declarative specification of each of them [4]. The resulting model is depicted in Figure 1.5. The *space* describes the environment itself and is further subdivided into the *domain*, i.e. parts of the environment that are independent from agents, and the *interaction*, which establishes a connection between the domain and the agents that are meant to inhabit the environment. The domain representation of the environment state is used by two further aspects of EnvSupport. The *observer* provides a visual representation of the environment, e.g. as a 2D map, and the *evaluation* component extracts environment data for statistical analysis. All these aspects are described as part of an application XML file and are interpreted by the Jadex platform at runtime. In the following sections, each aspect will be explained in more detail.

1.3.2.1 Domain

The underlying assumption regarding the domain is that the state of an environment can be described as a set of objects, so called *space objects*. For each application, the developer can freely define the available object types in the environment, where each type defines a set of properties for describing the object, such as position, size, etc. For static environments it is sufficient to describe the types and instances of all objects. Dynamic environments may change without any actions being performed by agents. Therefore the developer can specify such changes in the environment in two ways. *Tasks* are

attached to an object and may continuously change the state of this object (e.g. movement of a car, growth of a plant) until the task is stopped or the object is destroyed. *Processes* are applied to the environment as a whole and may induce changes on all objects as well as destroy some existing objects or create new ones. A typical use case is the creation of new objects according to some predefined stochastic distribution (e.g. arrival of cars at a junction, when the environment should only represent the junction itself).

1.3.2.2 Interaction

The interaction describes the information flow between the agents and the environment as represented by the state objects. For making each agent situated in the environment, an agent usually has an *avatar*, i.e. a space object that is owned by the agent. In this constellation, the agent represents the brain and the avatar space object represents the body of the situated entity. The interaction is divided into *percepts*, which are environmental states or changes observed by the agent's avatar and forwarded to the agent, and *actions*, that allow an agent to manipulate the environment state. *Percept generators* can be defined that describe how and when percepts are produced, e.g. by assigning a vision range to an avatar object and generating a percept whenever objects enter or leave the vision range according to the avatars current position in the environment. To simplify dealing with percepts, *percept processors* further describe how a percept enters the internal reasoning process of the agent. E.g. a ready-to-use BDI percept processor allows mapping percepts directly to some belief or belief set of an agent and therefore achieves a seamless integration of EnvSupport with the BDI architecture. For simple micro agents typically custom percept processors are defined by the application developer for triggering appropriate reactive behavior of the agent. Actions are requested by the agent and performed by the *space executor*. The space executor takes care of proper synchronization of agent actions, object tasks and environment processes. Depending on the scenario, the developer may choose a round-based or a continuous time space execution. The first model allows each agent to perform only one action per time step and is especially useful in conjunction with simulation clocks. The second model resembles are natural evolvement of time and only settles conflicts, e.g. if two agents try to pick up the same item at the same time, the executor will make sure that only one of these actions succeeds and the other one produces a failure.

1.3.2.3 Observer

The purpose of visualization is usually gaining a better understanding of the behavior of the application, either to use the application (e.g. a game or training simulation) or to analyze and debug the application. It largely

depends on the structure and properties of the space objects how an environment can be visualized. Typically, space objects are assigned a position in a two-dimensional area. Therefore, common visualizations for 2D maps are readily available in EnvSupport (e.g. continuous areas or discrete grids). In a so called *perspective*, the developer can assign a visual representation called *drawable* to each type of space object. A drawable may consist of an arbitrary number of drawing primitives (geometric shapes, external images, text), which can be further configured using properties of the space object (e.g. using different images according to the age of a plant). *Pre-* and *post-layers* can be added to a perspective to show the image of a map behind other drawables or to paint a grid on top of the visualization. Multiple perspectives can be defined for each application and each perspective can be used to create a visual representation of all objects or a selected subset according to *data views* defined in the domain. Current developments are directed towards extending the observer for incorporating also 3D visualizations based on the JMonkey engine.[7]

1.3.2.4 Evaluation

The observer allows producing an intuitive and highly accessible way of understanding and analyzing application behavior. For the numerical analysis of simulations an evaluation component is provided. It allows keeping track of any space related information during application execution. Just like the observer, the evaluation component takes as input all space objects or a subset as defined in a data view and continuously extracts property values according to the specification of *data providers*. A data provider is a query producing a database table structure, i.e. for each time point of the application execution, the data provider takes the state of the environment and produces a row of data values extracting the relevant information from the space objects. The data is then used in *data consumers* that allow, e.g. writing it to a file for later off-line analysis or plotting it into a chart for real-time observation.

1.3.3 Agent-based Simulation: City Bikes

Nowadays, bicycle sharing systems are deployed in many cities, allowing quick and easy 24/7 access to bikes for tourists, commuters, or any other person interested in using a bike for a short period of time. In these systems, the bikes can be checked out and returned at various stations in a more or less dense network of stations. E.g. on her journey to work a commuter can check out a bike near her home location and return it near her work place. An open

[7] `http://jmonkeyengine.org/`

problem in these systems is the distribution of bikes to the different stations. If too many bikes are at a station, no more bikes can be returned there, but if too few bikes are present, the station might run out of bikes. This problem is typically addressed using dispatchers, which transport bicycles between stations by van to establish a balanced distribution of bikes.

In the context of the *StadtRAD Hamburg*[8] system in Germany, an agent-based simulation model was built [8]. The model served the purpose to test and evaluate different scenarios to determine factors that influence the effectiveness and efficiency of the bicycle sharing system. Two concrete aspects were further investigated in the performed simulation studies. First, it was analyzed how the addition of new stations at certain places would affect the overall bicycle use. Second, several different strategies for dispatching were evaluated. Therefore, the goal of the model was to obtain realistic behavior for the bicycle usage that wasn't based on historical data, but would rather respond to the changes that were made to the environment for the different simulation studies.

1.3.3.1 Environment Model

As a virtual environment, the network of StadtRAD stations in Hamburg was modelled using EnvSupport. Besides the StadtRAD stations, also the public transportation network was modelled, because it was considered that for long distances a combination of subway/urban train and bicycle would be preferred. The simulation model makes use of EnvSupport by defining the domain and interaction aspects in an XML description as follows.

The StadtRAD bicycle stations as well as train stations are represented as domain objects with a fixed location on the map. For the bicycle stations, the number of currently available bikes and the number of total slots[9] have been modelled as properties of the station object. Additionally, traffic participants are domain objects with a dynamic location, i.e. their location changes according to their travels. Each participant is assigned a random mobility value that influences how fast she can travel on foot or with a bike. Furthermore, train schedules have been modelled as domain processes, i.e. the processes encode the logic of moving traffic participants that board/unboard trains at certain locations. When executed, the simulated environment can be visualized as shown in Figure 1.6. For the visualization, drawable representations such as icon images are assigned to each of the modelled domain objects, such as train and bicycle stations as well as traffic participants.

The behavior of the traffic participants should be controlled by agents. Therefore an avatar mapping is defined that specifies the agent type corre-

[8] *http://stadtrad.hamburg.de*

[9] At the time the model was built, StadtRAD did only support returning bikes at a station, when there was a free slot. This was changed recently, such that now bikes can also be returned when there are no free slots.

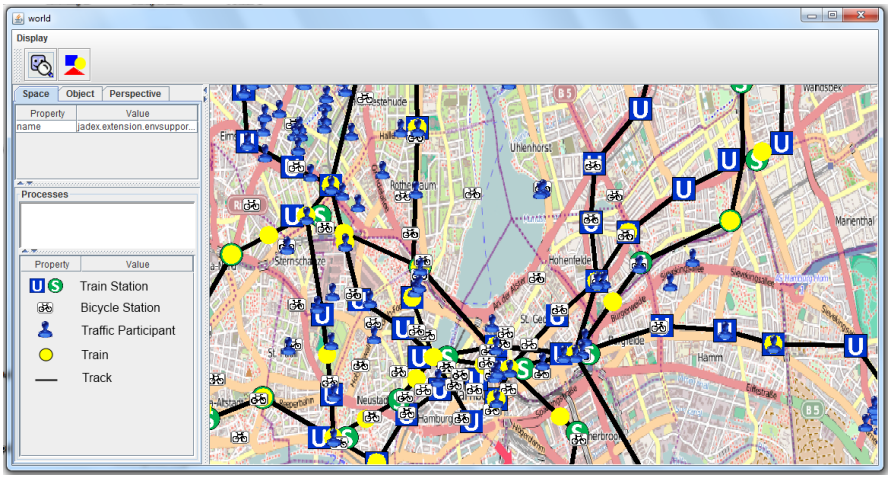

Fig. 1.6 Screenshot of the simulated StadtRAD environment

sponding to the traffic participant object. As a result, for each traffic participant in the simulation, a corresponding agent instance is automatically created. The agent may use declared actions to interact with the environment. Actions are modelled as Java classes and referred to from the XML environment description. The following actions have been defined in the StadtRAD model. First, the traffic participant may check out or return a bike, if her location matches the position of the station. Similarly, the participant can board/unboard trains at train stations. Finally, unless boarded on a train, a traffic participant can travel by herself to any chosen location, whereby the traveling speed depends on the participant's mobility value and if it currently has checked out a bike.

1.3.3.2 Agent Behavior

The aim of the simulation model is to achieve realistic bicycle usage behavior for being able to analyze the effect of changes to the StadtRAD system. Therefore, the traffic participants are represented as goal-directed agents that autonomously decide about if and how they would use a bike. The agents are created with a set of recurring goals to visit certain locations for leisure or commuting purposes. Following the BDI model, the agents perform a reactive goal/plan decomposition of their traveling activities (cf. Figure 1.7). The BDI reasoning starts from the top-level goal: *visit location*. For each target location, the agent decides to choose a previously used route (*plan: use known route*) or to try out a new route (*plan try new route*). For this purpose, the agent has knowledge about its recently travelled routes (*belief: previous routes*). Each route is a sequence of segments, i.e. intermediate locations and

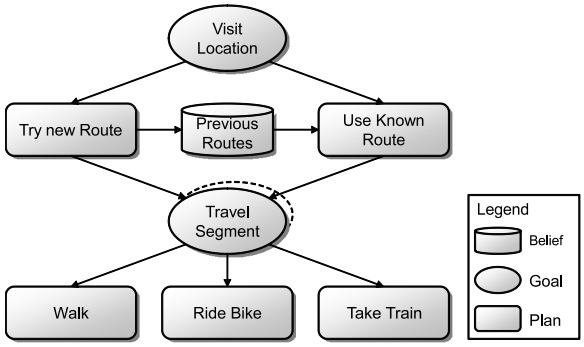

Fig. 1.7 Goal/plan tree of the traffic participant agent

corresponding transportation means. As an example an agent might decide to switch to using a bike, instead of following a previous route that included a lot of changes between trains and waiting times. Afterwards the agent would remember the time taken for this new route and depending on its personal preferences, would possibly choose the new route again for future travels.

1.3.3.3 Simulation Studies

The behavior of the agents depends on their personal preferences, i.e. maximum distances that they would prefer walking, using a bike or taking the train. Before running the simulation studies, the simulation model was calibrated to more closely match the real user behavior observed in the field. Therefore historical data of the StadtRAD system was compared to results of simulation runs and the parameter distributions for the agents were adapted until the behavior appeared sufficiently realistic. Afterwards the simulation studies were performed by altering the environment and observing how the bicycle usage changes.

The calibration as well as the studies themselves rely on the evaluation features of EnvSupport. By specifying data providers in the environment XML description, various results of the simulation (e.g. mean distance travelled, average number of bicycle checkouts per day) are automatically gathered during execution. Additionally specified data collectors consume the data and provide it to the developer, e.g. as graphical chart views, or export it to files for offline analysis.

The first study was a simple scenario analysis that investigated the effect of introducing an additional station in the StadtRAD network. The simulation allowed estimating the increase of bicycle usages that could be expected by introducing a new station at an important junction point with many train lines ("Schlump"). Most importantly, it could be verified, that the new station would not lead to significantly less bicycle use at other stations in the vicinity.

The second study was much more complex as it involved to comparison of different dispatching strategies. For this study an additional dispatcher agent was introduced, that performed a certain dispatching strategy by moving bicycles from overloaded stations to stations with few bicycles. Three strategies were analyzed that differed in the decision when to start moving bikes between stations. In the first strategy, the dispatcher would become active, when a station runs out of bikes. It would take a certain amount of bikes from the fullest station and transport it to the empty one. The second strategy is an adaptation of the first that introduced a threshold, i.e. already starting to transport bikes if their number drops below a certain value. The last strategy uses historic data of bicycle use and would transport bikes based on previously observed shortages (e.g. from the last day), regardless of the current situation. Simulation results showed that the threshold strategy performed best with respect to achieving the highest value of bicycle usage.

1.3.4 Summary

The EnvSupport extension allows defining the structure and behavior (domain) of an application environment in terms of objects as well as tasks and processes. Using avatars and actions, the interaction between the environment and the application agents can be clearly defined. This allows testing applications in virtual settings before real deployment. The visualization is further helpful for understanding application behavior either for teaching purposes or for debugging during application development. Furthermore, the visualization may also be part of the application, e.g. for games or training applications. The evaluation module allows flexible measuring of application performance by observing interesting application values and producing various outputs, such as dynamically updated graphical charts or data files for off-line analysis. In this respect, the evaluation module can e.g. be used for benchmarking alternative implementations of application components.

EnvSupport is currently implemented for local simulations only, even though the principles behind it are general enough to be applied for distributed simulations as well. This requires allowing remote interactions of agents with the environment space. One simple solution to this problem is to create a service interface for the environment and let the agent hosting the environment expose a provided service that the participating agents use to interact with it. Furthermore, to allow also remote observers the world and visualization data of the environment need to made accessible per remote service as well. It has to be noted that such a simple solution may have performance problems due to the high amount of data that needs to be transferred between clients and environment. To avoid this, more advanced but also complex schemes have to be taken into consideration, e.g. by letting

the clients perform partial calculation and rendering tasks on their own and synchronize with the environment only at specific rendezvous points.

1.4 Conclusion and Outlook

Simulation is a very interesting technique in combination with multi-agent systems. First, simulation studies may benefit from a multi-agent perspective as in scenarios with autonomous entities these can be adequately and individually modelled. Second, agent applications may profit from an upstream simulation analysis of specific application aspects before a real deployment is targeted. In the following the lessons learnt regarding simulation support for agent systems is summarized:

- A necessary key technique for supporting simulations is time control. It should be possible to choose the simulation mode that fits best to the simulation task to be performed, i.e. use real-time driven, time-driven or event-driven time advancement. For example if high efficient simulations are necessary due to long periods of time to be simulated or due to extraordinary complexity of the secario a fast-as-possible simulation execution is advantageous.
- An important part of simulations is the simulation environment, which is many cases requires much attention and effort to be built. For this reason, specific support for developing simulation environments should be available. Simulation environments are useful for several reasons. First, they allow describing the boundary of the system and thus its external interface. Second, the environment can help understanding if a system works properly. Especially, visualizing the environment facilitates a better understanding of the system dynamics.

The guiding principle for simulation support consists in establishing simulation transparency, i.e. the application code should not be polluted with simulation specific aspects. This serves two purposes. On the one hand it enables code reuse, as the implementation of a simulation model that can later serve as basis for the implementation of the target system (with exception of the environment). Furthermore, if the simulation is used to test the system implementation, exact reuse of the code assures that no implementation details of a reimplementation, that would normally have to be created, cause malfunctions. On the other hand, no simulation specific programming language or environment needs to be learnt. Following this principle led to a non-invasive approach towards time as well as environment mechanism realization. Time control has been built in on infrastructure layer in order to hide timing aspects from the agents. The clock abstraction allows for keeping the agent code unaware of timing aspects. Different clocks are supplied which bring about the different simulation modes so that it can be determined at

runtime if the application should be executed time-driven, event-driven or real time. Also environment support has been designed to be an optional part of applications. Therefore, the platform supports a general extension mechanism that allows for creating custom functionalities of an agent. The EnvSupport has been designed to follow this extension mechanism and offers its own description model. In general EnvSupport cleanly separates the domain model from its visualization in order to be able to create different views for one application.

References

1. L. Braubach, A. Pokahr, W. Lamersdorf, K.-H. Krempels, and P.-O. Woelk. A generic time management service for distributed multi-agent systems. *Applied Artificial Intelligence*, 20(2-4):229–249, 2 2006.
2. N. Collier. RePast: An Extensible Framework for Agent Simulation. Working Paper, Social Science Research Computing, University of Chicago, 2001.
3. D. Frey, T. Stockheim, P.-O. Woelk, and R. Zimmermann. Integrated Multi-agent-based Supply Chain Management. In *Proceedings of the 12th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2003)*, pages 24–29. IEEE Computer Society, 2003.
4. K. Jander, L. Braubach, and A. Pokahr. Envsupport: A framework for developing virtual environments. In *Seventh International Workshop From Agent Theory to Agent Implementation (AT2AI-7)*. Austrian Society for Cybernetic Studies, 2010.
5. S. Kirn, C. Heine, R. Herrler, and K.-H. Krempels. Agent.Hospital - agent-based open framework for clinical applications. In G. Kotsis and S. Reddy, editors, *Proceedings of the 12th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2003)*, pages 36–41. CS Press, 2003.
6. F. Klügl and F. Puppe. The Multi-Agent Simulation Environment SeSAm. In H. Kleine Büning, editor, *Proceedings of SiWiS'98: Simulation in Wissensbasierten Systemen*, 1998. Technical Report tr-ri-98-194, Universität Paderborn.
7. A. Pokahr, L. Braubach, J. Sudeikat, W. Renz, and W. Lamersdorf. Simulation and implementation of logistics systems based on agent technology. In T. Blecker, W. Kersten, and C. Gertz, editors, *Hamburg International Conference on Logistics (HICL'08): Logistics Networks and Nodes*, pages 291–308. Erich Schmidt Verlag, 2008.
8. D. Reichelt. Agentenbasierte Simulation von Fahrradverleihsystemen . Bachelorarbeit, Distributed Systems and Information Systems Group, Computer Science Department, University of Hamburg, December 2011. (in German).
9. A. Ricci, M. Viroli, and A. Omicini. The A&A programming model and technology for developing agent environments in MAS. In Mehdi Dastani, Amal El Fallah Seghrouchni, Alessandro Ricci, and Michael Winikoff, editors, *Programming Multi-Agent Systems, 5th International Workshop (ProMAS 2007)*, pages 89–106. Springer Berlin / Heidelberg, 2007.
10. J. Sudeikat, L. Braubach, A. Pokahr, W. Renz, and W. Lamersdorf. Systematically engineering self–organizing systems: The sodekovs approach. In M. Wagner, D. Hogrefe, K. Geihs, and K. David, editors, *Proceedings des Workshops über Selbstorganisierende, adaptive, kontextsensitive verteilte Systeme (KIVS 2009)*, page 12. Electronic Communications of the EASST, 3 2009.
11. J. Sudeikat and W. Renz. Building complex adaptive systems: On engineering self-organizing multi-agent systems. In G. Hunter, editor, *Strategic Information Systems:*

Concepts, Methodologies, Tools, and Applications, pages 767–787. IGI Publishing, 2 2010.

12. T. Warden, R. Porzel, J. D. Gehrke, O. Herzog, H. Langer, and R. Malaka. Towards ontology-based multiagent simulations: The plasma approach. In *In Proceedings of the 24th European Conference on Modelling and Simulation (ECMS 2010)*, pages 50–56, 2010.
13. D. Weyns, A. Omicini, and J. Odell. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, 2007.
14. A. Zöller, L. Braubach, A. Pokahr, T. Paulussen F. Rothlauf, W. Lamersdorf, and A. Heinzl. Evaluation of a multi-agent system for hospital patient scheduling. *International Transactions on Systems Science and Applications (ITSSA)*, 1:375–380, 2006.