A Simulation Framework to Analyze Knowledge Exchange Strategies in Distributed Self-adaptive Systems

Christopher Werner, Sebastian Götz, and Uwe Aßmann

Technische Universität Dresden
Software Technology Group, Dresden, Germany
{christopher.werner,uwe.assmann}@tu-dresden.de
sebastian.goetz@acm.org
https://tu-dresden.de/ing/informatik/smt/st

Abstract. Distributed self-adaptive systems are on the verge of becoming an essential part of personal life. They consist of connected subsystems which work together to serve a higher goal. The highly distributed and self-organizing nature of the resulting system poses the need for runtime management. Here, a particular problem of interest is to determine an optimal approach for knowledge exchange between the constituent systems. In the context of multi-agent systems, a lot of theoretical work investigating this problem has been conducted over the last decades, showing that different approaches are optimal in different situations. Thus, to actually build such systems, the insights from existing theoretical approaches need to be validated against concrete situations. For this purpose, we present a simulation platform to test different knowledge exchange strategies in a test scenario. We used Siafu as a basis. The described platform enables the user to easily specify new types of constituent systems and their communication mechanisms. Moreover, the platform offers several integrated metrics, which are easily extensible. We evaluate the applicability of the platform using three different collaboration scenarios.

Keywords: Distributed self-adaptive systems, simulation, multi-agent systems, role-oriented programming.

1 Introduction

Mobile devices with the ability to sense and adapt themselves to a changing environment are getting omnipresent in our society. Among them are smart watches, fitness trackers, (cleaning) robots, and wearables, to name but a few. To fully utilize these devices, they need to be integrated, which leads to complex systems, where different subsystems have to communicate with one another and establish different kinds of collaborations on the fly. For example, in a future (smart) office with two or more cleaning robots, where some are specialized on dry cleaning and others on wet cleaning, there is a need for coordination

among them. The highly distributed nature of such systems demands runtime management of each individual subsystem and optimization of the system as a whole to assure user-specified higher goals.

A promising approach for such systems is to develop them as self-adaptive systems by using self-organization and self-optimization techniques [13]. Such systems can be characterized as distributed self-adaptive systems (D-SAS). What implies, that each subsystem is autonomous and makes decisions based on its own knowledge. But, in addition, each subsystem has to take into account the invariants of the system as a whole and its environment. To ensure that actions of one subsystem do not negatively impact another subsystem, a coordination mechanism is required. To realize this coordination, a spectrum of approaches can be used. At one extreme, a single, central system can be used to collect the knowledge from all subsystems and to influence them. At the other extreme, all systems directly exchange their knowledge in a peer-to-peer manner and, thereby are enabled to reason about the effects of their decisions on other subsystems. An approach for systems with vast amounts of subsystems is a hierarchic coordination, where the children of a node are coordinated by their parent node.

In either case, the central research question, which we partially addressed in our previous work [8], is: which knowledge distribution strategy is the best for the current collaboration in a D-SAS.

The answer to this question heavily depends on (a) the environment, (b) the characteristics of constituting self-adaptive systems, and (c) the goals imposed on the system as a whole. Among these parameters, the trade-off to negotiate can be characterized as follows. The less knowledge individual subsystems exchange with each other, the less are the implied costs, but the higher is the probability of them to make decisions having a negative effect on other subsystems. To practically decide which knowledge exchange strategy is the best, a more detailed specification of the costs, which are usually domain-specific, is required. For the case study, presented in this paper, we used the following properties as cost/quality:

- **Q1 Performance.** The performance of the system will decrease if there is unnecessary knowledge exchange. This means more network communication and more computational work for the system.
- **Q2 Real-Time.** Systems can have time restrictions (deadlines), which are not to be missed.
- **Q3 Energy Consumption.** The more knowledge is exchanged, the more energy is spent on it, but the capacity of the participating subsystems is often restricted.
- **Q4 Memory Consumption.** Small devices are often limited in terms of their memory. Thus, gathering all available knowledge on a single device can be impossible.
- **Q5 Privacy.** If the system comprises devices from different owners, policies to prevent unauthorized knowledge exchange are required.

The goal of this paper is to enable system developers to identify the optimal strategy before deploying it and researchers to investigate novel algorithms and approaches for knowledge exchange in D-SAS.

The research questions addressed in this paper are:

- **RQ1:** How to analyze the quality of a knowledge exchange strategy in a D-SAS?
- **RQ2:** How to identify which knowledge exchange strategy in a D-SAS is the best among several alternatives?

Therefore, in this paper we present a reusable simulation framework called SAKE which allows to test different strategies in concrete test scenarios. The framework is easily extensible w.r.t. new system types, knowledge exchange strategies and cost/quality characterizations (i.e., metrics). The simulator framework uses Siafu [11] as a basis and is accessible on GitHub[1].

As evaluation, we show three experiments made using the simulation framework, where different knowledge exchange strategies for a fleet of specialized cleaning agents are investigated on different maps.

The remainder of this paper is structured as follows. The next section provides an in-depth discussion on the concepts provided by the simulation framework, its extensibility, and its metrics. The three case studies we conducted are presented in Section 3. We demarcate our approach from related work in Section 4. Finally, in Section 5, we conclude the paper and discuss possible lines of future work.

2 A Simulation Infrastructure for Knowledge Exchange Strategies

In this section, we introduce Siafu (cf. Section 2.1) which is used as time simulation framework and front-end for SAKE and we provide an overview of our proposed simulation framework in terms of its key concepts (cf. Section 2.2), its metrics (cf. Section 2.3), and its extensibility (cf. Section 2.4).

2.1 Siafu Simulator

The open source context simulator Siafu [11] acts as user interface and controller part for each SAKE simulation. Siafu is implemented in Java and works on two dimensional maps and models a few agents and places which are located in the map. The concept of the model is shown in Figure 1 and contains in the middle the *World* which holds several *Agents*, *Places*, and *Overlays*. The *Agents* and the *Places* have a *Position* to specify the location in the map, whereas the *Overlays* hold information about the context of the world like the temperature, sun intensity, and dirt level. To create the start information, Siafu reads the data from images, which must have the same size as the map. The input data is

[1] `http://github.com/sgoetz-tud/sake`

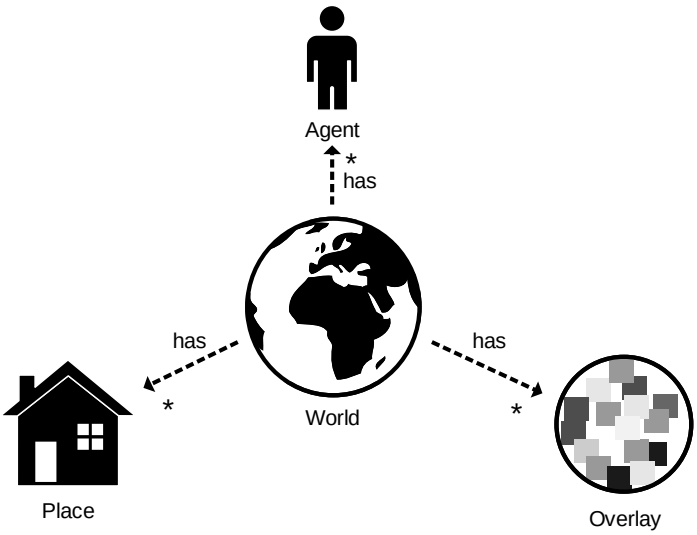

Fig. 1. Concept-Model from Siafu

transferred over an interface to an external simulation. This part represents the connection between Siafu and SAKE and can connect also other simulations.

The advantages of Siafu for our approach are the open Java source code, the fast start up time, and the good documentation and examples, but Siafu besides has some disadvantages. It is not possible to start more than one simulation at the same time only the opportunity to start the entire tool more than one time exists. In addition, the last commit in the GitHub repository of Siafu was two years ago which means it is not under development anymore. As well Siafu uses Eclipse SWT as user interface environment which only support Mac Os cocao and all Linux and Microsoft operating systems.

For our evaluation, it was important to run multiple simulation in parallel, which means each simulation run in one thread. Therefore, we extend Siafu to get a configuration file as input and create as much simulations in parallel as configured.

2.2 Concepts

The central concept of the SAKE framework is depicted in Figure 2. All system constituents have a physical and a virtual part. The physical part comprises an ensemble of hardware components (e.g., computers or engines). The virtual part of the system comprises its roles, goals, and behaviors. Each agent plays different roles aiming to reach specified goals using available behaviors, which, in turn, are determined by the present hardware components.

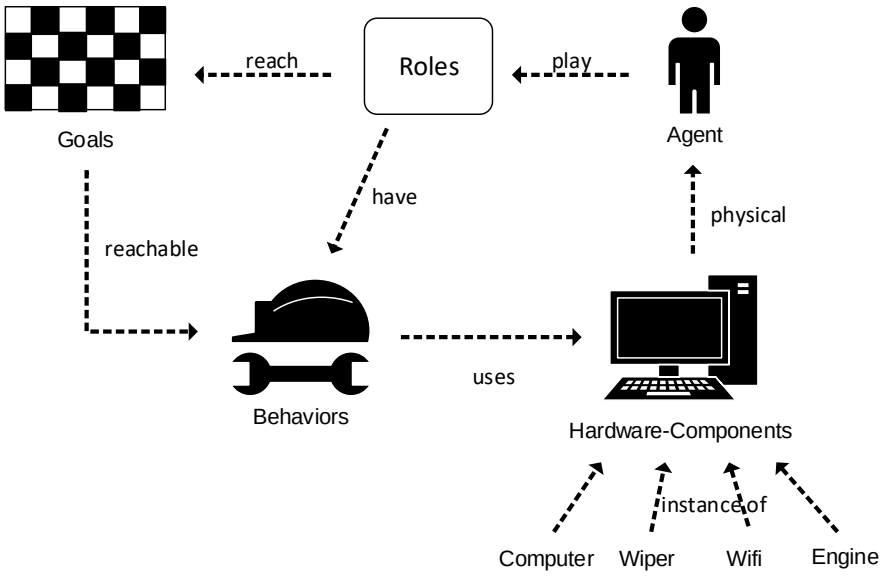

Fig. 2. Concept of the Agent Representation

Figure 3 depicts an overview of the framework as UML class diagram. At the top level, the framework is comprised of four packages: `SiafuSimulator`, `AgentFactory`, `AgentCore` and `GoalBehaviorsHardware`.

The *SiafuSimulator* package shows, which classes of the Siafu simulator we reuse and specialize to create a framework focusing on evaluating knowledge exchange strategies. We reuse the `World` class, which represents the world model of the Siafu simulator, i.e., denotes the common global environment of all system constituents and is the same for all `SiafuAgents`. We specialize the `Agent` and `BaseAgentModel` classes, where the second comprises a method to create the former and specifies how the simulation proceeds iteration-wise.

The *AgentFactory* package is represented by three classes in the UML class diagram: `AgentModel`, `AgentFactory`, and `SiafuAgent`. The first is a specialization of Siafu's `BaseAgentModel` and realizes the creation of agents using the factory method design patter [7, pp. 107].

Next, the *AgentCore* package applies the role-object pattern [1] to enable a dynamic management of agent goals and related behaviors. The pattern comprises the abstract `Agent` class and the two classes `AgentCore` and `AgentRole`, where the first contains the implementations of management methods to add and remove roles and the second only delegates to the former. Besides such management methods, the `Agent` class defines a property `newInformation` to indicate, whether the agent has collected information since its last exchange. The `AgentCore` class specifies, e.g., a property `name`, `localModel`, and `shutdown` for the agent and contains the only accumulator component of an agent.

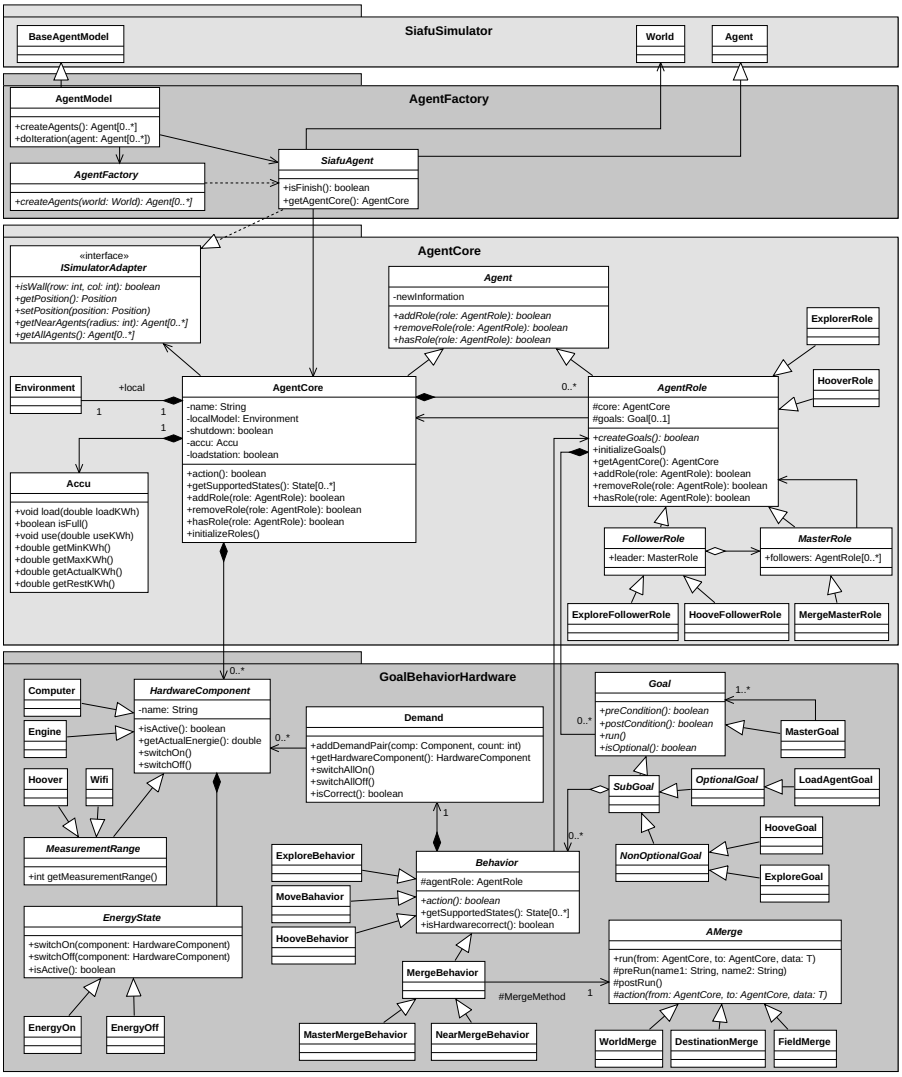

Fig. 3. Package Class Diagram of the Simulator

The Simulator is constructed to work with location data relying on maps to create a distance value between two agents. The *AgentCore* package describes the main structure and the collaborations of all agents, where each agent plays different roles, e.g., master or follower. These two roles and thereof specialized roles create a hierarchical structure of the agent (they can be master and follower at the same time).

The last package is the *GoalBehaviorHardware* package which contains the modeling of `Goal`s, `Behaviour`s, and `HardwareComponent`s. The different goals will be added to the roles of each agent, if it has appropriate hardware-components. In every time step, the action method of each goal is executed once. When the post condition is met, the goal is achieved and deletes itself. With the composite pattern, it is possible to create a hierarchical structure of goals for each role. For the representation of a load-station it must be possible to have goals, which will never be achieved, but also are not hindering the system as a whole to finish. Therefore, we distinguish between `OptionalGoal` and `NonOptionalGoal`. With this structure, it is possible to create elements which are relevant for the end of a scenario and elements which must work without affecting the highest goal. The structure with some example goals is illustrated in Figure 3 and further described in [16].

2.3 Metrics

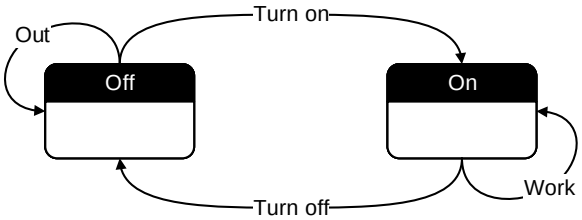

Fig. 4. Energy States for Hardware Components

For every running example, we collect four different metrics–time, data-exchange, memory consumption, and energy–to evaluate a strategy. The time is measured on the one hand as the computing time for each agent. On the other hand, the simulator saves the number of time steps an agent performs to reach a specific goal. Energy consumption is measured for each component an agent consists of. Each hardware element has different energy states which represent working states. In Figure 4, two exemplary energy states are depicted. Another example was shown in [6], where a bigger energy state model was used with respect to the accumulator state representing the probability of working success of the hardware-component. In our example, we only use the *on* and *off* state to

represent a working component. Notably, besides energy states, the transitions between states are also annotated with their respective energy demand. With this strategy every hardware-component can be represent in the simulator.

For the data exchange metric, the whole communication is monitored to abstract on the network load. To create a representative view of data exchange metrics, the number of exchanged elements and the data-stream itself are stored in a specific optimal, save format. The memory consumption is monitored at the end of each simulation for every agent. This concept works under the assumption that each agent has its maximal local knowledge and the end of a simulation and did not delete anything before which would falsify the results.

This four metrics help to measure the cost quality properties as defined in Section 1. The **Performance (Q1)** is measured with the computing time and data exchange metrics. Furthermore, the **Real-Time (Q2)** arise from the number of time steps measured for each complete simulation and the **Energy Consumption (Q3)** and **Memory Consumption (Q4)** results from their corresponding metrics energy and memory consumption. The **Privacy (Q5)** property is not measurable by the SAKE simulator, but the more data an agent want to exchange the less is the privacy of this agent and the entire system.

2.4 Extensibility

To enable reuse, it must be possible to easily extend the simulator and to test different novel exchange strategies. Our framework provides different points to modify and add new strategies. The first point is the factory package, which creates every agent with its own hardware-components and roles. In the *Agent-Core* package, new agent roles can be implemented to specify different collaboration structures, goals, and responsibilities for the agents. The goals, behaviors, and hardware-components are modified like the agent roles, so it is possible to add new functions and components. The energy states show currently only the `EnergyOn` and `EnergyOff` states and are modeled with the State-pattern. The State-pattern offers an extension point to create different new states like idle, busy or standby.

Each agent saves in every time step the metric values of the simulation and adds them to an evaluation file. For that, the evaluation function is called every time step and can easily be modified for different metric values. After the complete simulation run, the simulator actual saves every JSON result value in a document. This documents can be used to load these value-list in a new tool and create diagrams or tables for analysis.

3 Evaluation

We illustrate the results of our simulator based on a specific running example. Three different strategies are used to test and get comparable values for the evaluation. The easiest way is a complete collaboration between each agent. This mean that, if an agent meet another agent, he will give him his complete

local model of the world. To get a higher hierarchical collaboration we use in the other strategies a master agent which handles the communication and makes work decisions.

The environment of the test scenario includes different parameters, e.g., the size and the nature of the environment change in a different context. Otherwise the dependencies of agent types influence the results of each test case. To get representative results in the test cases we run them five times and use the average values for the depicted diagrams. This five runs are necessary, because if the agent has two new destinations with the same distance, it decides randomly where to drive next.

The first part of this section contains the running example with different start parameters and agent dependencies and the used strategies. The next two parts show the evaluation results based on the time measurement and the knowledge exchange of a strategy for one environment and the last subsection gives an overview of our insights.

3.1 Running Example

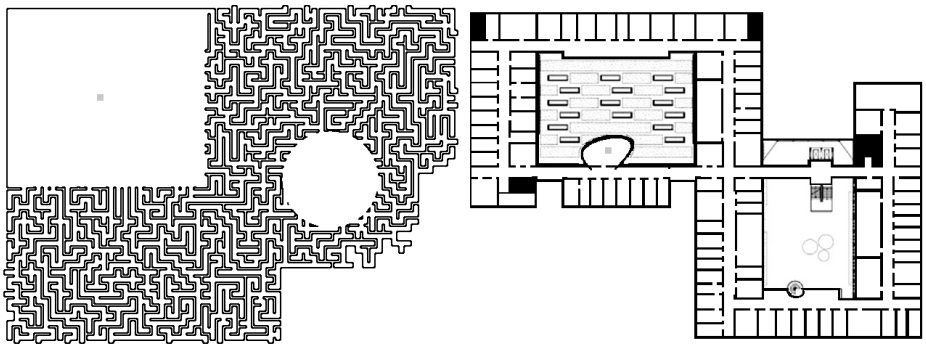

Fig. 5. (left) Labyrinth from Siafu [11] and (right) Computer Science Faculty

The running example is an office cleaning scenario for one floor. In order to not interfere with office work the cleaning of office spaces has to take place outside working hours. This creates one important requirement for cleaning robots that are used in this scenario which is to satisfy deadlines in various different working spaces. The easiest way is to use one agent, which must do it alone, but it is possible, that he does not finish in time. In that care, it is important that different agents share the work and communicate about the areas they already cleaned. For that, we create three different strategies:

- **C1: Complete Collaboration.** An agent exchanges his complete model with a near agent, but with a time delay to avoid exchanges in very short periods.

- **C2: Communication with Master.** A master coordinates and handles the communication with near agents. The master is located at the load-station and exchanges the models when the agents are loading. Every agent computes his drive destinations based on its own information. This approach reduces the locally needed memory and minimizes the knowledge exchange.
- **C3: Communication and Coordination with Master.** The master always communicates with the agents and tells them what to do. The working agents need a communication infrastructure to stay connected with the master, but also less logic.

In Figure 5 we show two of three different maps on which we tested the three-strategies. The gray points in the maps represent the load station and with it the location of the master agent. The first map represents a labyrinth which is predefined in Siafu [11]. This is the biggest map we use with a lot of dead ends and narrow ways. The second map represents a floor of the computer science faculty of the TU Dresden. It gives the best real world example with little rooms and big corridors which is why we only show the results from this map in the following subsections. At last we also used a quadratic hall map, which is not depicted in Figure 5. The results of the other maps and agent types are shown in [16]. In the maps, each white pixel represents a point which should be cleaned and the black walls show the borders.

To create dependencies between the agents, we used three different types of them. Before, e.g., a hoover (vacuum) agent can clean the world he needs a map of the area. To realize that a hierarchical structure is used. A master agent communicates with explore agents about the world and exchanges his knowledge with the hoover agents. This three stage pyramid visualizes the agent dependencies. The three steps of the cleaning process are now (a) create the map with explore agents, (b) vacuum the area with (hoover) agents and (c) wipe the environment after vacuuming it with (wiper) agents. This step dependencies mean that every agent needs parts of the information from an agent one step before to start with his work. This part creates waiting periods for agents from a higher level in the hierarchy and has an influence on the deadline.

The number of agents influences the deadline, too. Because of that, we first increase the number of explore agents from one to ten and then add hoover agents and increase them from one to ten, too. This creates a usable set of data for analysis.

3.2 Time Measurement

In this subsection, we present the time measurement result from the faculty map and three strategies. In Figure 6 the average number of time steps is shown with first one to ten explore agents and then additionally one to ten hoover agents. A time step is one iteration run in the simulator, where every agent runs one time all its behaviors. The deviations in the diagram come from the average values of five runs. Each agent searches randomly the next nearest destination to work. This background strategy creates the different results in the strategies.

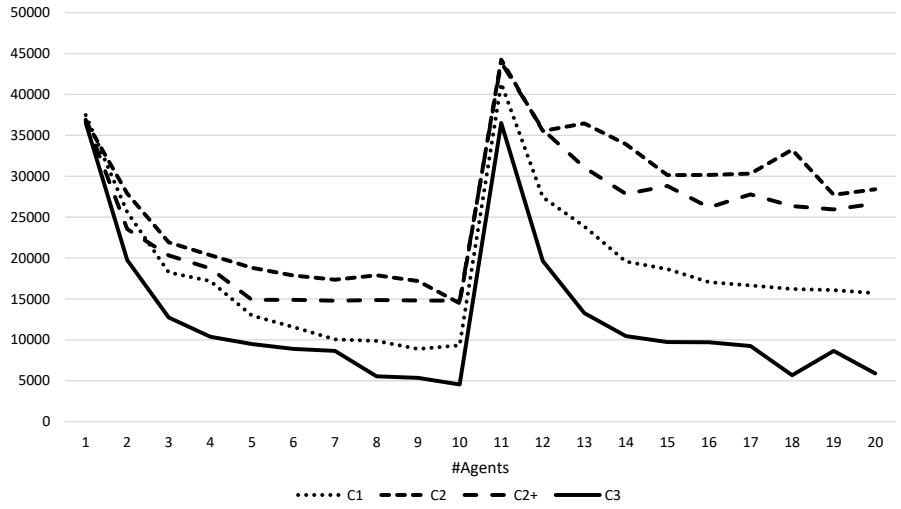

Fig. 6. Average Number of Timesteps in the Faculty map.

In Figure 6 one time step is the same like one second in real time. From the tenth to the eleventh agent a big step arises because of the incoming agent dependency.

As we can see, C3 is the optimal strategy, because the master always has all information and can make optimal decisions with complete knowledge every time. In C2, the step is always there and cannot be removed with adding agents. The strategy that the master gets the information, when one agent comes back to the load-station, always creates such a step for new types of agents. Also, the extended C2+ has such a stage. In this strategy, the master defines the first destination after loading to spread around all agents. C1 can also not remove the stage by adding new agent types. After five agents of a type the time savings will get very low. In addition, C2+ gets better and C1 worse in bigger maps than in smaller ones.

3.3 Knowledge Exchange

In this part, a short overview about the knowledge exchange results is given. In Figure 7 the results of a test run are shown for a small-quadratic hall map. The diagram looks the same as for the faculty map only with less accessible areas. The y-axis shows the number of elements in one run, which are exchanged between all agents in logarithmic scale. In this diagram one value is either an integer, string or double value, because all primitive data elements count as one element. This approach only takes care of the real exchanged knowledge without any optimization on input and output agent strategies. C1 has the most exchanged elements because the agents make a total exchange every time they meet. The un-optimized C2 has five times less data exchange than C1 and the

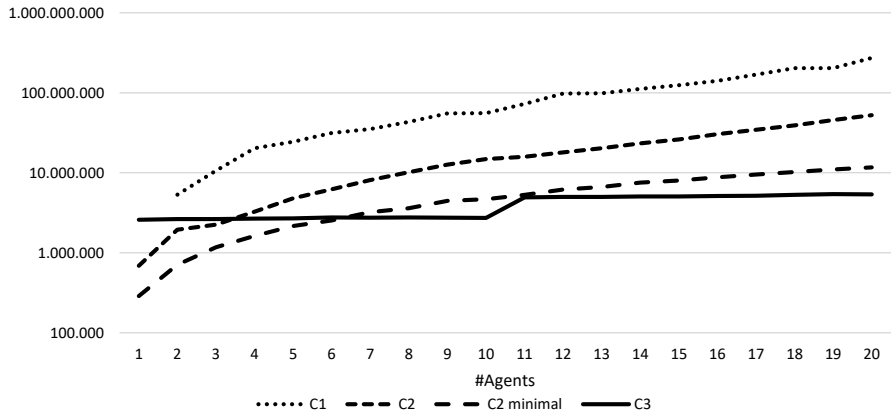

Fig. 7. Complete Knowledge Exchange Values.

optimized variant (C2 minimal) has ten to twenty times less data-exchange elements depending on the number of agents. The optimized option uses internal timestamps and states to reduce the redundant exchange elements. This optimization raises the memory consumption on the working agents. C3 has the least knowledge exchange, because the master gets in time the new information from each agent and only has to send new destinations. With more agents, C3 has the best results in w.r.t. the overall knowledge exchange.

3.4 Results

In the complete evaluation results, we find different dependencies between input parameters and the final outcome. With the number of agents and the choice of a strategy the results can be changed in different ways. If the number of agents rises, the overall time decreases, but nevertheless the energy consumption rises, too. This means that the influence of reducing time with more agents is smaller than the rising energy consumption per time step. The diagrams with the energy consumption results are presented in [16].

The three maps with their different characteristics have influence on the number of meetings, because large open spaces increases the probability of meetings. In a labyrinth with narrow ways an agent rarely meets others and cannot exchange his knowledge. In bigger maps the strategy, where the master spreads the agents, is faster because the probability of same working areas decreases.

The evaluation results of the different strategies show, that a master reduces the amount of data, which must be exchanged between all agents and creates an interface for human interaction. In all test cases, we found out, that in a perfect world the control from a master returns the best results. The requirements for this C3 however are too far from reality. It is likely, that this configuration will produce different results in a real-world case study. Strategies with a master

show that the more control options the master has, the faster the run is, but the more configurations to test. A master is also every time a bottle neck in the infrastructure which could be removed with more masters or combinations of different strategies.

4 Related Work

The problem of a good simulation framework is that there must be a lot of possibilities to modify the simulator for new test case and give evaluation information as much and detailed output. There are existing much simulators for different specific scenarios. For example the SUMO [2] simulator simulate traffic in an urban area. SUMO can be used to test traffic light control algorithms to get the best light control system for a city. It takes therefore respect to traffic tips which can be controlled with characteristics. UdelModels [10] is another simulation framework for urban networks. It take care on realistic propagation and gives a user interface for city creation. The OMNeT++ [15] simulation sweet includes different tool and simulates network protocols in varying areas.

There are although other simulators like Siafu for example JAS [14]. JAS is implemented in JAVA and gets his complete functional scope from third party libraries. It represents agents in components and brings a variety of collections from components and rules for the simulation, but JAS do not present any new versions since 2006.

For the knowledge exchange different strategies could be used. For example the FIPA [12] produces software specifications for multi-agent systems like communication protocols to maximize the compatibility of MAS. JADE [3] for example is an platform for peer-to-peer agent based applications. It describes a middleware which uses the FIPA specifications for the communication between agents. Thereby, it provide a graphic tool and facilitate the troubleshooting and deployment phase of the system. The platform is implemented in JAVA and could be used to realize different kinds of agent architectures. In the background for the representation of a agent it uses containers. The commercial tool is JACK [9] which is although implemented in java. JACK is developed from the Agent Oriented Software Pty, Ltd. (AOS) and is a progression of the Procedural Reasoning System (PRS) and the Distributed Multi-Agent Reasoning System (dMARS). As JADE it helps to create MAS. Every agent works in JACK in accordance with the BDI (Belief, Desire and Intentions) principle, which say that every agent can be described with its goals, his knowledge and his social skills and acts from the environmental input.

For our simulator we use only a simple information exchange based on the real data objects. To get a some various exchange strategies we look on the paper of Götz et al. [8] where three different strategies are mentioned. This are the total exchange strategy where all agents collaborate with each other and exchange there complete knowledge. Then the partial complete method where each subsystem exchange his complete knowledge with his direct collaborators and the third strategy the partial-subset where the agents only change part

of their own knowledge with her direct collaborators. This strategies give the template for our strategies and implementations.

Knowledge exchange is important in all kinds of MAS and is therefore often used in different ways. For example DEECo [5], SeSaMe [4] and DECIDE [6] are frameworks to create multi agent systems. DEECo is self named as an ensemble-based component system where an ensemble represents dynamic binding of a set of components and thus determines their composition and interaction. The ensemble component describes the collaboration and data connection. Although when only some data is used in the other component it proactive share all his information. SeSaMe coordinates distributed components in various selforganizing inter-composed groups based on the types of roles they can play and make a direct interaction between the supervisors and followers. DECIDE splits the control-loops on many nodes of a distributed self-adaptive system. This gives more flexibility and reduces the single point error when a master node fail. The goal of DECIDE is to proof the system at runtime to guarantee the quality requirements of critical self-adaptive software.

5 Conclusion and Future Work

In this paper, we present a simulator for different kinds of Multi-Agent Systems and test it in one test scenario with three configurations on three different maps. For the simulator, it is important to change different start parameters to create a wide range of test cases and environments. To analyze the tested strategies, we demonstrated some predefined metrics to evaluate them and give the possibility to easily extend them. With our evaluation results, it was possible to find some basic correlation between the input parameters, the nature of the environment and the metric results. The simulator is suited to test new systems before the first real world implementation and to find a good strategy.

As future work, we plan to implement more knowledge exchange strategies and prove the results for one strategy in the real world. For a real world implementation and testing, it is important to have changeable environments, which not yet implemented and realized. This point will be arriving in office scenarios with new obstacles. In the real world, it is possible that elements fall down or crash and the system has to create a new structure. For this, a probability to simulate agent crashes or hardware-component failures should be considered.

Acknowledgments. This work has been funded by the German Research Foundation within the Collaborative Research Center 912 Highly Adaptive Energy-Efficient Computing and within the Research Training Group Role-based Software Infrastructures for continuous-context-sensitive Systems (GRK 1907).

References

1. D. Bäumer, D. Riehle, W. Siberski, and M. Wulf. The role-object pattern. In *Proceedings of the 4th Pattern Languages of Programming Conference (PLoP'97)*, 1997.

2. Michael Behrisch, Laura Bieker, Jakob Erdmann, and Daniel Krajzewicz. Sumo–simulation of urban mobility. In *The Third International Conference on Advances in System Simulation (SIMUL 2011), Barcelona, Spain*, 2011.
3. Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade–a fipa-compliant agent framework. In *Proceedings of PAAM*, volume 99, page 33. London, 1999.
4. Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In Ian Horrocks and James Hendler, editors, *The Semantic Web ISWC 2002*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68. Springer Berlin Heidelberg, 2002.
5. Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. Deeco: An ensemble-based component system. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, pages 81–90, New York, NY, USA, 2013. ACM.
6. Radu Calinescu, Simos Gerasimou, and Alec Banks. Self-adaptive software with decentralised control loops. In *Fundamental Approaches to Software Engineering*, pages 235–251. Springer, 2015.
7. Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.
8. Sebastian Götz, Ilias Gerostathopoulos, Filip Krikava, Adnan Shahzada, and Romina Spalazzese. Adaptive exchange of distributed partial models@run.time for highly dynamic systems. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2015.
9. Nick Howden, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. Jack intelligent agents-summary of an agent infrastructure. In *5th International conference on autonomous agents*, 2001.
10. Jonghyun Kim, Vinay Sridhara, and Stephan Bohacek. Realistic mobility simulation of urban mesh networks. *Ad Hoc Networks*, 7(2):411 – 430, 2009.
11. Miquel Martin and Petteri Nurmi. A generic large scale simulator for ubiquitous computing. In *Third Annual International Conference on Mobile and Ubiquitous Systems: Networking & Services, 2006 (MobiQuitous 2006)*, San Jose, California, USA, July 2006. IEEE Computer Society.
12. Stefan Poslad. Specifying protocols for multi-agent systems interaction. *ACM Trans. Auton. Adapt. Syst.*, 2(4):15, November 2007.
13. Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):14:1–14:42, May 2009.
14. Michele Sonnessa. Jas: Java agent-based simulation library, an open framework for algorithm-intensive simulations. *Industry and Labor Dynamics: The Agent-Based Computational Economics Approach, World Scientific, Singapore*, 2004.
15. András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
16. Christopher Werner. Adaptive knowledge exchange with distributed partial models@run.time. Master's thesis, Technische Universität Dresden, January 2016.