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

Christopher Werner
Software Technology Group
Technische Universität Dresden
Email: christopher.werner@tu-dresden.de

Sebastian Götz
Software Technology Group
Technische Universität Dresden
Email: sebastian.goetz@acm.org

Uwe Aßmann
Software Technology Group
Technische Universität Dresden
Email: uwe.assmann@tu-dresden.de

Abstract—Distributed self-adaptive systems are on the verge to become an essential part of personal life. They consists 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 case scenario. We used Siafu as a basis. The described platform enables 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.

I. 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 to dry cleaning and others to wet cleaning, there's a need for coordination among them. The highly distributed nature of such systems demands for 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 [6]. Such systems can be characterized as distributed self-adaptive systems (D-SAS). That 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 respectively. At the other extreme, all systems directly exchange their knowledge in a peer-to-peer manner and, by this, 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 [4], 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 the 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:

- **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.
- **Real-Time.** Systems can have time restrictions (deadlines), which are not to be missed, in an environment.
- **Energy Consumption.** The more knowledge is to be exchanged, the more energy is spent for it, but the capacity of the participating subsystems is often restricted.
- **Memory Consumption.** Small devices are often limited in terms of their memory. Thus, gathering all available

knowledge on a single device can be impossible.

- **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 (a) system developers to identify the optimal strategy before deploying it and (b) researchers to investigate novel algorithms and approaches for knowledge exchange in D-SAS.

Therefor, in this paper, we present a simulation framework, 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 [5] 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 where investigated on different maps.

The remainder of this paper is structured as follows. The next section summarizes the research questions and contributions of this paper. Section III 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 IV. Finally, in Section V, we conclude the paper and discuss possible lines of future work.

II. RESEARCH QUESTIONS AND CONTRIBUTION

The aim of this paper is to enable (a) researchers, to investigate novel approaches of knowledge exchange in D-SAS and (b) to enable developers to identify the best strategy for their system before actually deploying it.

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?

The contribution of this paper is an easily reusable and extensible simulation framework, which allows to analyze knowledge exchange strategies in D-SAS and, by this, to identify, which is the best.

III. A SIMULATION INFRASTRUCTURE FOR KNOWLEDGE EXCHANGE STRATEGIES

In this section, we provide an overview of our proposed simulation framework in terms of its key concepts (cf. Section III-A), its metrics (cf. Section III-B) and its extensibility (cf. Section III-C).

A. Concepts

The central concepts of the framework are depicted in Figure 1. All system constituents have a physical and a virtual part. The physical part comprises an ensemble of hardware components (e.g., a computer or an engine). The virtual part of the system comprises its roles, goals and behaviors. Each
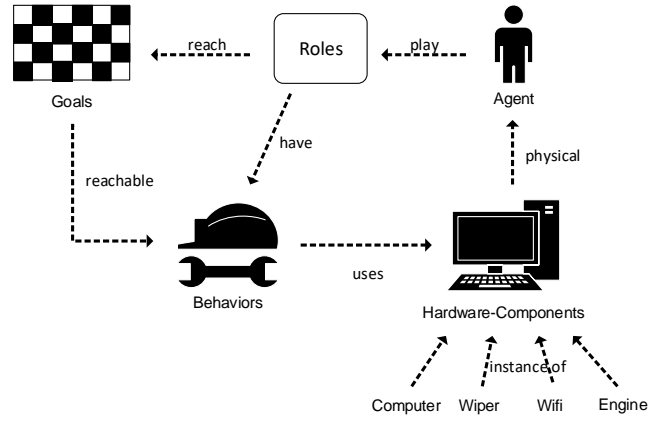
Fig. 1. Concept of the Agent Representation

agent plays different roles aiming to reach specified goals using available behaviors, which, in turn, are determined by the present hardware components.

Figure 2 depicts an overview of the framework as UML class diagram. At the top-most 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 [3, 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.

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

SiafuSimulator

BaseAgentModel

BaseMultiAgentModel

World

Agent

AgentFactory

AgentModel
+createAgents(): Agent[0..*]
+doIteration(agent: Agent[0..*])

AgentFactory
+createAgents(world: World): Agent[0..*]

SiafuAgent
+isFinish(): boolean
+getAgentCore(): AgentCore

AgentCore

«interface»
ISimulatorAdapter
+isWall(row: int, col: int): boolean
+getPosition(): Position
+setPosition(position: Position)
+getNearAgents(radius: int): Agent[0..*]
+getAllAgents(): Agent[0..*]

Agent
-newInformation
+addRole(role: AgentRole): boolean
+removeRole(role: AgentRole): boolean
+hasRole(role: AgentRole): boolean

ExplorerRole

HooverRole

Environment   +local

AgentCore
-name: String
-localModel: Environment
-shutdown: boolean
-accu: Accu
-loadstation: boolean
+action(): boolean
+getSupportedStates(): State[0..*]
+addRole(role: AgentRole): boolean
+removeRole(role: AgentRole): boolean
+hasRole(role: AgentRole): boolean
+initializeRoles()

AgentRole
#core: AgentCore
#goals: Goal[0..1]
+createGoals(): boolean
+initializeGoals()
+getAgentCore(): AgentCore
+addRole(role: AgentRole): boolean
+removeRole(role: AgentRole): boolean
+hasRole(role: AgentRole): boolean

Accu
+void load(double loadKWh)
+boolean isFull()
+void use(double useKWh)
+double getMinKWh()
+double getMaxKWh()
+double getActualKWh()
+double getRestKWh()

FollowerRole
+leader: MasterRole

MasterRole
+followers: AgentRole[0..*]

ExploreFollowerRole

HooveFollowerRole

MergeMasterRole

GoalBehaviorHardware

Computer

Engine

Hoover   **Wifi**

HardwareComponent
-name: String
+isActive(): boolean
+getActualEnergie(): double
+switchOn()
+switchOff()

Demand
+addDemandPair(comp: Component, count: int)
+getHardwareComponent(): HardwareComponent
+switchAllOn()
+switchAllOff()
+isCorrect(): boolean

Goal
+preCondition(): boolean
+postCondition(): boolean
+run()
+isOptional(): boolean

MasterGoal

MeasurementRange
+int getMeasurementRange()

ExploreBehavior

MoveBahavior

HooveBehavior

Behavior
#agentRole: AgentRole
+action(): boolean
+getSupportedStates(): State[0..*]
+isHardwarecorrect(): boolean

SubGoal

OptionalGoal

LoadAgentGoal

HooveGoal

NonOptionalGoal

ExploreGoal

EnergyState
+switchOn(component: HardwareComponent)
+switchOff(component: HardwareComponent)
+isActive(): boolean

EnergyOn   **EnergyOff**

MergeBehavior

MasterMergeBehavior

NearMergeBehavior

#MergeMethod

AMerge
+run(from: AgentCore, to: AgentCore, data: T)
#preRun(name1: String, name2: String)
#postRun()
#action(from: AgentCore, to: AgentCore, data: T)

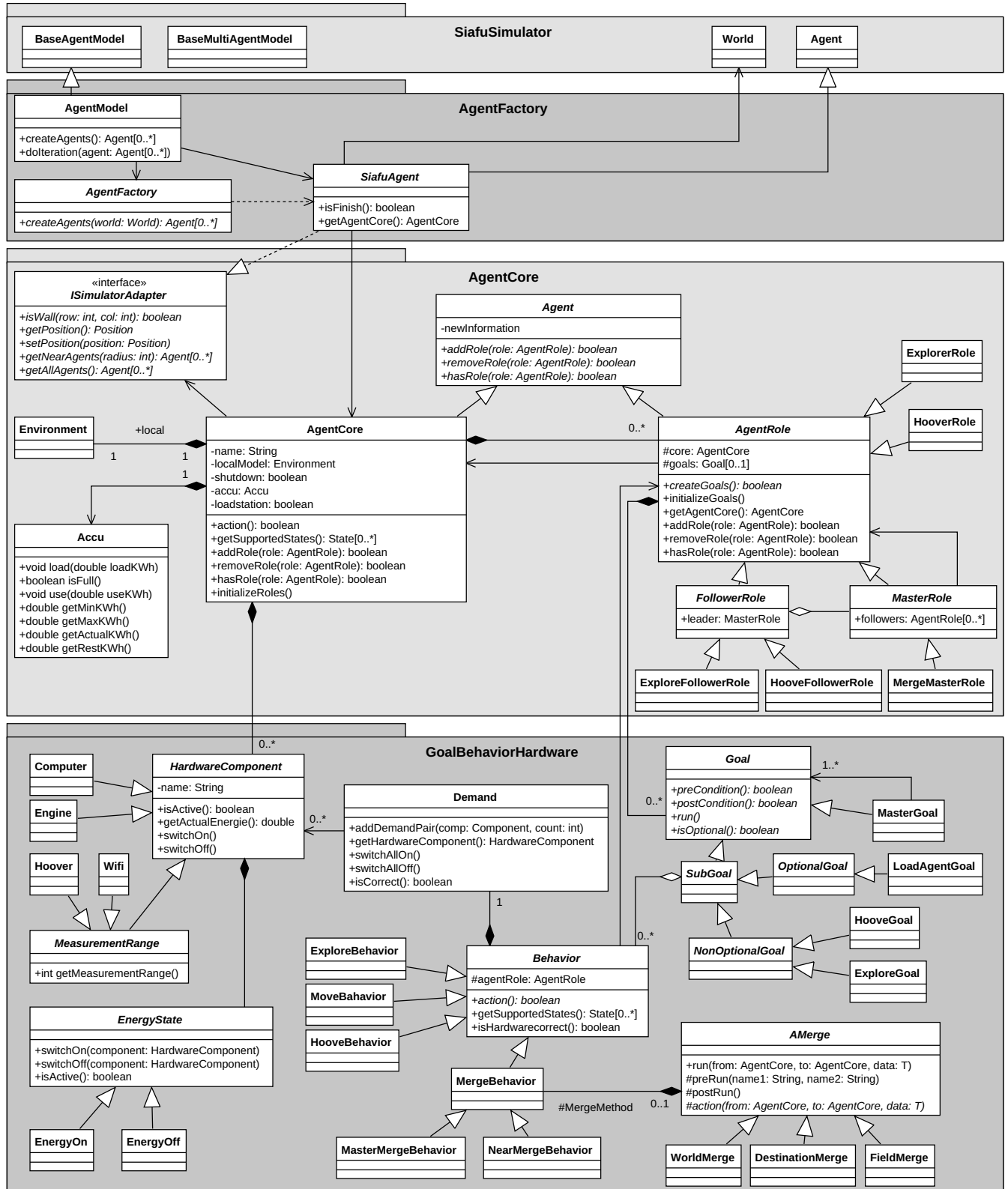**WorldMerge**   **DestinationMerge**   **FieldMerge**

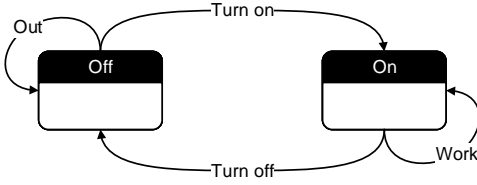Fig. 2. Package Class Diagram of the Simulator

Fig. 3. Energy States for Hardware Components

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 `Goals`, `Behaviours` and `HardwareComponents`. 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. Therefor, 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 2 and further described in [7].

B. Metrics

For every running example, we collect three different metrics–time, data-exchange and energy–to evaluate a strategy. On the one hand, the time is measured as system time which represents the computing time for each agent, doing his behaviors. 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 3 two exemplary energy states are depicted. Another example was shown in [2], 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 output elements and the data-stream itself are stored in a specific optimal, save format. For each optimization strategy

the output values can be adapted by a post-evaluation to be optimized for marshalling and unmarshalling.

C. Extensibility

To reuse the simulator, 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 *AgentCore* package, new agent roles can be implemented to specify different collaboration structures, goals and responsibilities for the agents. The goals, behaviors and hardware-components will be modified like the agent roles, so it is possible to add new functions and components. The energy states show actual only the `EnergyOn` and `EnergyOff` states, but this can also be extended by 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 saves every result value in a document. This documents can be used to load these values in a new tool and create concrete diagrams or tables to make an analysis.

IV. 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 case 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.

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 about our insights.

A. Running Example

Every company has different buildings with other structures and characteristics and must clean them every week. Therefor, different agents can be used to clean the building after the working hours. But they have every day a deadline, because they must be finished before the work starts in the morning. The easiest way is to use one agent, which must do it alone, but it is possible, that he does not finish in time.
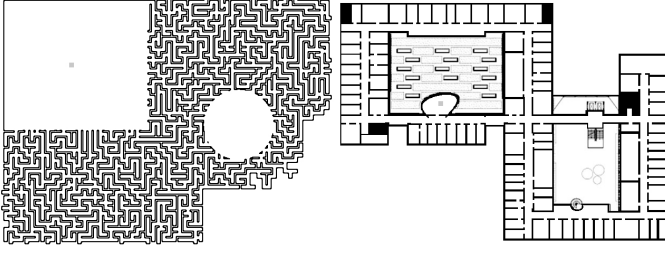
Fig. 4. ((left) Labyrinth from Siafu [5] and (right) Computer Science Faculty

Fig. 5. Average Number of Timesteps in the Faculty map.

Then it is important that different agents share the work and communicate about the areas they already cleaned. For that, we create three different strategies:

- **Complete Collaboration.** An agent exchanges his complete model with a near agent, but with a time delay to avoid exchanges in very short periods (Configuration 1).
- **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 creates his drive destinations based on its own information. This approach reduces the locally needed memory and minimizes the knowledge exchange (Configuration 2).
- **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 (Configuration 3).

In Figure 4 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 [5]. This is the biggest map we use with a lot of dead ends and small 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 that's 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 4. The results of the other maps and agent types are shown in [7]. 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) hoove the area with (hoover) agents and (c) wipe the environment after hooving 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
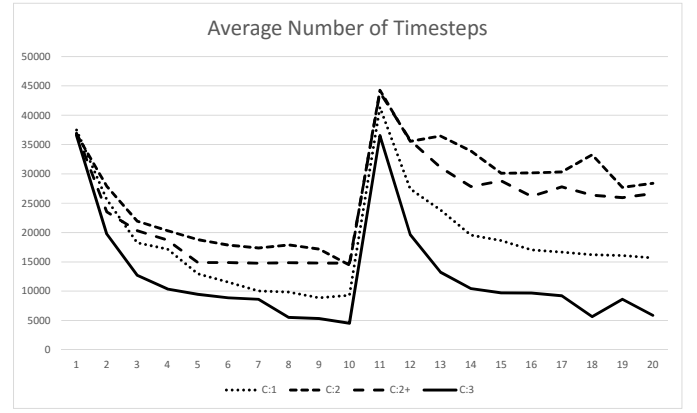
his work. This part creates waiting periods for agents from a higher level in the hierarchy and has an influence on the deadline.

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

B. Time Measurement

In this subsection, we present the time measurement result from the faculty map and three strategies. In Figure 5 the average number of time steps is shown with first one to ten explore agents and then plus 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. In Figure 5 one time step is the same like one second in real time. From the tenth to the eleventh agent a big stage arises because of the incoming agent dependency.

As we can see, configuration 3 is the optimal strategy, because the master always has all information and can make optimal decisions with complete knowledge every time. In configuration 2, the stage 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, creates always such a stage for new types of agents. Also, the extended configuration 2+ has such a stage. In this strategy, the master defines the first destination after loading to spread around all agents. Configuration 1 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, configuration 2+ gets better and configuration 1 worse in bigger maps than in smaller ones.

C. Knowledge Exchange

In this part, a short overview about the knowledge exchange results is given. In Figure 6 the results of a test run are shown
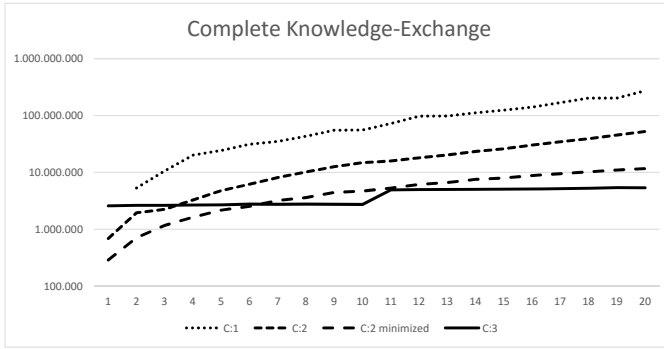
Fig. 6. Complete Knowledge Exchange Values.

for a small-quadratic hall map. The diagram looks the same as for the faculty map only with less knowledge. The y-axis shows the number of elements in one run, which are exchanged between all agents in logarithmic scale. In this diagram one element is either an integer, string or double value, because all primitive data elements count as one element. So, a special marshalling algorithm does not fake the results. Configuration 1 has the most exchanged elements because the agents make a total exchange every time they meet. The no-optimized configuration 2 has five times less data exchange than configuration 1 and the optimized variant 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 rises the memory consumption on the working agents. Configuration 3 has the lowest knowledge exchange, because the master gets in time the new information from each agent and only has to send new destinations. With more agents, configuration 3 has the best results in w.r.t. the overall knowledge exchange.

D. 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 rise, the overall time will decrease, but nevertheless the energy consumption will rise, too. This means that the influence of reducing time with more agents is smaller than the rising energy consumption per time step.

The three maps with their different characteristics have influence on the number of meetings, because in a big room two agents easily meet a lot. In a labyrinth with small 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 number of elements, which must be exchange 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 configuration (3) 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.

V. 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 parameter to create a wide range of test cases and environments. To analyse 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 proof 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.

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] 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.
[3] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.
[4] 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.
[5] 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.
[6] 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.
[7] Christopher Werner. Adaptive knowledge exchange with distributed partial models@run.time. Master's thesis, Technische Universität Dresden, January 2016.